Lecture 2: Virtualization and System Calls

CS 3281

Daniel Balasubramanian, Shervin Hajiamini

Overview

- Recap & questions/issues in setting up VM, discuss assignment
- Compilation basics in Linux – from source to executable programs
- Some concepts and definitions
- Introduction to system calls
- (Digression if we have time …)
- Summary & what's next

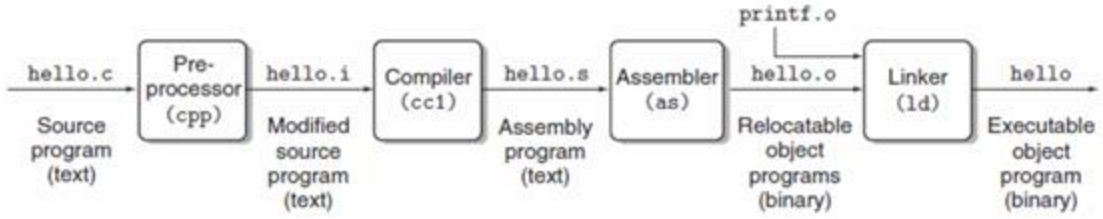
How do we get from source code to a running program?

From source code to running program (1)

- How do we go from text to a running program?
- We begin with a source file saved in a text file
  - This is a sequence of bits organized into 8-bit chunks called bytes. Each byte represents a character.
  - ASCII: unique byte sized integer value for each character. Source file stored as sequence of bytes. Files that consist exclusively of ASCII chars is a text file; binary file otherwise.
  - Everything is bits: the distinguishing difference is the context in which we view them

From source code to running program (2)

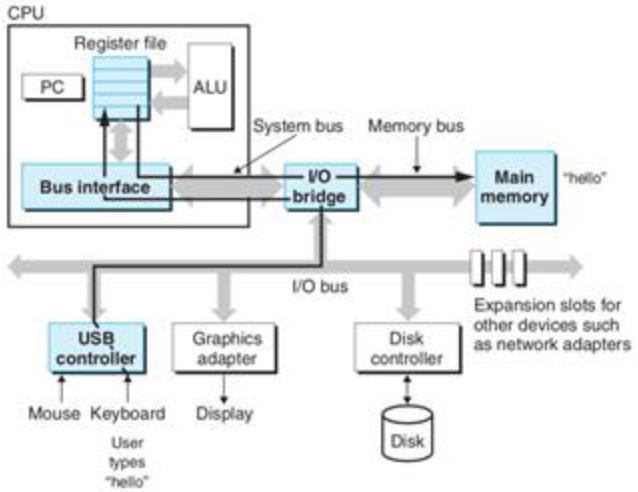
- Before it can run, must be translated into a sequence of machine-language instructions.
  - These are packaged into a form called an executable object program and stored as a binary disk file. Also called executable object files.
- Translation on Unix is done by a compiler driver.
- Do this individually with:
  - gcc -E hello.c -o hello.i // produces a modified source program
  - gcc -S hello.i // produces an assembly language program
  - gcc -c hello.s // produces binary object files
  - gcc hello.o // produces linked executable



From source code to running program (3)

- Running hello world
  - Initially, the shell is executing its own instructions
  - As we type “./hello” at the keyboard, the shell program reads each char into a register, then stores it in memory. We press enter and it knows we are finished typing the command.

Figure 1.5
Reading the `hello` command from the keyboard.



From source code to running program (4)

- Running hello world
  - The shell then loads the executable hello file by executing instructions that copy the code and data in the hello object file into main memory. DMA allows the data to go from disk to main memory directly.
  - Once the code and data are in memory, the OS switches to the hello process and begins executing its instructions.

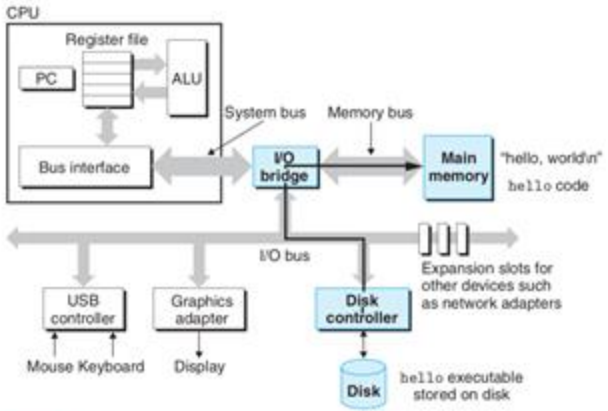
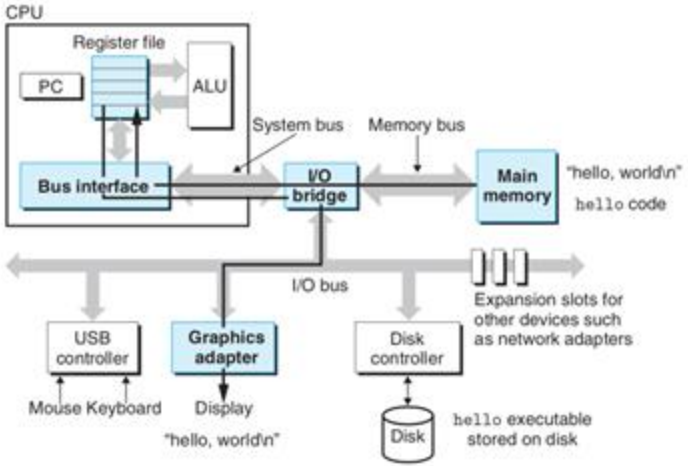


Figure 1.6 **Loading the executable from disk into main memory.**

From source code to running program (5)

- Running hello world
  - These instructions copy the data bytes “hello, world\n” from main memory to a register. From there they go to the display device where they are displayed on the screen.

Figure 1.7
Writing the output string from memory to the display.



What is an Operating System?

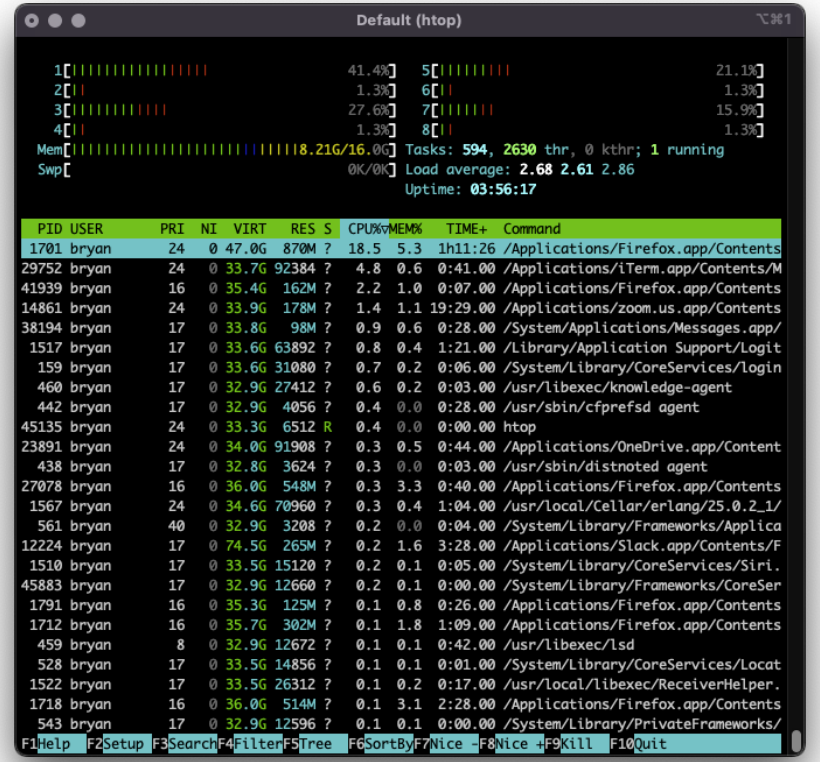
Operating system: definition

Operating systems

- Many definitions
  - A program that shares a computer among multiple programs and provides a more useful set of services than the hardware alone
  - A program that makes the hardware do “something useful”
- Accomplishes this by “virtualizing” the hardware
  - The OS makes it appear as though every process has the hardware all to itself
- Provides services through a *system call* interface
- The core part of the OS is called the *OS kernel*, or just *kernel*
- Layers:
  - Hardware: CPU, memory, disks
  - OS kernel: provides many services
  - Libraries: provide access to the above services
  - Applications: gcc (compiler), bash (shell), vi (editor), etc
- Key point: the operating system is itself a program!
  - But unlike other programs, it should have full access to all resources! How can we accomplish this?

Processes

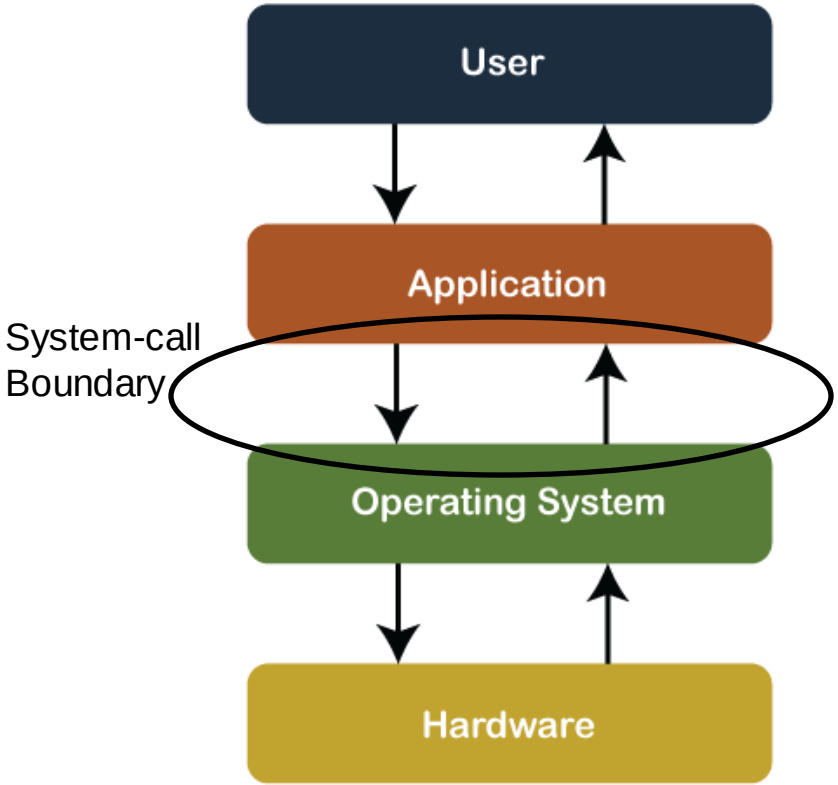
- A process is a running instance of a program
- It's the primary unit of isolation/abstraction provided by the OS
- The process is the abstraction that gives the illusion to a program that it has the machine to itself
  - Each process thinks it has the whole CPU
  - Each process thinks if has all memory to itself
- Serves several purposes:
  - Prevents a process X from corrupting or spying on a process Y
  - Prevents a process from corrupting the operating system itself



"ps" "top" or "htop" show all active processes

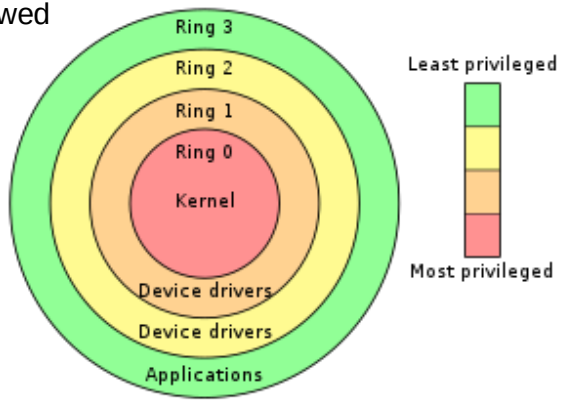
System calls

- The system call API is the interface that makes OS services available to user-level programs
- What are examples of these services?
  - Create a process
  - Wait for a process to terminate
  - Request memory
  - Tell me what my unique process identifier is
  - Read/write from/to a file
  - Send data over the network
- A process requests these service from the OS by making a *system call*
- Why does the OS provide these services?
  - User-level processes cannot be trusted to be non-malicious
  - Example: one malicious process could erase your entire hard drive!



Rings: Kernel mode vs user mode

- The kernel needs full access to all hardware and CPU instructions
- User-level processes (like Chrome or Firefox) should not be allowed full access to all hardware nor should they be able to execute all CPU instructions
- How should we enforce this? With hardware support!
  - A flag in a CPU register determines whether privileged instructions are allowed
  - On the x86 it's called the CPL (current privilege level)
    - The bottom two bits of the cs register.
    - CPL = 0 means kernel mode: privileged
    - CPL = 3 means user mode: no privilege
  - The CPL enforces isolation in several ways:
    - It guards access to the CS register
    - It checks every memory read/write
    - I/O port access
    - Control register access (like the EFLAGS register)



- In summary:
  - The CPL, represented by the bottom two bits of the cs register, determine whether instructions can access privileged hardware.
  - Only the kernel should be allowed to operate in the privileged mode.

Invoking a system call

- So how and when should the CPL change?
- On x86 it happens when a system call is invoked
  - The instruction used to invoke the system call sets CPL=0.
  - Execution jumps to a specific entry point in the kernel of the OS, which can then do further validation.
  - The system call return sets CPL=3 before returning to user code.

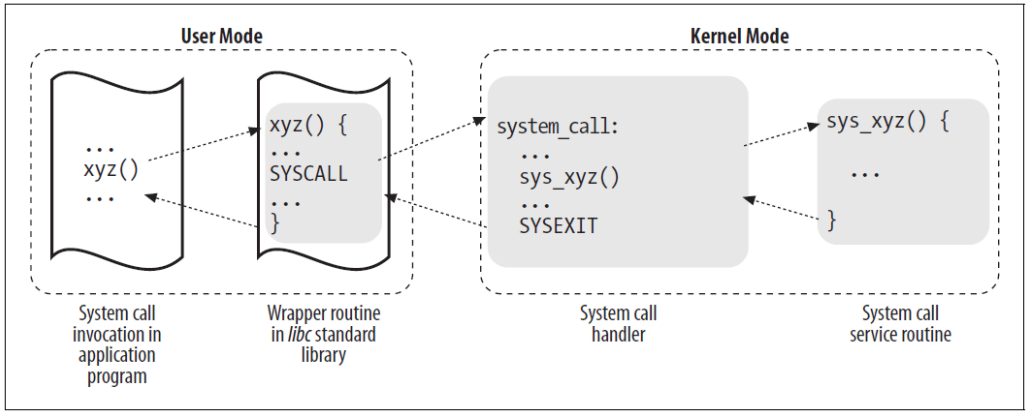


Figure 10-1. Invoking a system call

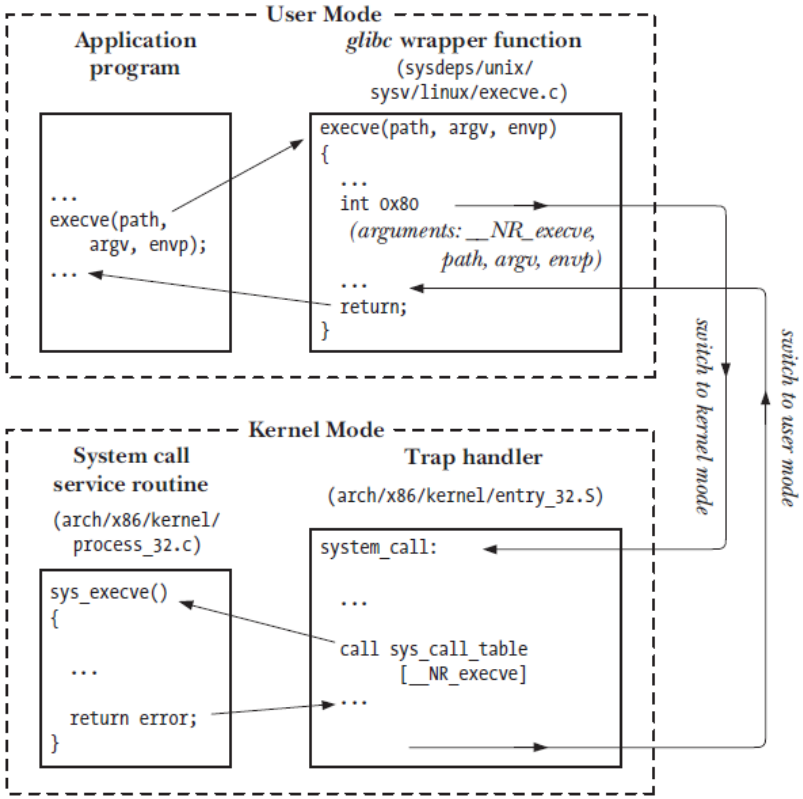


Figure 3-1: Steps in the execution of a system call

Details of system call implementation

- The program invokes a wrapper function in the C library. The reason for this is that the wrapper function sets up the system call arguments as expected by the OS. If you've ever noticed files like "libc.so", that's the C library.
- The wrapper function puts any arguments to the system call on the stack.
- All system calls enter the kernel the same way, so the kernel must be able to identify them. The wrapper function in the C library copies the system call number into the %eax register of the CPU.
- The wrapper function executes a *trap* machine instruction. This is *int 0x80*. This causes the processor to switch from user mode to kernel mode (that is, it sets CPL=0) and executes the code pointed to by location *0x80* of the system's trap vector.
- The kernel invokes its *system_call* routine to handle the trap to location *0x80*. This is where the meat of the system call logic happens: the kernel does some bookkeeping, checks the validity of the arguments, invokes the service routine, and finally the service routine returns a result status to the *system_call* routine.
    - Kernel looks up syscall number in the interrupt vector (we will show that table in next slide)
- The wrapper function checks if the service returned an error, and if so, sets a global variable named *errno* with this error value. The wrapper function then returns to the caller and provides an integer return value to indicate success or failure.

System call numbers

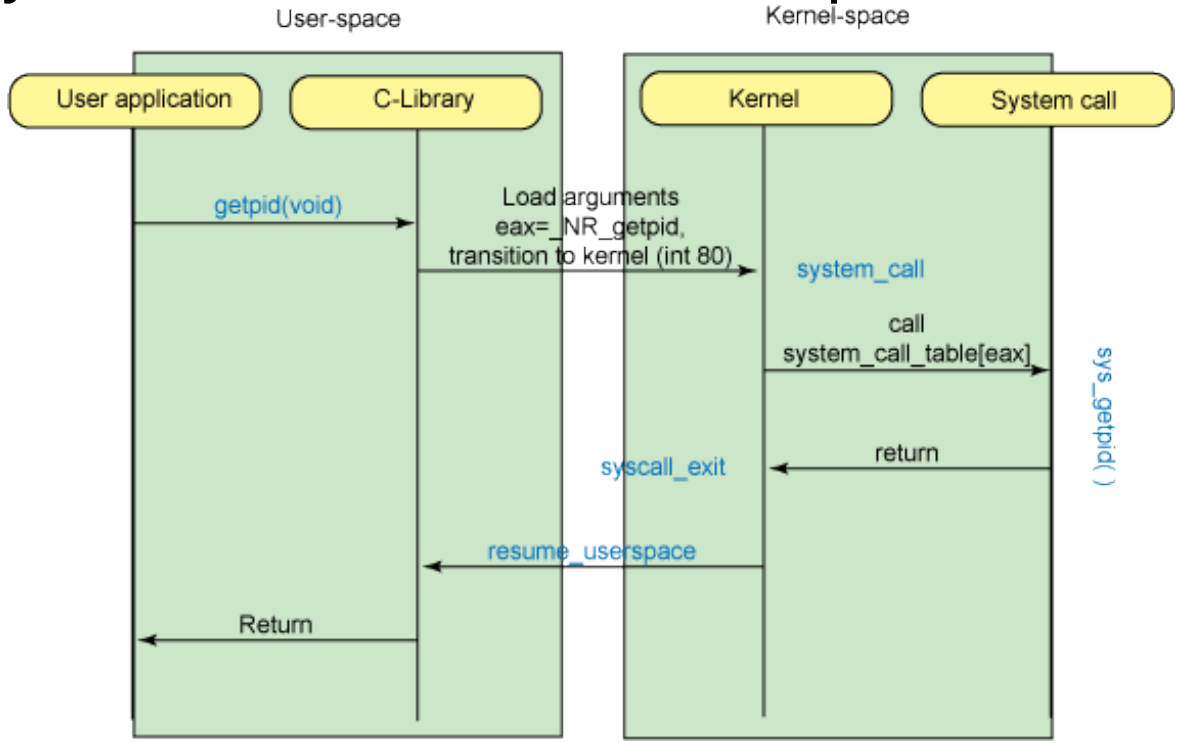
The syscall interface relies on an integer

- To associate each system call number with its corresponding service routine, the kernel uses a *system call dispatch table*, which is stored in the sys_call_table array
  - The *n-th* entry contains the service routine address of the system call having number *n*.

/ arch / x86 / entry / syscalls / syscall_32.tbl

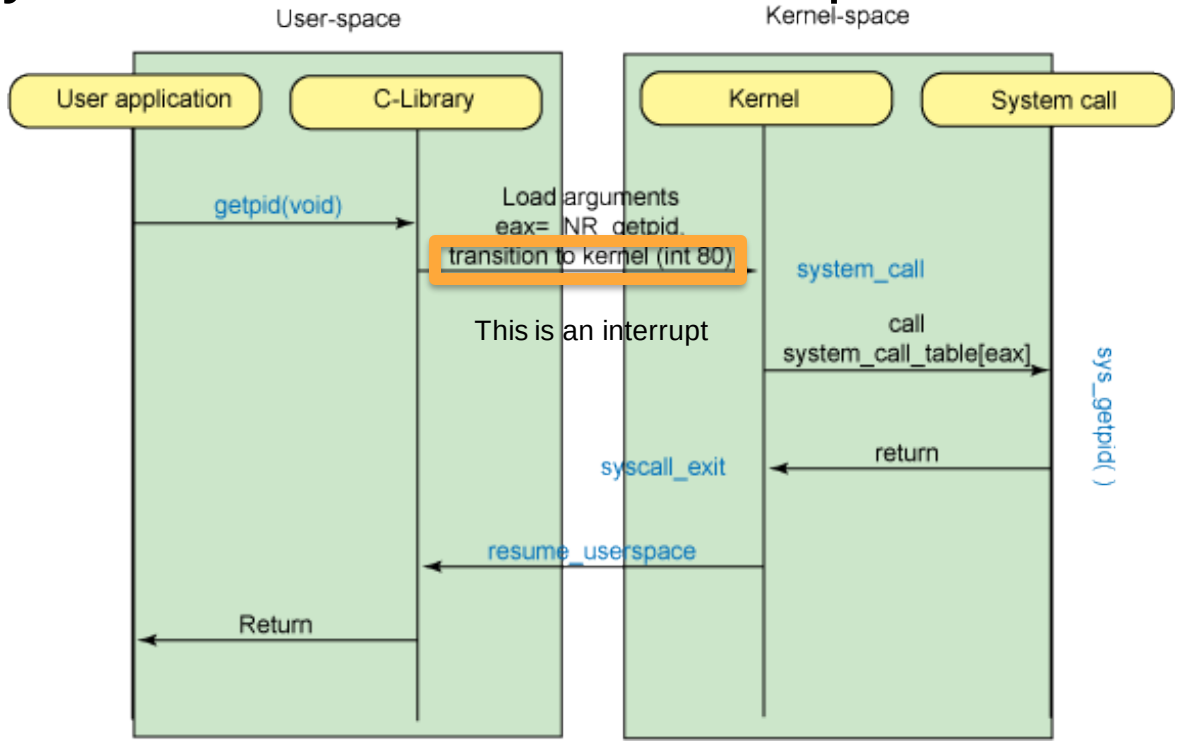
```
#
# 32-bit system call numbers and entry vectors
#
# The format is:
# <number> <abi> <name> <entry point> <compat entry point>
#
# The __ia32_sys and __ia32_compat_sys stubs are created on-the-fly for
# sys_*() system calls and compat_sys_*() compat system calls if
# IA32_EMULATION is defined, and expect struct pt_regs *regs as their only
# parameter.
#
# The abi is always "i386" for this file.
#
0       i386    restart_syscall         sys_restart_syscall             __ia32_sys_restart_syscall
1       i386    exit                    sys_exit                        __ia32_sys_exit
2       i386    fork                    sys_fork                        __ia32_sys_fork
3       i386    read                    sys_read                        __ia32_sys_read
4       i386    write                   sys_write                       __ia32_sys_write
5       i386    open                    sys_open                        __ia32_compat_sys_open
6       i386    close                   sys_close                       __ia32_sys_close
7       i386    waitpid                 sys_waitpid                     __ia32_sys_waitpid
8       i386    creat                   sys_creat                       __ia32_sys_creat
```

Another system call control flow example



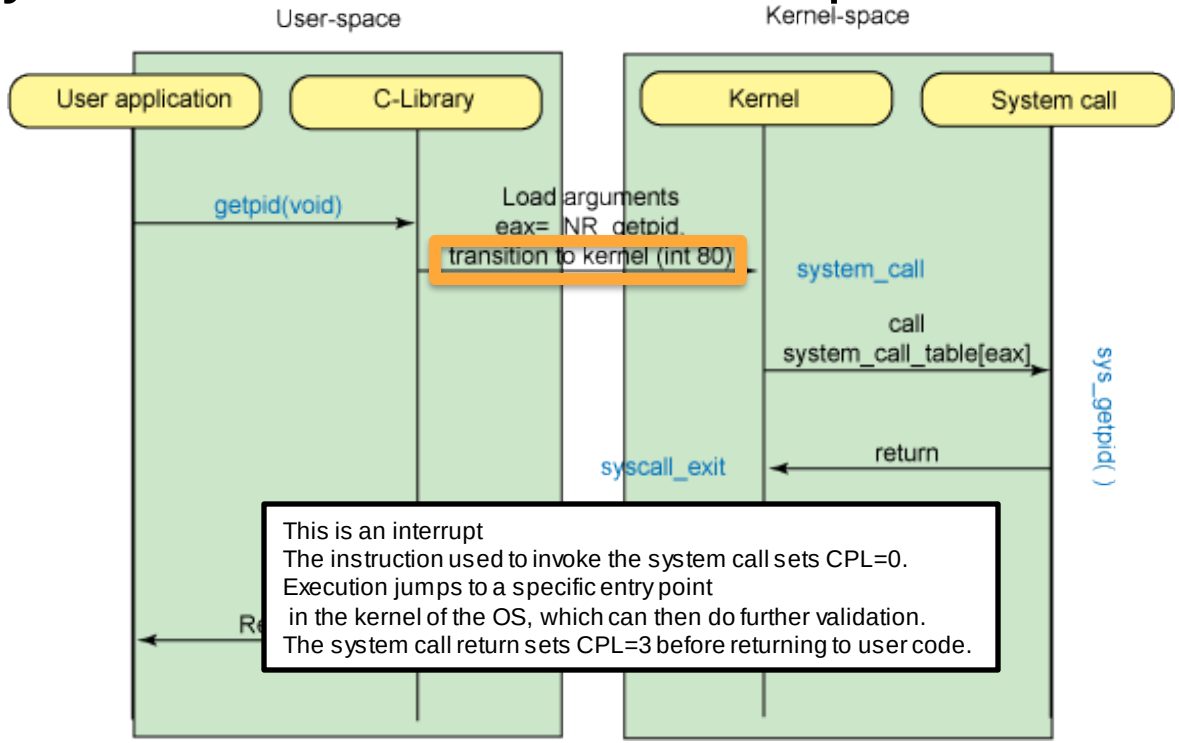
http://www.ibm.com/developerworks/library/l-system-calls/

Another system call control flow example



http://www.ibm.com/developerworks/library/l-system-calls/

Another system call control flow example



http://www.ibm.com/developerworks/library/l-system-calls/

Tracing system calls

- The *strace* command lets you see the system calls that are made by a process
- Example: type “strace ls” at a terminal to see all the system calls that the ls (which lists files in the current directory) makes. You might see calls like:
  - execve -- this loads a new program and starts running it
  - open -- open a file
  - read -- read from a file
  - close -- close a file
  - fstat -- get information about a file
- See the examples in the repository

Tracing systems calls

- strace ls

Execv call loads in a new program into a process
- We will learn more about it in the next lecture

```
vagrant@cs281spring2017devbox:~$ strace ls
execve("/bin/ls", ["ls"], [/* 58 vars */]) = 0
brk(NULL)                                 = 0x1164000
access("/etc/ld.so.nohwcap", F_OK)        = -1 ENOENT (No
such file or directory)
…..
open("/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
fstat(3, {st_mode=S_IFREG|0644, st_size=70675, ...}) = 0
mmap(NULL, 70675, PROT_READ, MAP_PRIVATE, 3, 0) =
0x7f21e9c83000
close(3)
….
exit_group(0)                             = ?
+++ exited with 0 +++                                 = 0
```

The exit_group call terminates a process