Compilation and System Calls

CS3281 / CS5281

Spring 2024

Tel (615) 343-7472 | Fax (615) 343-7440
1025 16th Avenue South Nashville, TN 37212
www.isis.vanderbilt.edu

Overview

- C Language Overview
- Compilation Basics: From Source to Executable
- Concepts and Definitions
- Introduction to System Calls

Tel (615) 343-7472 | Fax (615) 343-7440
1025 16th Avenue South Nashville, TN 37212
www.isis.vanderbilt.edu

C Language Overview

Tel (615) 343-7472 | Fax (615) 343-7440
1025 16th Avenue South Nashville, TN 37212
www.isis.vanderbilt.edu

C vs C++: A Comparison

	C	C++
Paradigm	Procedural	Object-Oriented
Keywords	32	63
Inheritance	Not Supported	Fully Supported
Overloading	Not Supported	Functional Polymorphism
Allocation	malloc/free	new/delete
Encapsulation	struct	class + struct
Access Control	None	Access Modifiers

C vs C++: Compatibility

- C++ is fully backward compatible with C
  - C++ can directly call C code
  - C code can call C++ code that has been declared with `extern "C"`
- Shared structure, grammar, and syntax
- C++ is a superset of C
  - Commonly referred to as “C with class”
- C is historically used in OS kernels and device drivers

ISIS
Tel (615) 343-7472 | Fax (615) 343-7440
1025 16th Avenue South Nashville, TN 37212
www.isis.vanderbilt.edu
VANDERBILT UNIVERSITY

C Language Overview

- `struct`
  - Groups variables into a single object
  - Stored as a contiguous block of memory
  - Identical to a C++ `class` with a `public` access modifier
  - May contain variables of any data type
  - Multiple ways of instantiating objects
  - **Must** include the “struct” keyword when declaring variables of corresponding type

```
struct example {
    int var1, var2;
    char var3;
};

struct example ex1 = {
    1, 2, ‘c’ };

struct example ex2 = {
    .var1 = 1,
    .var2 = 2,
    .var3 = ‘c’
};
```

Tel (615) 343-7472 | Fax (615) 343-7440
1025 16th Avenue South Nashville, TN 37212
www.isis.vanderbilt.edu

C Language Overview

- Pointer
  - Stores the **memory address** of a variable
  - Also known as a **reference** to a variable
  - Declare pointer using “*” symbol
  - Obtain variable address using “&”
  - Dereference pointer using “->” or “*”
  - Ex:
    - `ex2->var1 == 1;`
    - What does ex2->var2 contain?

```
struct test {
    int var1, *var2;
    char var3;
};

struct test ex1 = {
    1, 2, ‘c’ };

struct test *ex2 = &ex1;
```

C Language Overview

- Function Pointer
  - Exact same as a variable pointer
  - Contains the address of a function
  - Can be stored in a variable and passed to other functions
  - See example on right for syntax
- **All pointers** are *typically* stored in a memory block of the same size as the underlying hardware architecture
  - E.g., A 64-bit OS uses 64-bit pointers

```
int func(int a, char b)
{
   return a + b;
};

// Function pointer decl
int (*fp)(int, char);

// Function assignment
fp = &func;
```

Tel (615) 343-7472 | Fax (615) 343-7440
1025 16th Avenue South Nashville, TN 37212
www.isis.vanderbilt.edu

C Language Overview

- Header Files
  - Have the extension ".h"
  - Used to *declare* (not *define*) types, variables, and definitions
- Compilation Unit
  - Originates from a **single** source file with the extension ".c"
  - Includes contents of **all** "#include"ed header files
  - All Compilation Units linked together during final stage of compilation
- `extern` keyword
  - Declares variables that are **defined** in a different Compilation Unit

Compilation: From Source Code to Runnable Program

From Source to Running Program in C

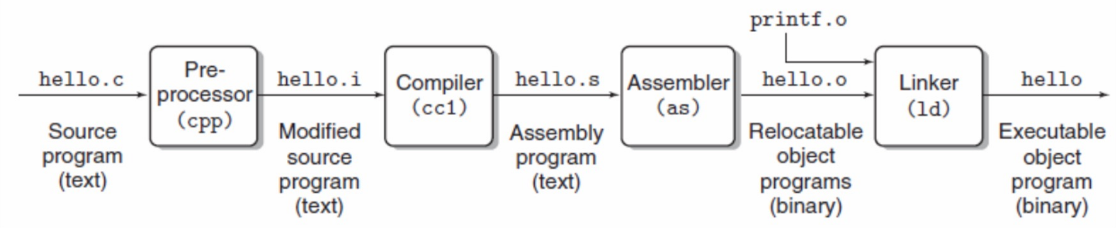
- We begin with a source file saved as text (e.g., prog.c)
  - This is a sequence of bits organized into 8-bit chunks called bytes
    - Each byte represents a character
  - ASCII/UTF-8: Unique byte-sized integer value for each character
    - Source file stored as sequence of bytes
    - Files consisting exclusively of ASCII or UTF-8 characters are text files; binary files otherwise
  - All files are just bits
    - Distinguishing difference is the context in which we view them

Tel (615) 343-7472 | Fax (615) 343-7440
1025 16th Avenue South Nashville, TN 37212
www.isis.vanderbilt.edu

VANDERBILT UNIVERSITY

From Source to Running Program

- Before it can run, source code must be translated into a sequence of machine-language instructions
  - Packaged into a form called an “executable object program” and stored as a binary file on disk
- Translation is done by a **compiler**
- Individual steps of compilation can be seen 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 a binary object file
  - gcc   hello.o // produces a linked executable



Tel (615) 343-7472 | Fax (615) 343-7440
1025 16th Avenue South Nashville, TN 37212
www.isis.vanderbilt.edu

From Source to Running Program

- 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

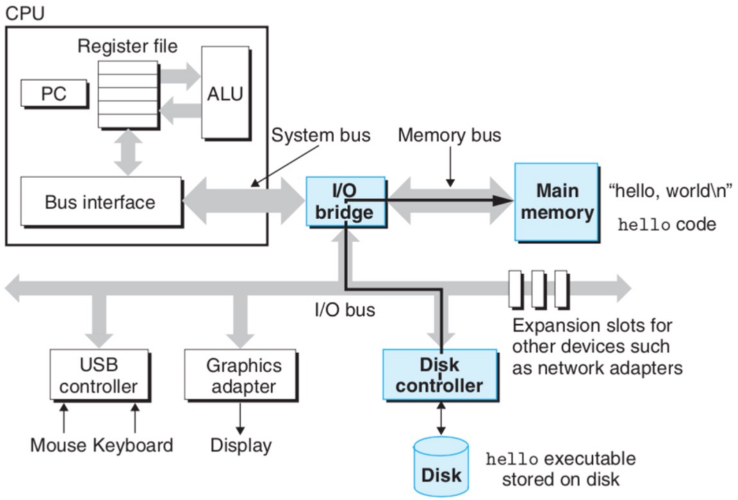
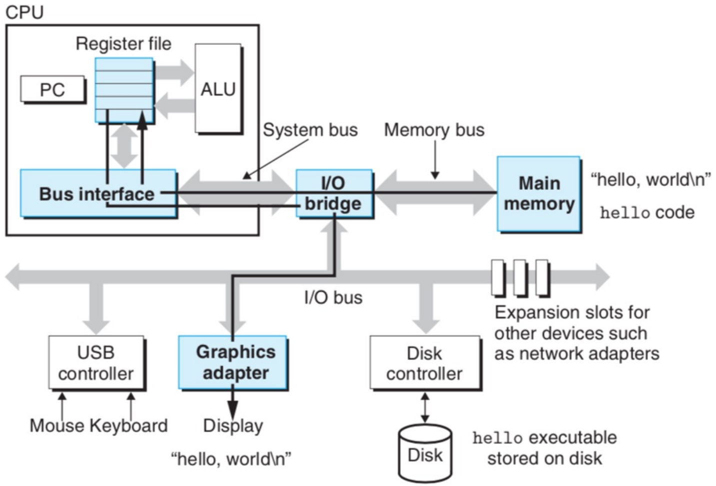


Figure 1.6 **Loading the executable from disk into main memory.**

From Source to Running Program

- 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.



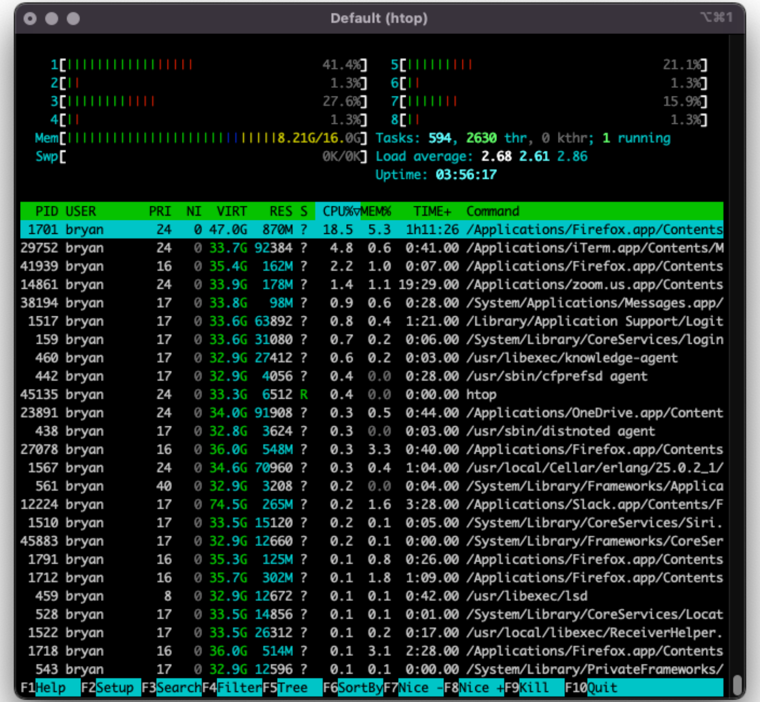
What is an Operating System?

Operating System: Definition

- 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*
- Key point: the operating system is itself a program!
  - But unlike other programs, it should have full access to <u>all</u> 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 it 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



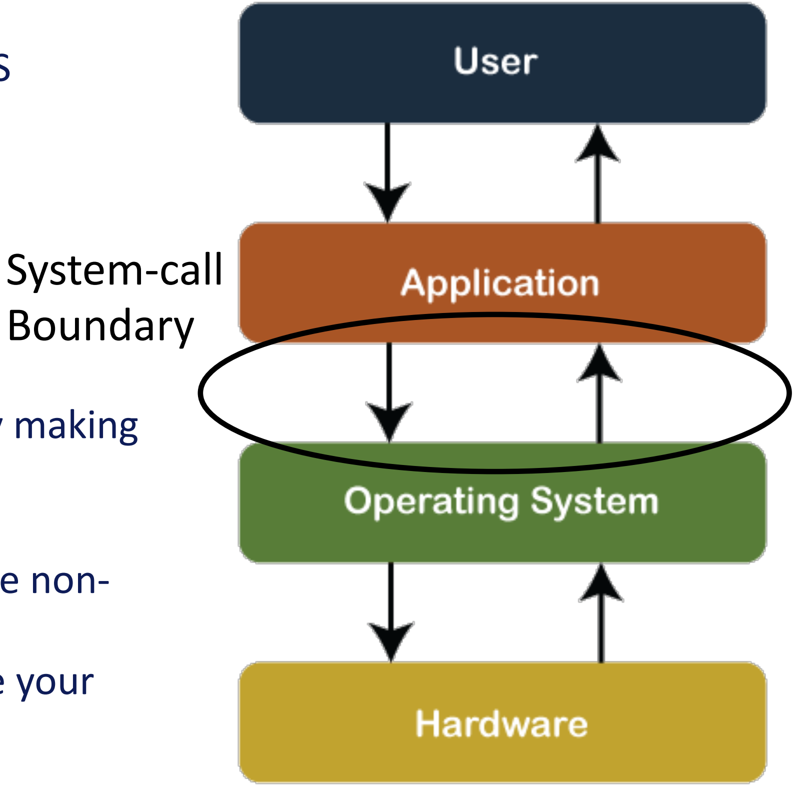
“ps” “top” or “htop” show all active processes

Tel (615) 343-7472 | Fax (615) 343-7440
1025 16th Avenue South Nashville, TN 37212
www.isis.vanderbilt.edu

VANDERBILT UNIVERSITY

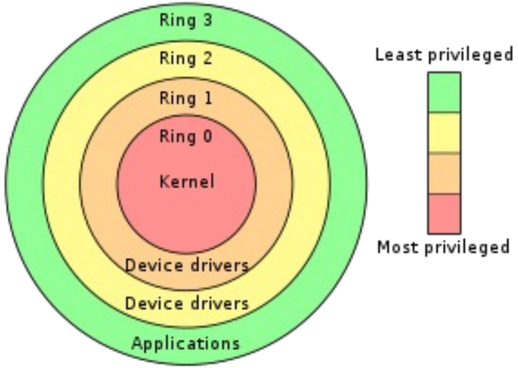
System Calls

- 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
  - Request memory
  - Read/write from/to a file
  - Send data over the network
- A process requests theses 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 <u>not</u> be allowed full access to all hardware <u>nor</u> 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 x86 it's called the CPL (current privilege level)
    - CPL = 0 (0b00) means kernel mode: privileged
    - CPL = 3 (0b11) means user mode: no privilege
  - The CPL enforces isolation in several ways:
    - Guards access to the privilege register
    - Checks every memory read/write
    - Checks every I/O port access
    - Controls register access (like the EFLAGS register)
- In summary:
  - The privilege level determines whether instructions can access privileged hardware
  - Only the kernel should be allowed to operate in a privileged mode



Invoking a System Call

- How and when should the privilege level change?
- When a system call is invoked:
  - Instruction used to invoke the system call sets CPL=0
  - Execution jumps to a specific entry point in the kernel, which can then do further validation
  - The system call return sets CPL=3 before returning to user code

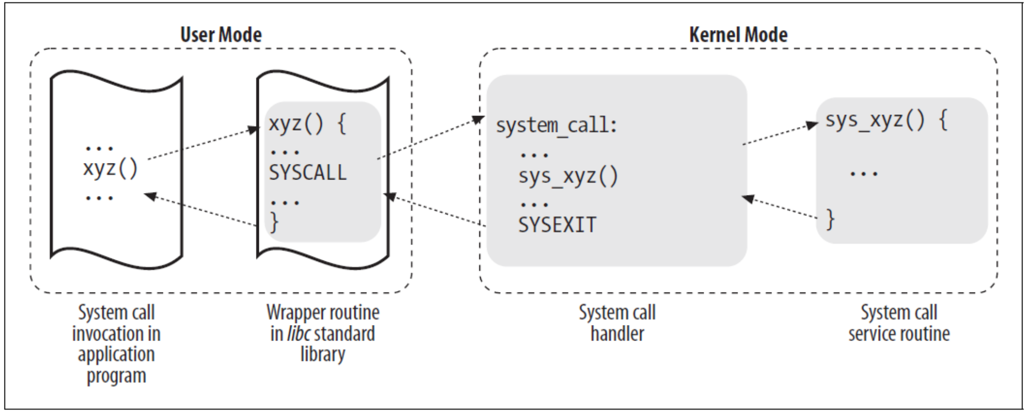


Figure 10-1. Invoking a system call

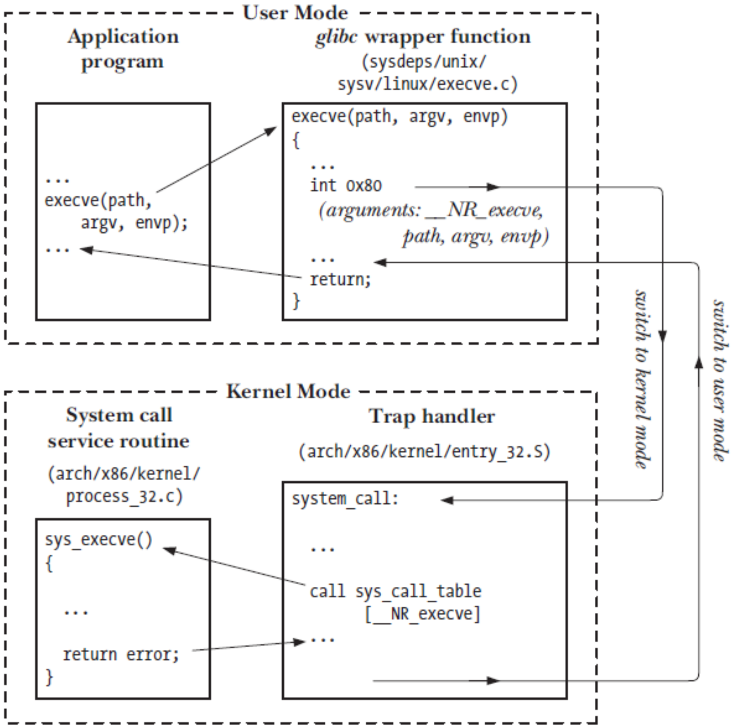


Figure 3-1: Steps in the execution of a system call

Tel (615) 343-7472 | Fax (615) 343-7440
1025 16th Avenue South Nashville, TN 37212
www.isis.vanderbilt.edu
VANDERBILT UNIVERSITY

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
- 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. In x86, 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
- 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 returns to the caller and provides an integer return value to indicate success or failure

System Call Numbers

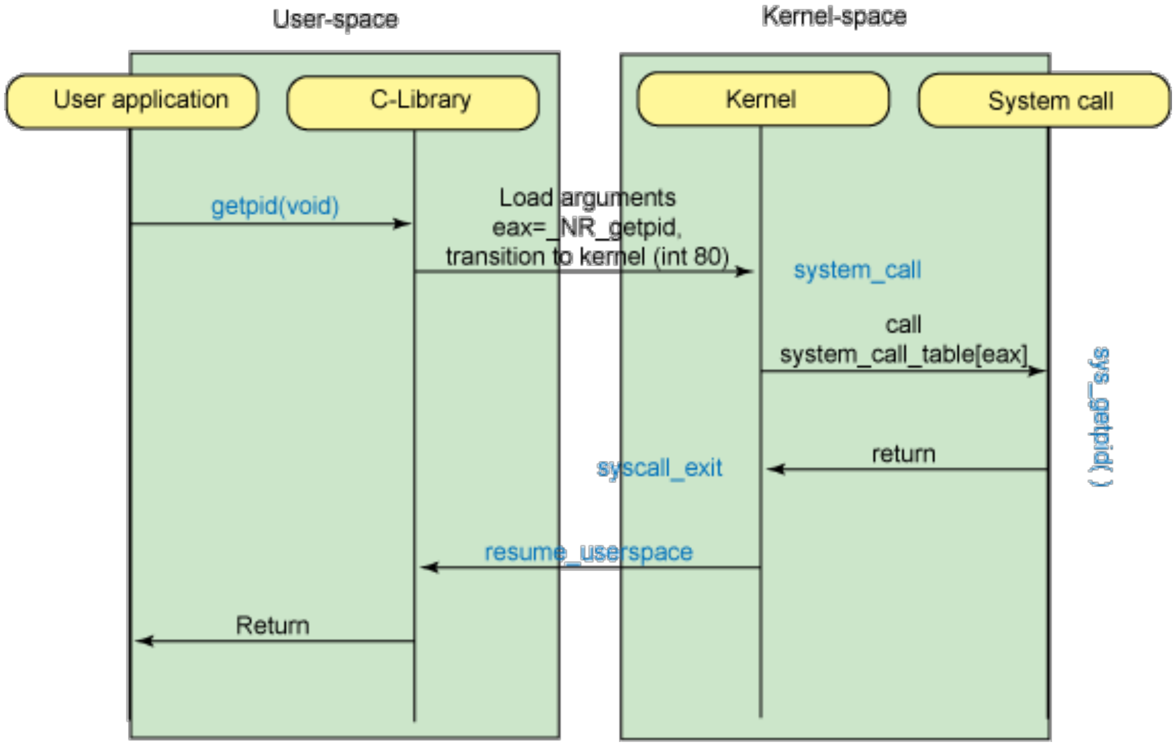
The [syscall interface](#)relies on an integer

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

```c
// An array mapping syscall numbers from syscall.h
// to the function that handles the system call.
static uint64 (*syscalls[])(void) = {
[SYS_fork]    sys_fork,
[SYS_exit]    sys_exit,
[SYS_wait]    sys_wait,
[SYS_pipe]    sys_pipe,
[SYS_read]    sys_read,
[SYS_kill]    sys_kill,
[SYS_exec]    sys_exec,
[SYS_fstat]   sys_fstat,
[SYS_chdir]   sys_chdir,
[SYS_dup]     sys_dup,
[SYS_getpid]  sys_getpid,
[SYS_sbrk]    sys_sbrk,
[SYS_sleep]   sys_sleep,
[SYS_uptime]  sys_uptime,
[SYS_open]    sys_open,
[SYS_write]   sys_write,
[SYS_mknod]   sys_mknod,
[SYS_unlink]  sys_unlink,
[SYS_link]    sys_link,
[SYS_mkdir]   sys_mkdir,
[SYS_close]   sys_close,
};
```

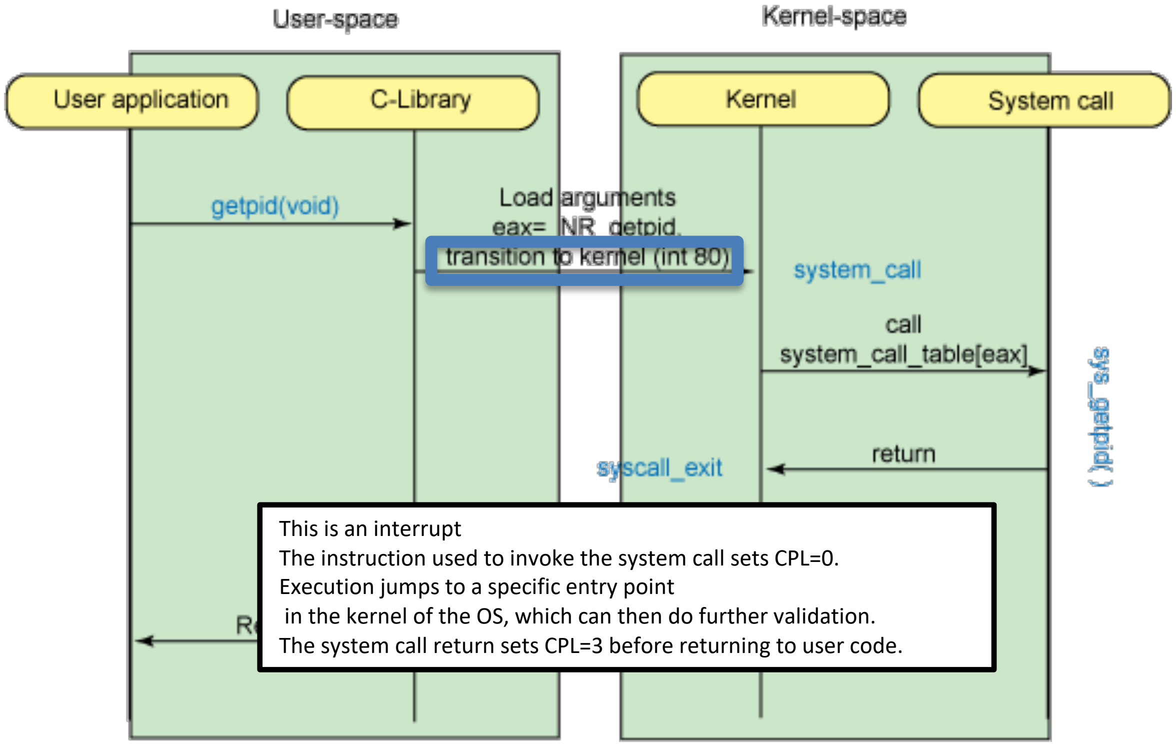
xv6 syscall table

Another system-call control flow example



Tel (615) 343-7472 | Fax (615) 343-7440
1025 16th Avenue South Nashville, TN 37212
www.isis.vanderbilt.edu

Another system call control flow example



This is an interrupt
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.

Tel (615) 343-7472 | Fax (615) 343-7440
1025 16th Avenue South Nashville, TN 37212
www.isis.vanderbilt.edu

VANDERBILT UNIVERSITY

Let’s see this in xv6!

https://github.com/mit-pdos/xv6-riscv/blob/riscv/kernel/trap.c#L67