



# Compilation and System Calls

CS3281 / CS5281

Fall 2025



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



# Overview

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





# C Language Overview



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



# C vs C++: A Comparison

	C	C++
<b>Paradigm</b>	Procedural	Object-Oriented
<b>Keywords</b>	32	63
<b>Inheritance</b>	Not Supported	Supported
<b>Polymorphism</b>	Not Supported	Functions, Templates
<b>Allocation</b>	malloc, free	new, delete
<b>Encapsulation</b>	struct	class, struct
<b>Access Control</b>	None	Access Modifiers
<b>Compatibility</b>		Superset of C
<b>Applications</b>	OS, device drivers	Gaming, Web



# 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



# 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 e;  
e.var1 = 1;  
e.var2 = 2;  
e.var3 = 'c';  
  
struct example ex1 = {  
    1, 2, 'c' };  
  
struct example ex2 = {  
    .var2 = 2,  
    .var1 = 1,  
    .var3 = 'c'  
};
```

# 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;
```



# 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



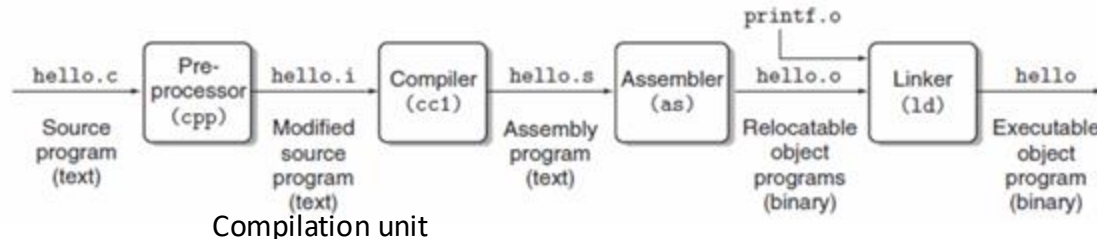
# From Source to Running Program in C

- 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



# 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 -o hello` // produces a linked executable



# 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.
  - Once the code and data are in memory, the OS switches to the “hello” process and begins executing its instructions

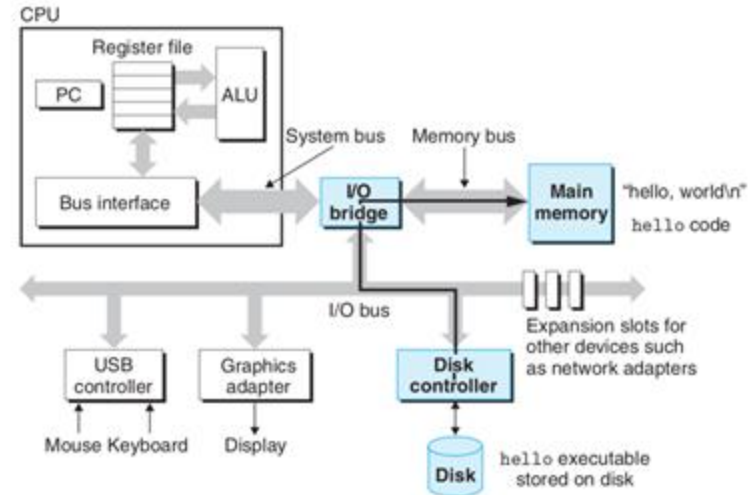
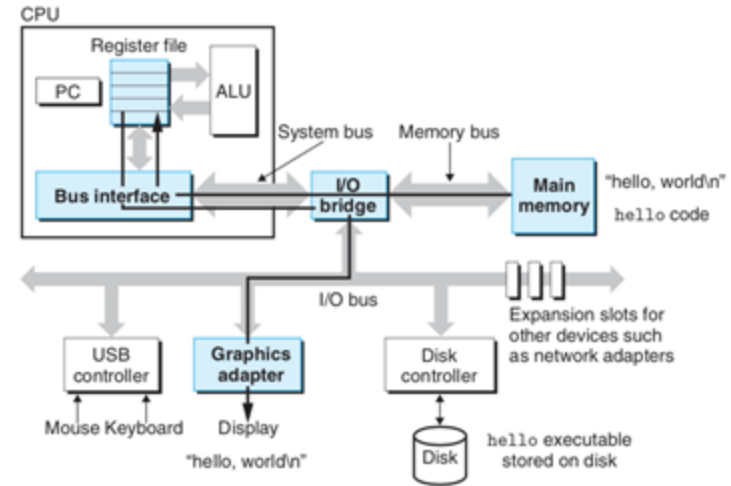


Figure 1.6 Loading the executable from disk into main memory.

# 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 all resources! How can we accomplish this?





# Processes

- A process is a running instance of a program
- It's the main unit of **abstraction** provided by the OS
- OS makes your machine easy to use even though many processes run in the machine
- OS does it through **virtualization**
  - Each process thinks it has the whole CPU
  - Each process thinks it has all memory to itself
- Virtualization provides (memory) isolation:
  - Prevents a process X from corrupting or spying on a process Y
  - Prevents a process from corrupting the operating system itself

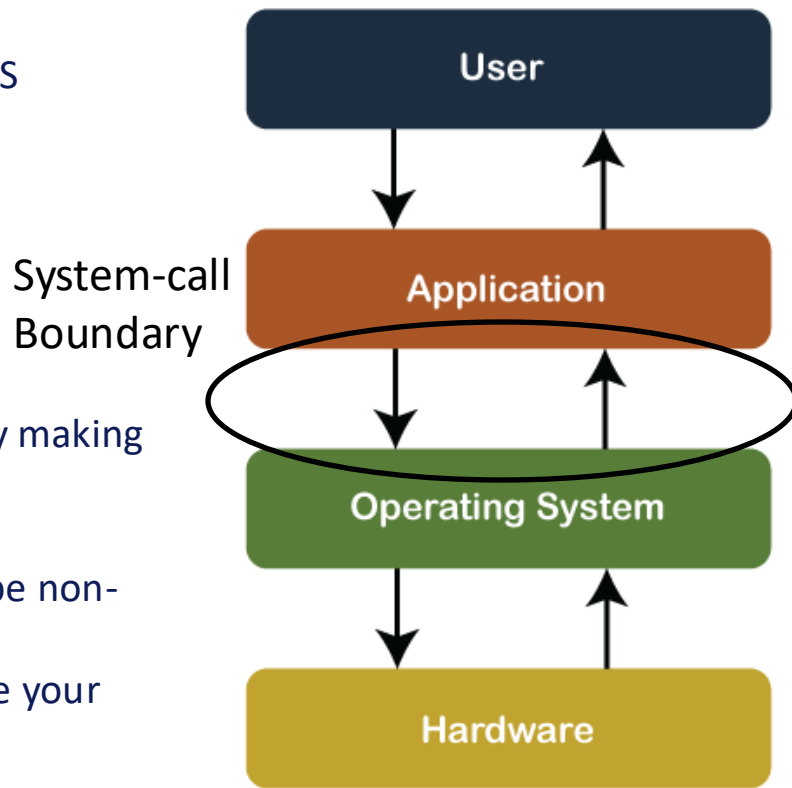
```
1[|||||] 41.4% 5[|||||] 21.1%
2[||] 1.3% 6[||] 1.3%
3[|||||] 27.6% 7[|||||] 15.9%
4[||] 1.3% 8[||] 1.3%
Mem[|||||] 18.21G/16.0G Tasks: 594, 2630 thr, 0 kthr; 1 running
Swp[|||||] 0K/0K Load average: 2.68 2.61 2.86
Uptime: 03:56:17

  PID USER      PRI  NI  VIRT   RES   S  CPU% MEM%   TIME+  Command
1701 bryan      24    0  47.0G  870M ?   18.5  5.3  1h11:26 /Applications/Firefox.app/Contents
29752 bryan     24    0  33.7G  92384 ?   4.8  0.6  0:41.00 /Applications/iTerm.app/Contents/M
41939 bryan     16    0  35.4G  162M ?   2.2  1.0  0:07.00 /Applications/Firefox.app/Contents
14861 bryan     24    0  33.9G  178M ?   1.4  1.1  19:29.00 /Applications/zoom.us.app/Contents
38194 bryan     17    0  33.8G   98M ?   0.9  0.6  0:28.00 /System/Applications/Messages.app/
1517 bryan     17    0  33.6G  63892 ?   0.8  0.4  1:21.00 /Library/Application Support/Login
159 bryan      17    0  33.6G  31080 ?   0.7  0.2  0:06.00 /System/Library/CoreServices/Login
460 bryan      17    0  32.9G  27412 ?   0.6  0.2  0:03.00 /usr/libexec/knowledge-agent
442 bryan      17    0  32.9G  4056 ?   0.4  0.0  0:28.00 /usr/sbin/cfprefsd agent
45135 bryan     24    0  33.3G  6512 R   0.4  0.0  0:00.00 http
23891 bryan     24    0  34.0G  91908 ?   0.3  0.5  0:44.00 /Applications/OneDrive.app/Content
438 bryan      17    0  32.8G  3624 ?   0.3  0.0  0:03.00 /usr/sbin/distnoted agent
27078 bryan     16    0  36.0G  548M ?   0.3  3.3  0:40.00 /Applications/Firefox.app/Contents
1567 bryan     24    0  34.6G  70960 ?   0.3  0.4  1:04.00 /usr/local/Cellar/erlang/25.0.2_1/
561 bryan      40    0  32.9G  3208 ?   0.2  0.0  0:04.00 /System/Library/Frameworks/Applica
12224 bryan     17    0  74.5G  265M ?   0.2  1.6  3:28.00 /Applications/Slack.app/Contents/F
1510 bryan     17    0  33.5G  15120 ?   0.2  0.1  0:05.00 /System/Library/CoreServices/Siri.
45883 bryan     17    0  32.9G  12660 ?   0.2  0.1  0:00.00 /System/Library/Frameworks/CoreSer
1791 bryan     16    0  35.3G  125M ?   0.1  0.8  0:26.00 /Applications/Firefox.app/Contents
1712 bryan     16    0  35.7G  302M ?   0.1  1.8  1:09.00 /Applications/Firefox.app/Contents
459 bryan       8    0  32.9G  12672 ?   0.1  0.1  0:42.00 /usr/libexec/lsd
528 bryan      17    0  33.5G  14856 ?   0.1  0.1  0:01.00 /System/Library/CoreServices/Locat
1522 bryan     17    0  33.5G  26312 ?   0.1  0.2  0:17.00 /usr/local/libexec/ReceiverHelper.
1718 bryan     16    0  36.0G  514M ?   0.1  3.1  2:28.00 /Applications/Firefox.app/Contents
543 bryan      17    0  32.9G  12596 ?   0.1  0.1  0:00.00 /System/Library/PrivateFrameworks/
FileMngt F2Setup F3Search F4Filter F5Tree F6SortBy F7Nice F8Nice F9Kill F10Quit
```

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

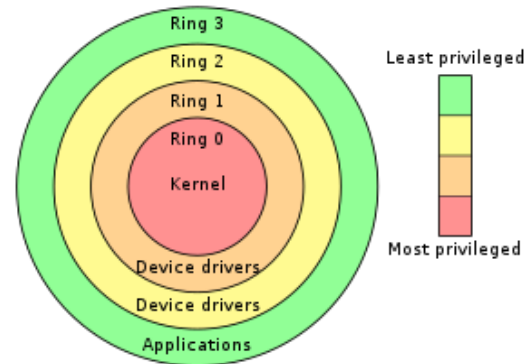
# System Calls

- 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 (running program)
  - Request memory
  - Read/write from/to a file
  - Send data over the network
- A process requests these services 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 (status 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
- 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:
  - A special instruction (supervisor 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

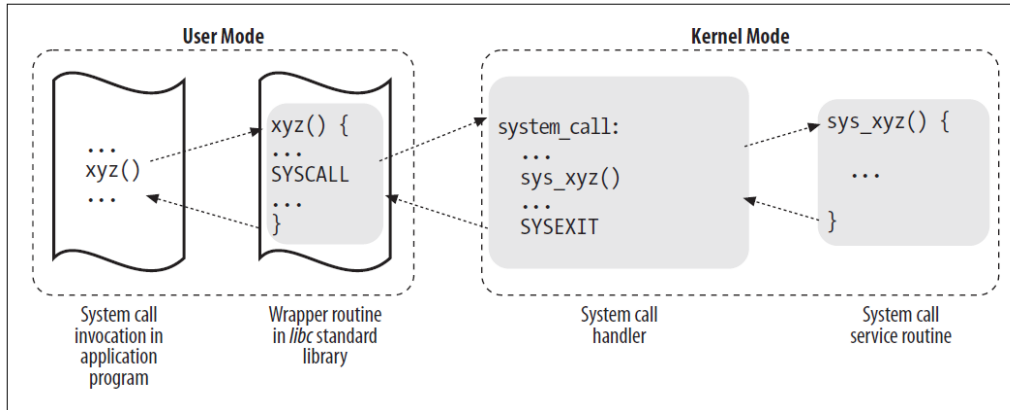


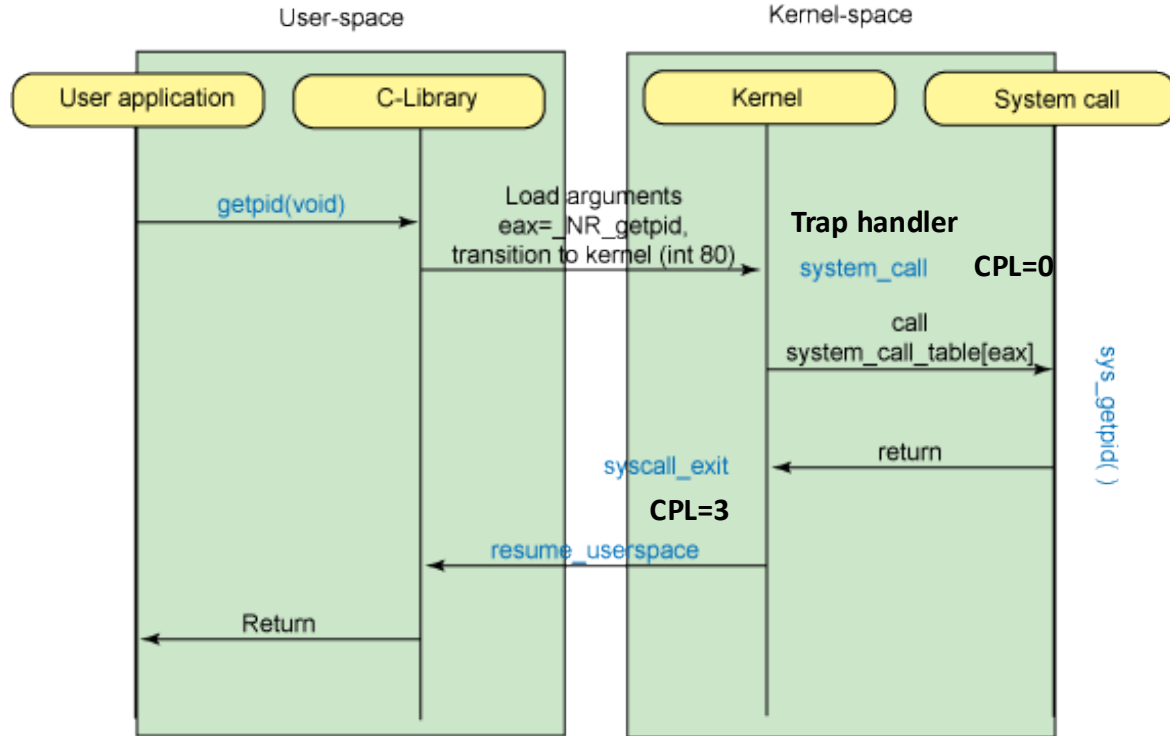
Figure 10-1. Invoking a system call

# Details of System-Call Implementation

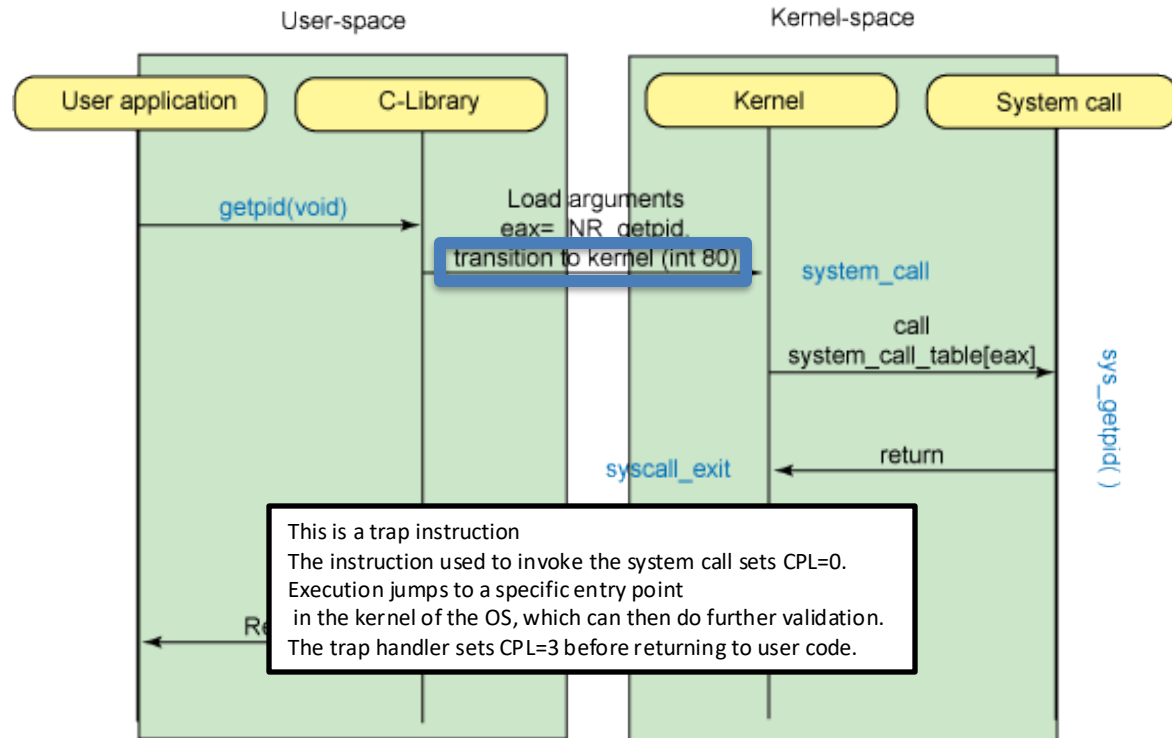
- The program invokes a wrapper function in the C library. The reason for this is that the wrapper function sets up the system-call arguments as expected by the OS
- The wrapper function puts any arguments to the system call on the stack
- All system calls enter the kernel the same way, so the kernel 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 (branch) vector. The executed code is a function that handles traps
- The kernel invokes its *system\_call* routine (trap handler) 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



# Another system-call control flow example



# Another system call control flow example



# 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





# 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

