

# CSE 333

## Lecture 7 - system calls, intro to file I/O

**Hal Perkins**

Department of Computer Science & Engineering

University of Washington



# Administrivia

Welcome to new TAs: Yibo Cao and David Porter

Upcoming lectures and sections: I/O and system calls

Essential material for next part of the project

Also interesting by itself

New exercise due Wednesday morning

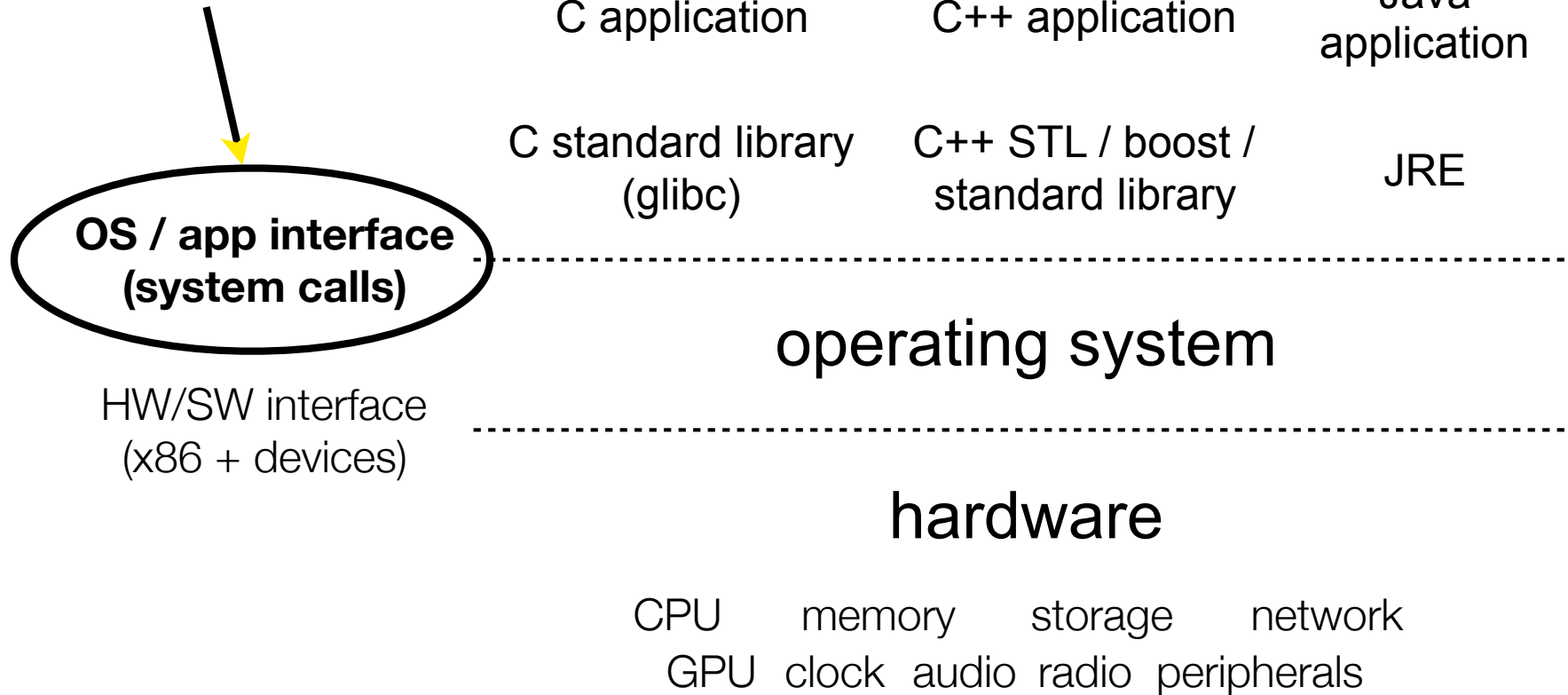
Next sections: POSIX I/O and reading directories

Yet another exercise out after that, due Monday morning

No exercise due Friday because hw1 due Thursday night

# Remember this picture?

## brief diversion



# What's an OS?

Software that:

- directly interacts with the hardware

  - OS is trusted to do so; user-level programs are not

  - OS must be ported to new HW; user-level programs are portable

- manages (allocates, schedules, protects) hardware resources

  - decides which programs can access which files, memory locations, pixels on the screen, etc., and when

- abstracts away messy hardware devices

  - provides high-level, convenient, portable abstractions

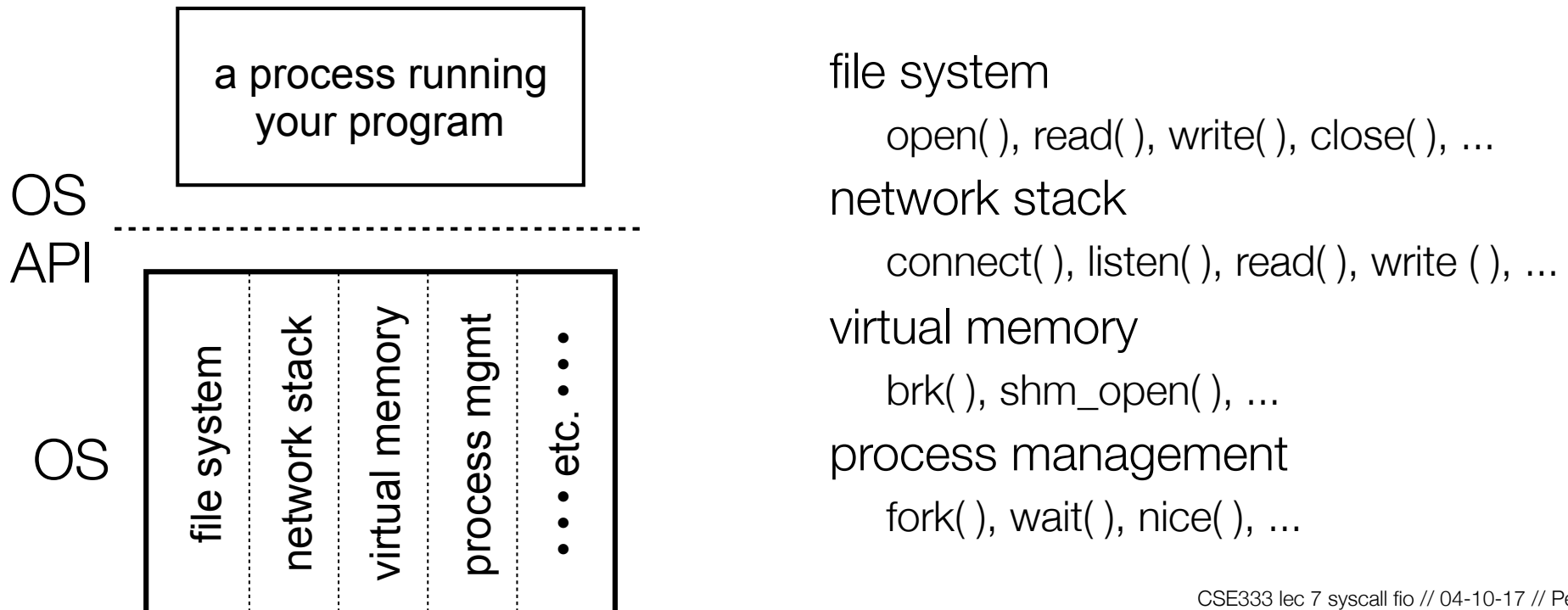
    - e.g., files vs. disk blocks

# OS as an abstraction provider

The OS is the “layer below”

a module that your program can call (with system calls)

provides a powerful API (the OS API - POSIX, Windows, ...)

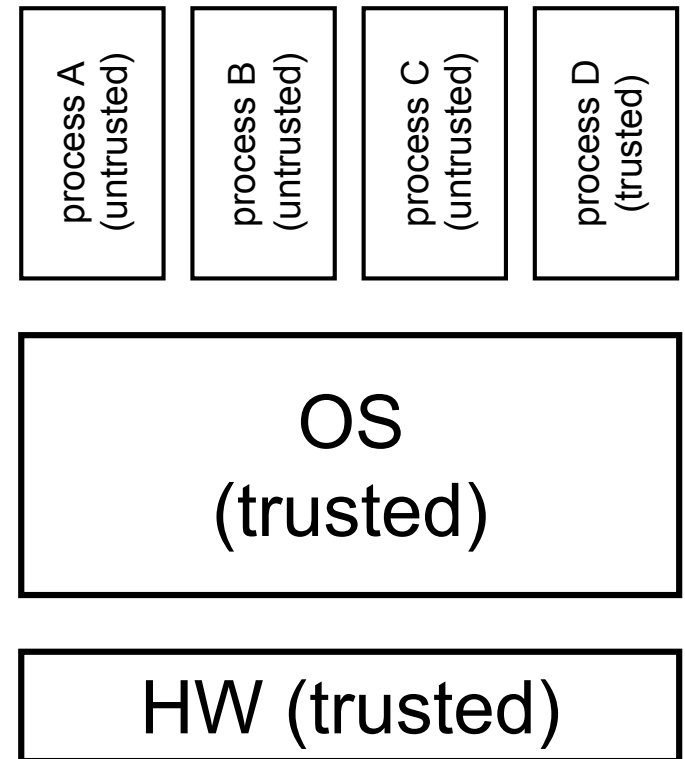


# OS as a protection system

OS isolates processes from each other  
but permits controlled sharing between them  
through shared name spaces (e.g., FS names)

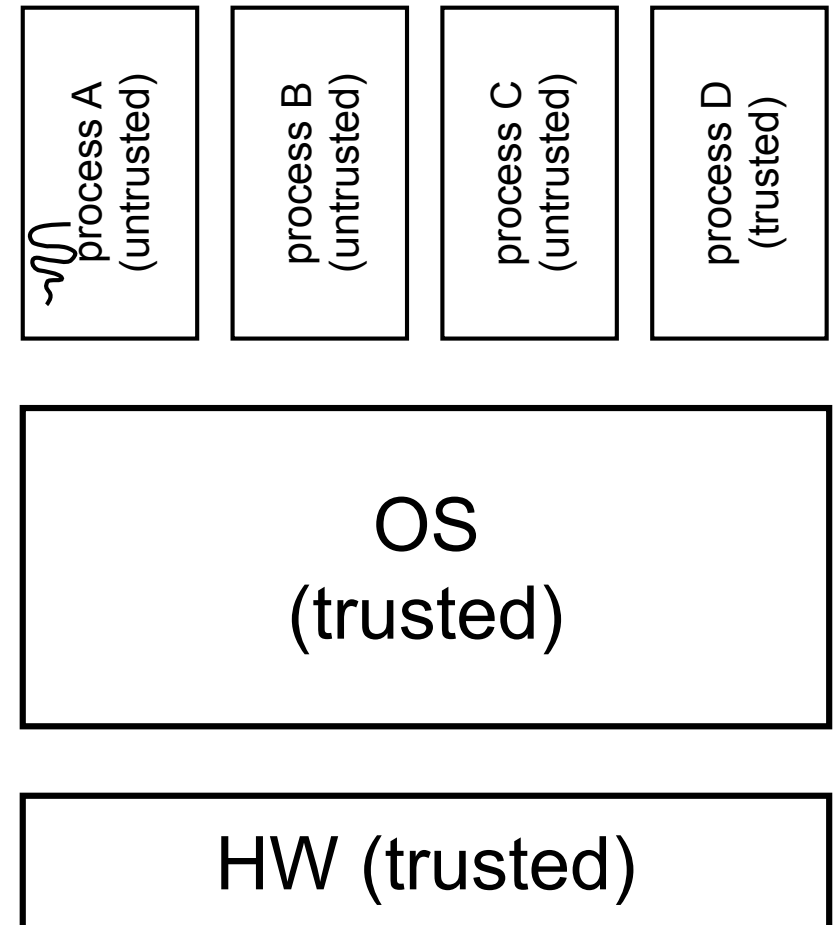
OS isolates itself from processes  
and therefore, must prevent processes from  
accessing the hardware directly

OS is allowed to access the hardware  
user-level processes run with the CPU in  
unprivileged mode  
when the OS is running, the CPU is set to  
privileged mode  
user-level processes invoke a system call to  
safely enter the OS



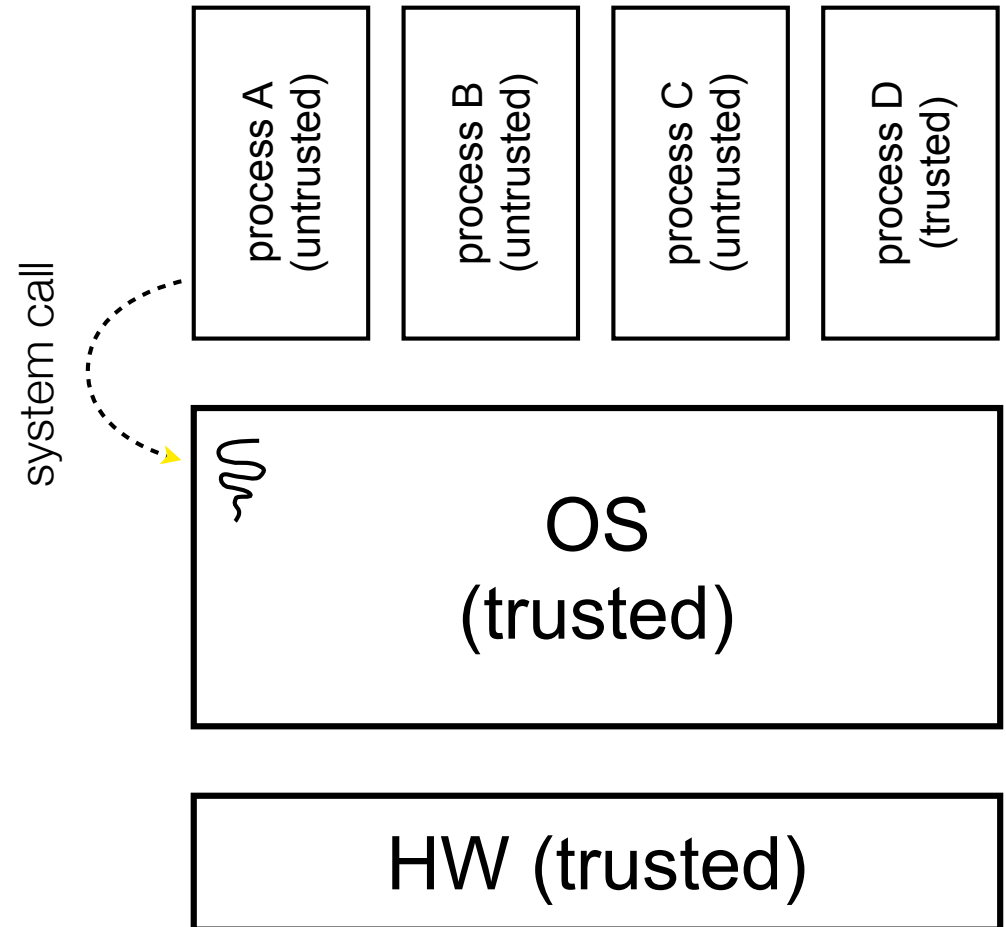
# OS as a protection system

a CPU (thread of execution) is running user-level code in process A; that CPU is set to unprivileged mode



# OS as a protection system

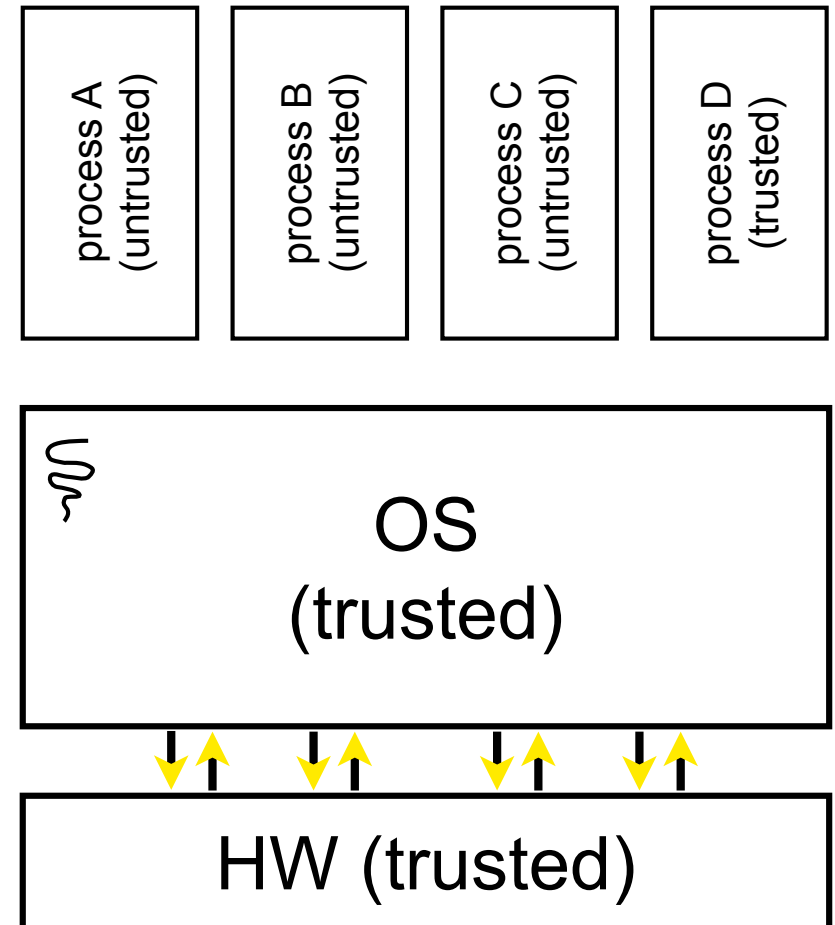
code in process A invokes a system call; the hardware then sets the CPU to privileged mode and traps into the OS, which invokes the appropriate system call handler





# OS as a protection system

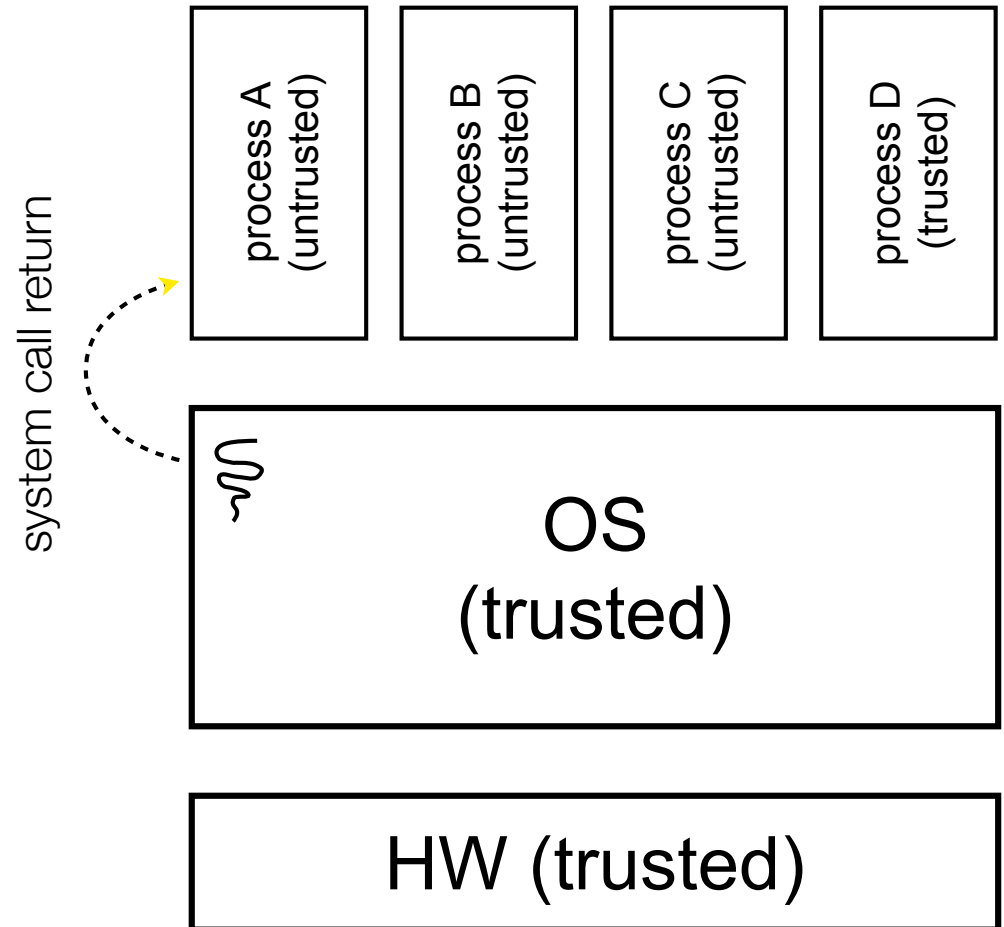
because the CPU executing the thread that's in the OS is in privileged mode, it is able to use privileged instructions that interact directly with hardware devices like disks



# OS as a protection system

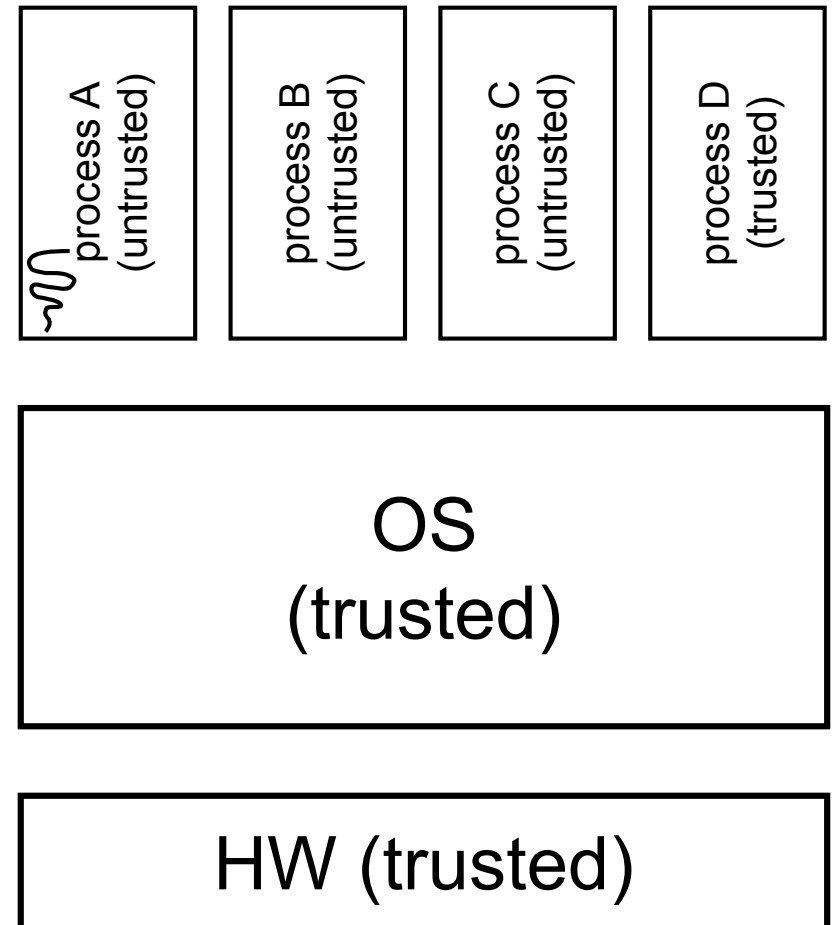
once the OS has finished servicing the system call (which might involve long waits as it interacts with HW) it:

- (a) sets the CPU back to unprivileged mode, and
- (b) returns out of the system call back to the user-level code in process A



# OS as a protection system

the process continues  
executing whatever code  
that is next after the  
system call invocation



Useful reference: *Computer Systems: A Programmer's Perspective* (CSE351 book) secs. 8.1-8.3

# Details on x86 / Linux

A more accurate picture:

consider a typical Linux process

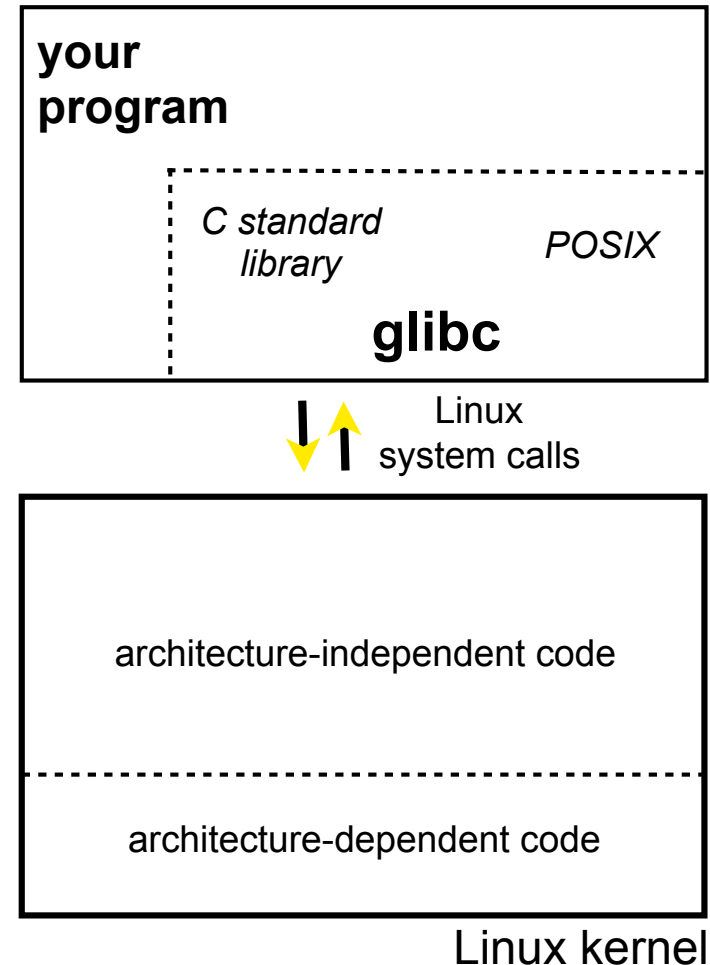
its thread of execution can be  
several places

in your program's code

in **glibc**, a shared library  
containing the C standard library,  
POSIX support, and more

in the Linux architecture-  
independent code

in Linux x86-32/x86-64 code



# Details on x86 / Linux

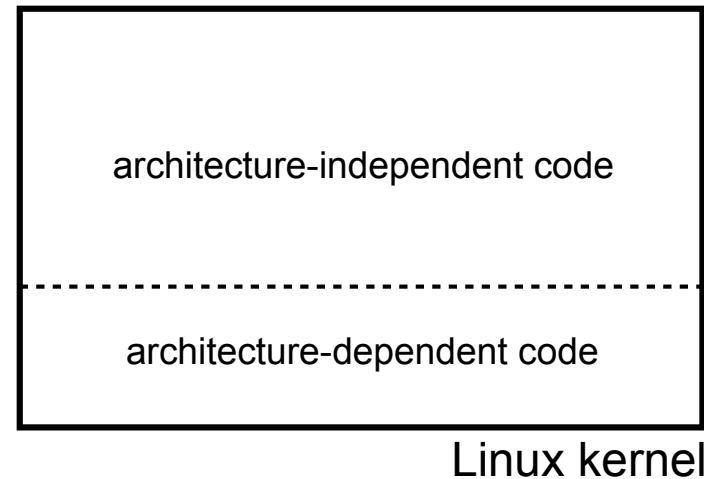
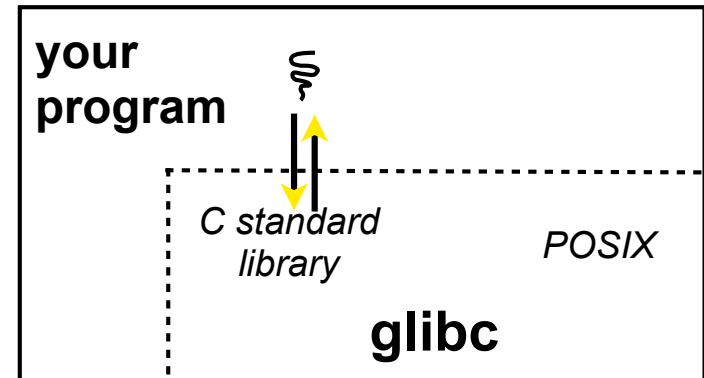
Some routines your program invokes may be entirely handled by glibc

without involving the kernel

e.g., **strcmp()** from `stdio.h`

∃ some initial overhead when invoking functions in dynamically linked libraries

but, after symbols are resolved, invoking glibc routines is nearly as fast as a function call within your program itself



# Details on x86 / Linux

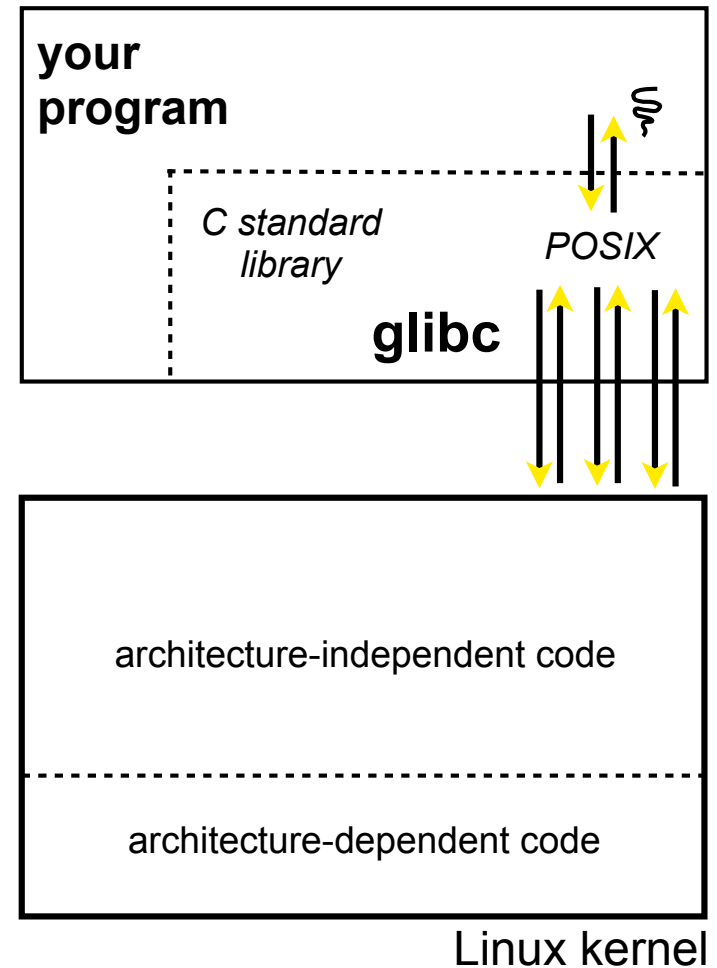
Some routines may be handled by glibc, but they in turn invoke Linux system calls

e.g., POSIX wrappers around Linux syscalls

POSIX `readdir( )` invokes the underlying Linux `readdir( )`

e.g., C stdio functions that read and write from files

`fopen( )`, `fclose( )`, `fprintf( )` invoke underlying Linux `open( )`, `read( )`, `write( )`, `close( )`, etc.

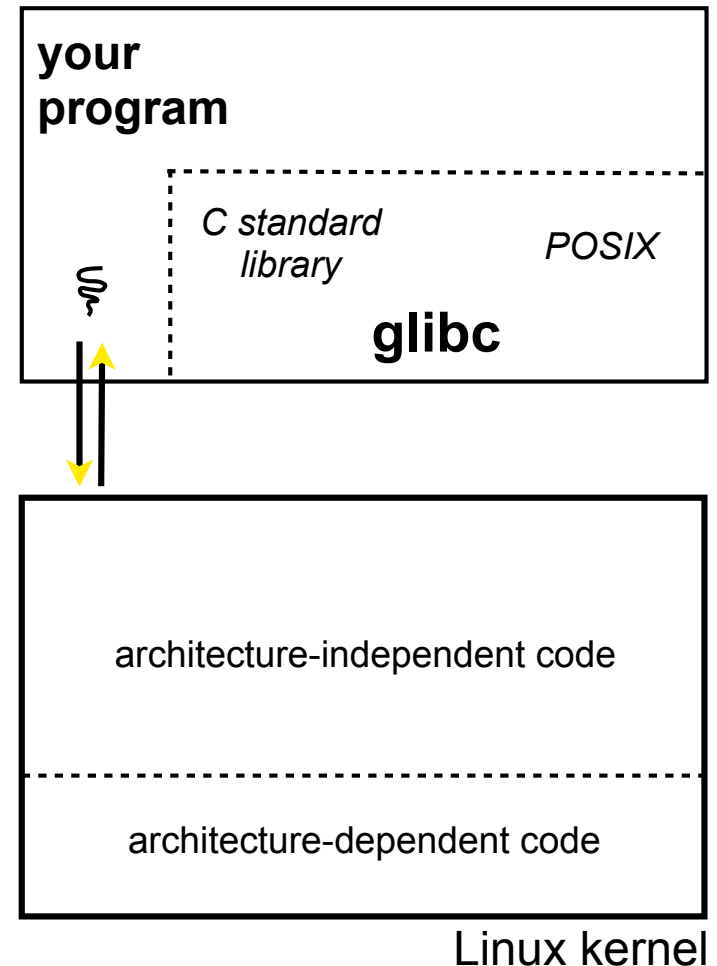


# Details on x86 / Linux

Your program can choose to directly invoke Linux system calls as well

nothing forces you to link with glibc and use it

but, relying on directly invoked Linux system calls may make your program less portable across UNIX varieties



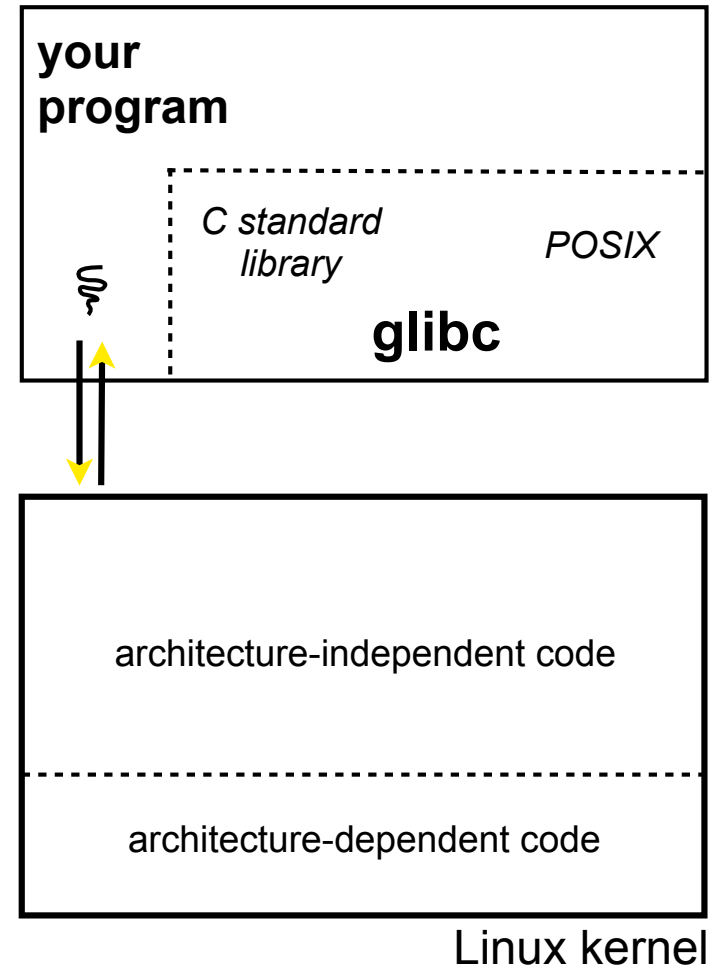
# Details on x86 / Linux

Let's walk through how a Linux system call actually works

we'll assume 32-bit x86 using the modern SYSENTER / SYSEXIT x86 instructions

64-bit code is similar

However, details change over time, so take this as an example - not a debugging guide

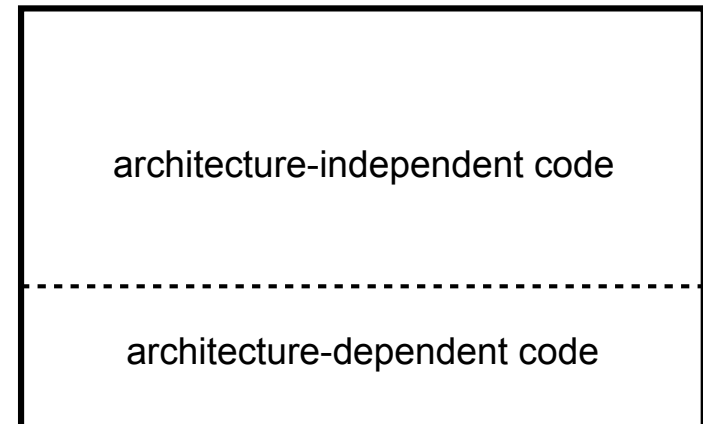
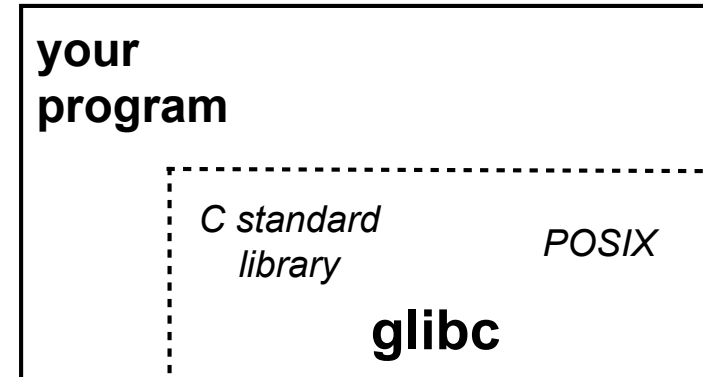
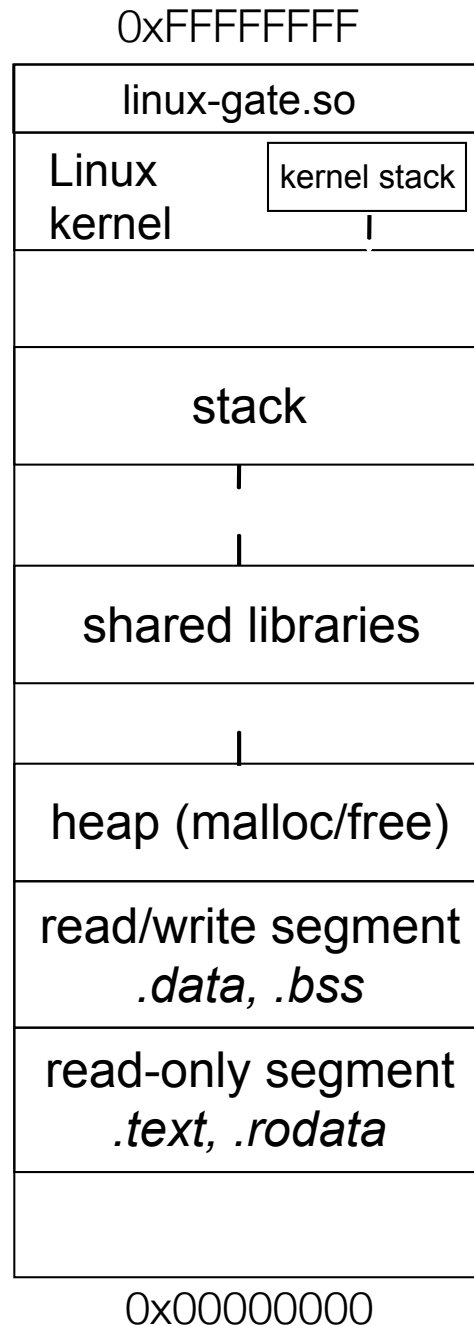




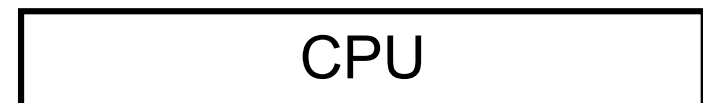
# Details on x86 / Linux

Remember our process address space picture

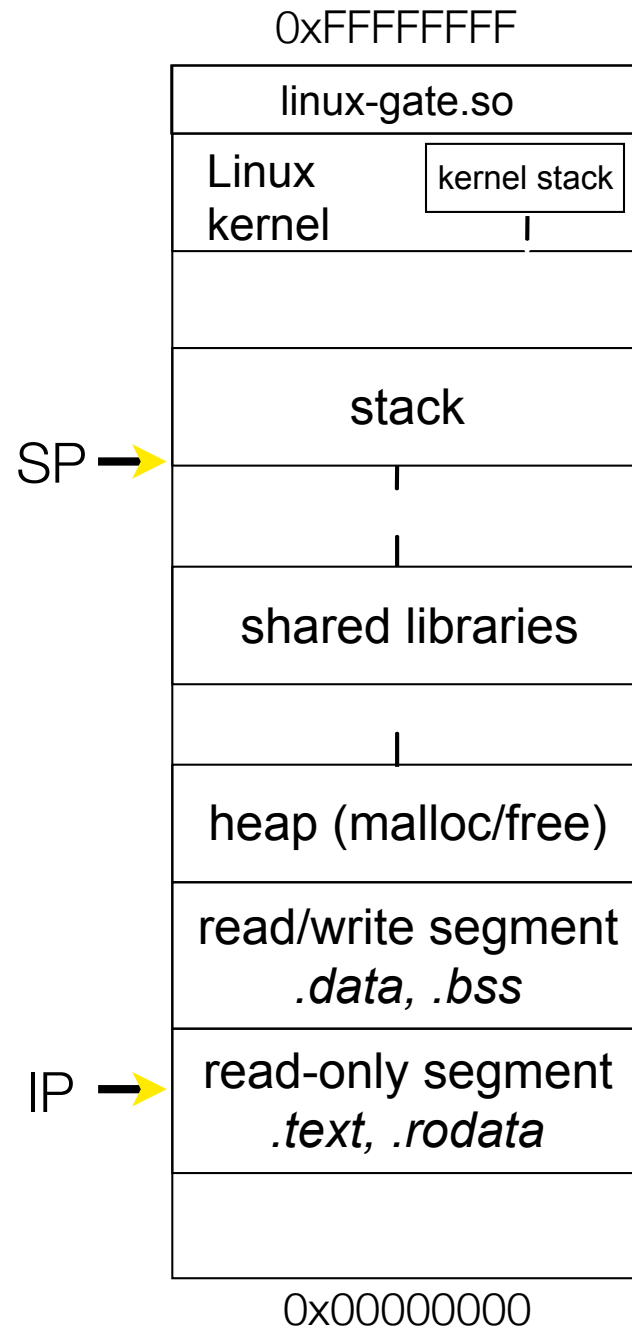
let's add some details



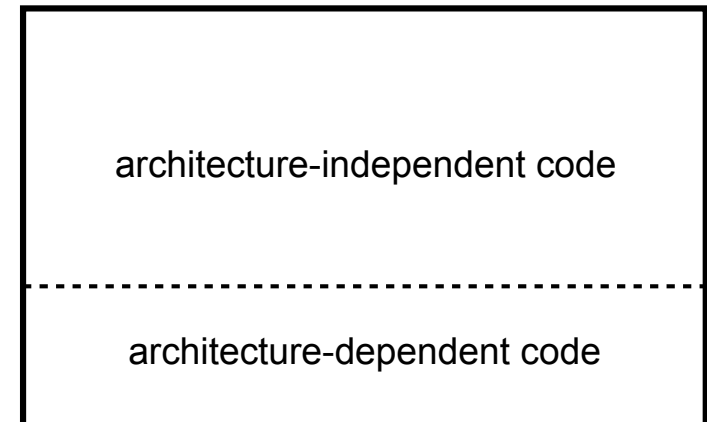
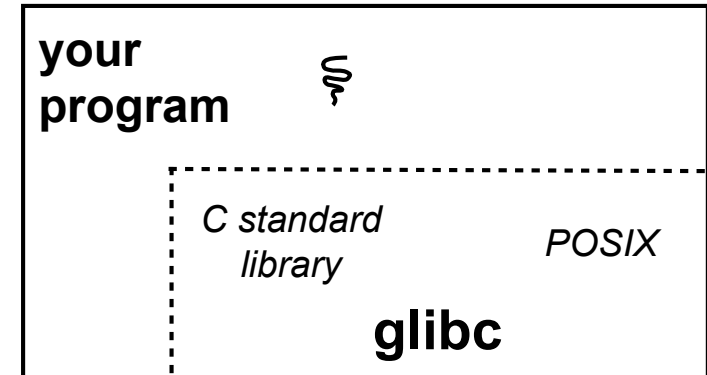
Linux kernel



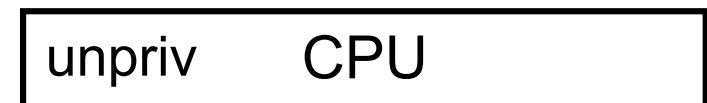
# Details on x86 / Linux



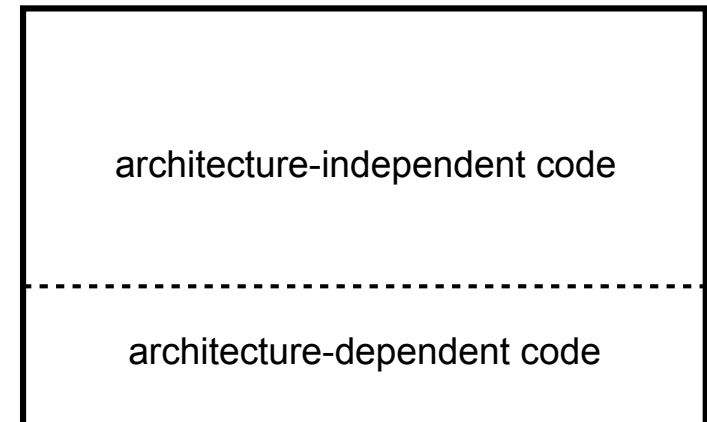
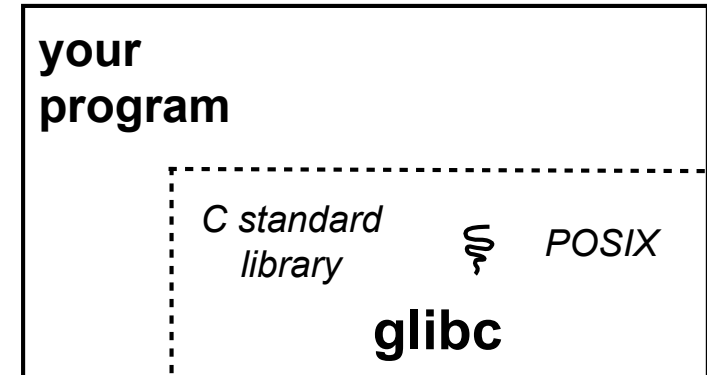
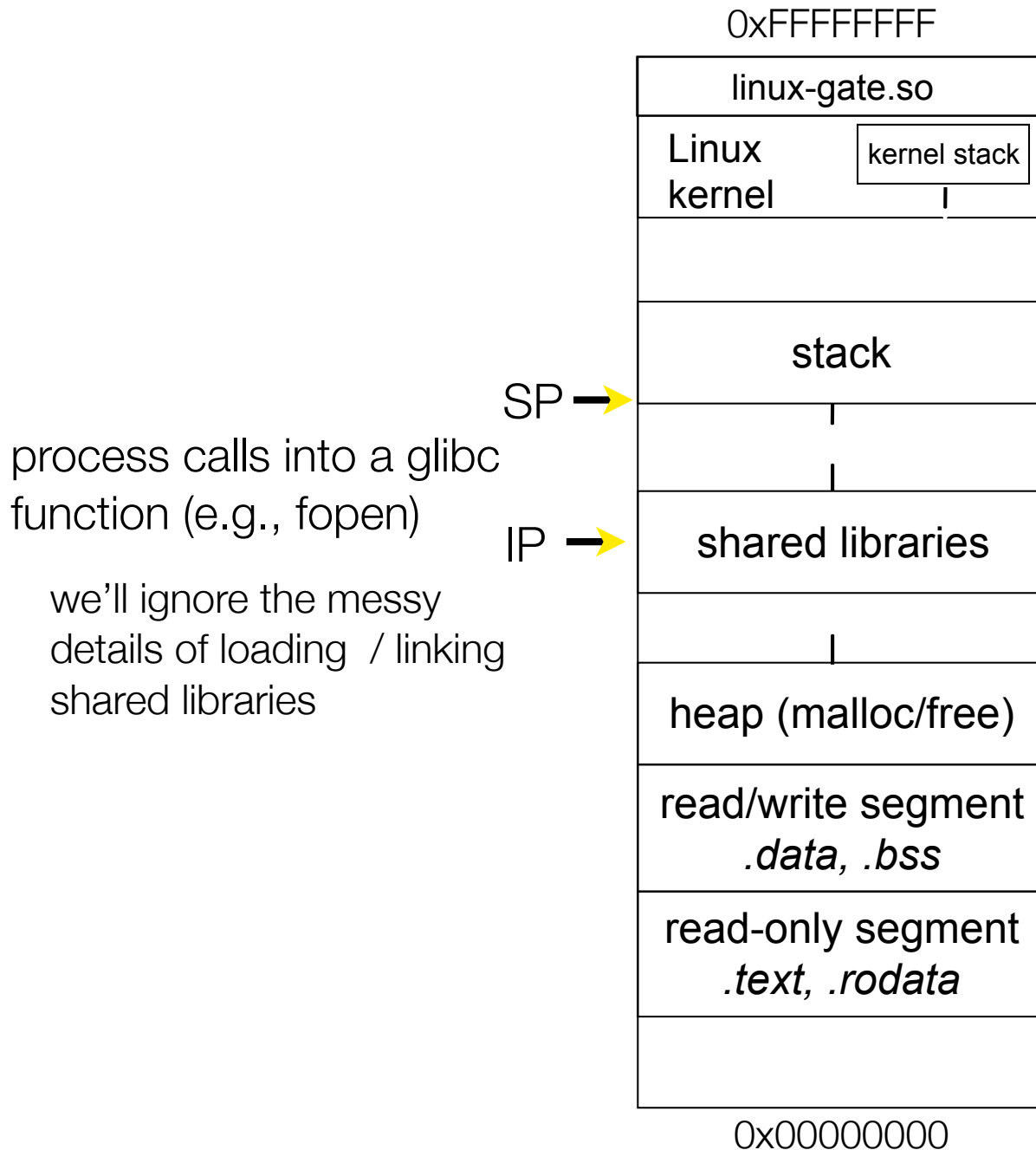
process is executing  
your program code



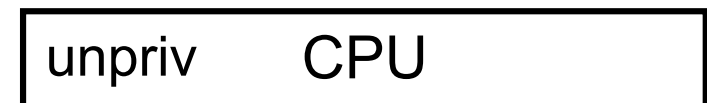
Linux kernel



# Details on x86 / Linux



Linux kernel



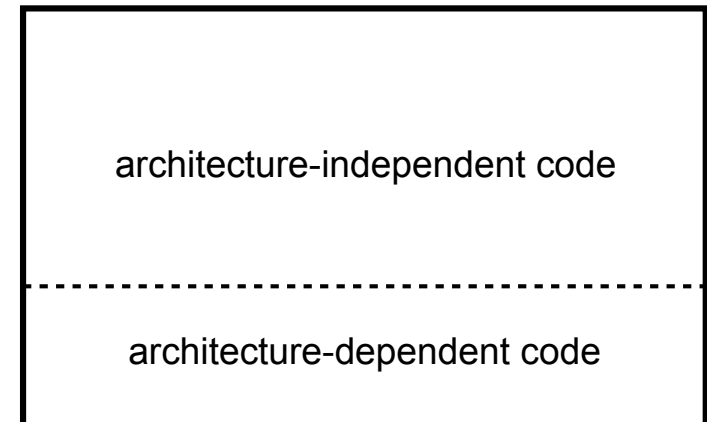
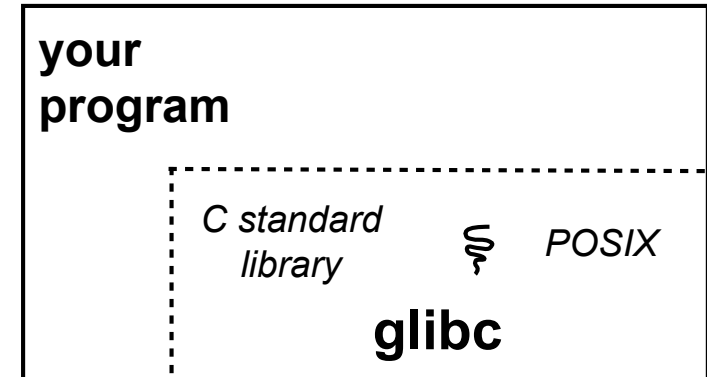
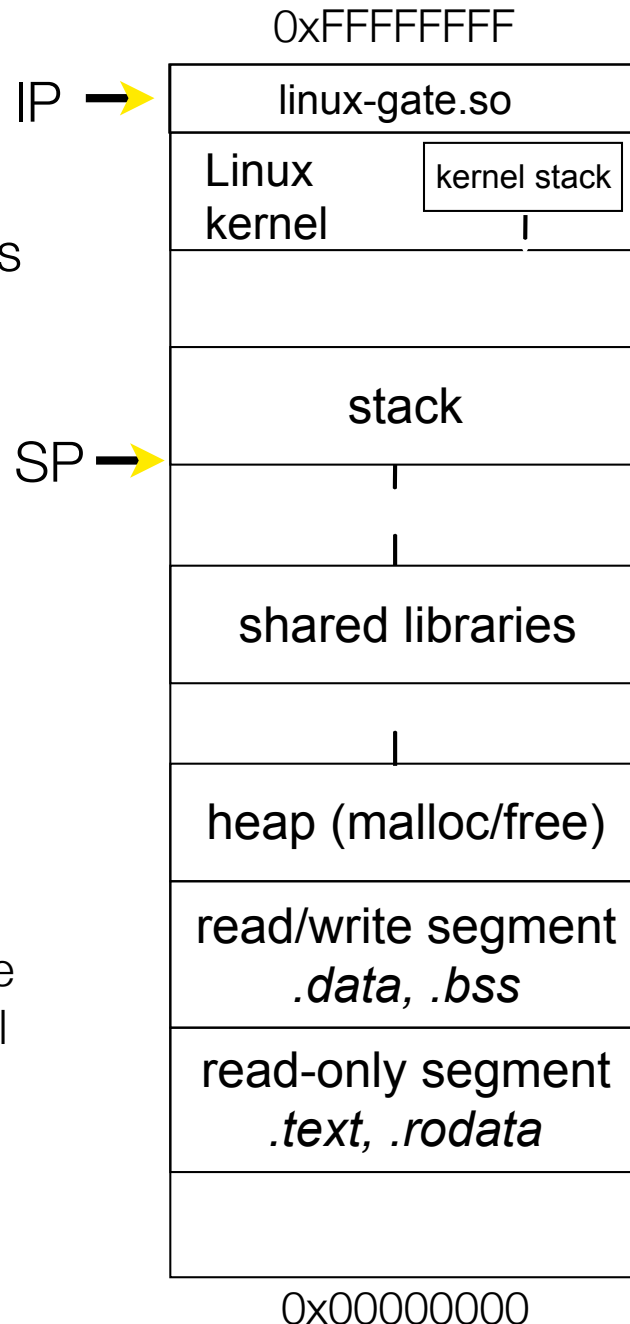
# Details on x86 / Linux

glibc begins the process of invoking a Linux system call

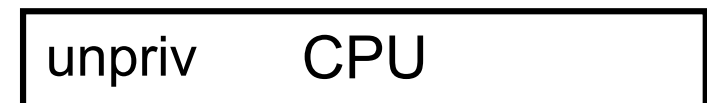
glibc's `fopen()` likely invokes Linux's `open()` system call

puts the system call # and arguments into registers

uses the **call** x86 instruction to call into the routine `__kernel_vsyscall` located in `linux-gate.so`



Linux kernel



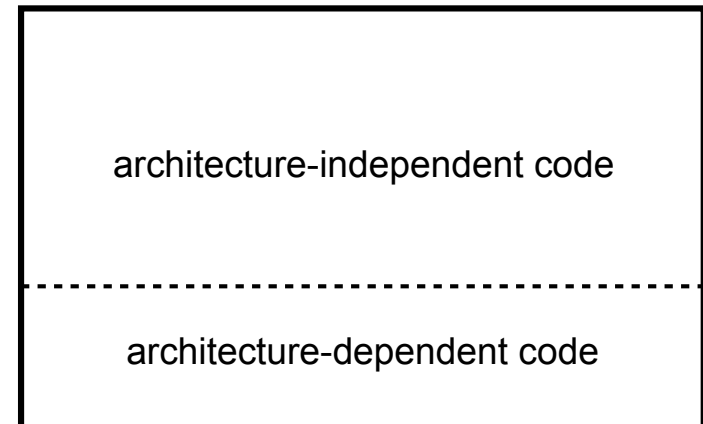
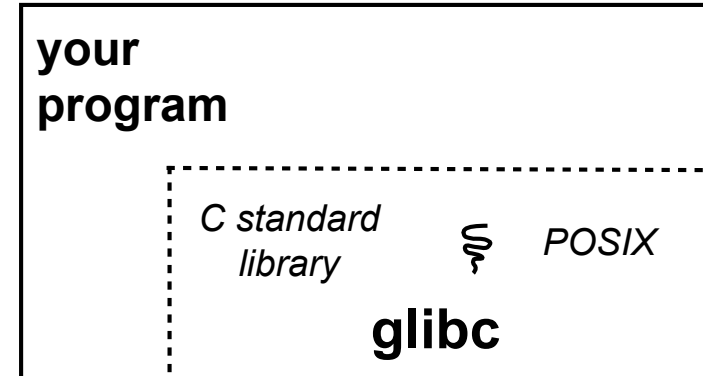
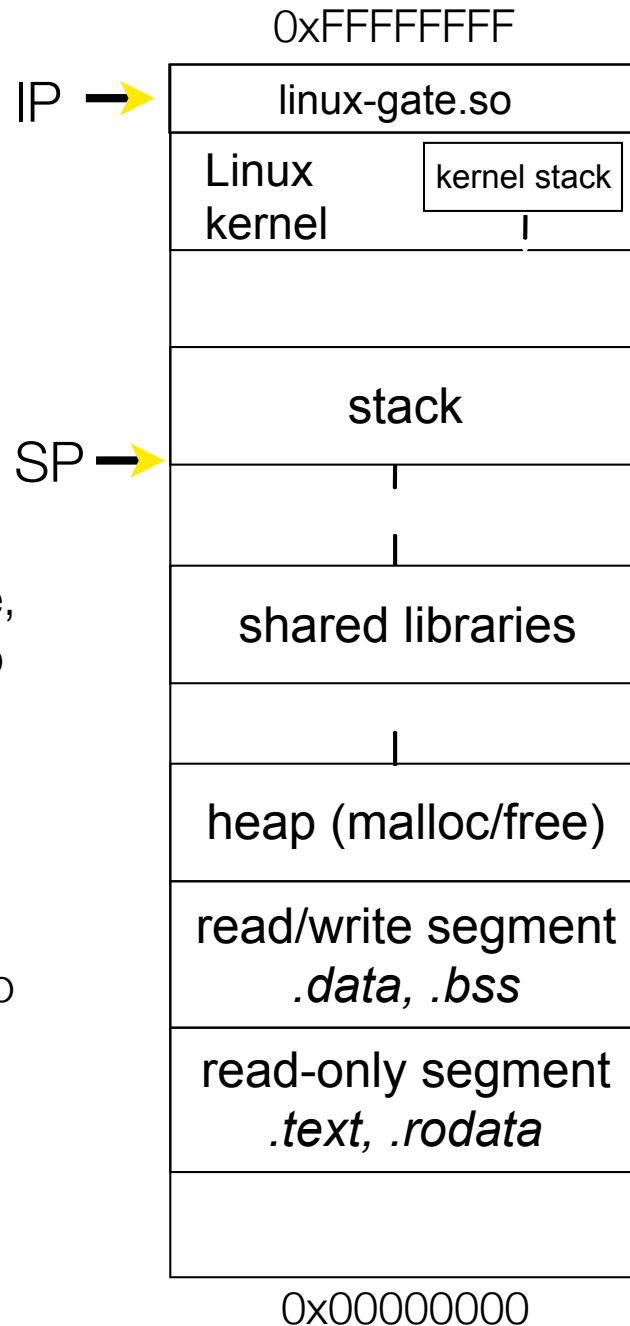
# Details on x86 / Linux

linux-gate.so is a **vdso**

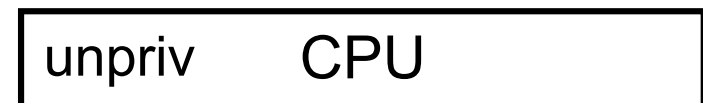
a virtual dynamically linked shared object

is a kernel-provided shared library, i.e., is not associated with a .so file, but rather is conjured up by the kernel and plunked into a process's address space

provides the intricate machine code needed to trigger a system call



Linux kernel



# Details on x86 / Linux

linux-gate.so eventually invokes the SYSENTER x86 instruction

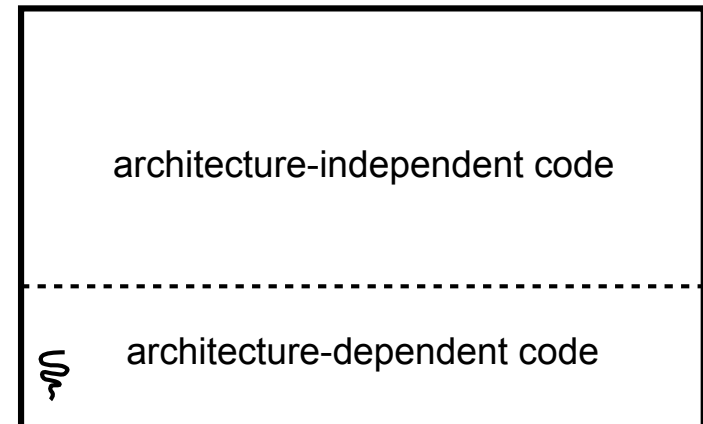
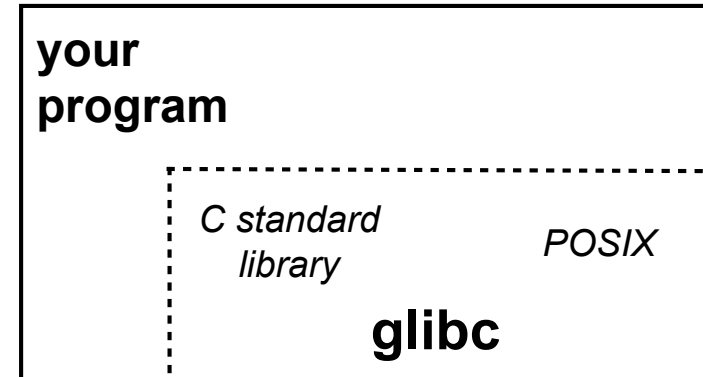
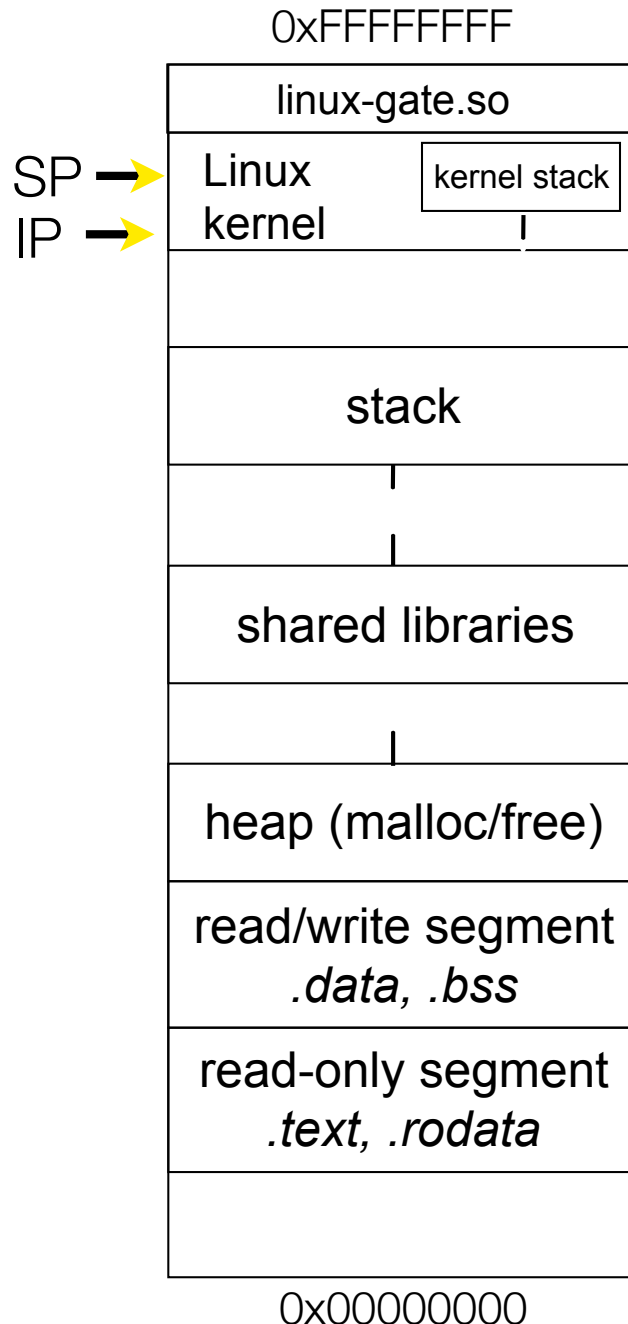
SYSENTER is x86's "fast system call" instruction

it has several side-effects

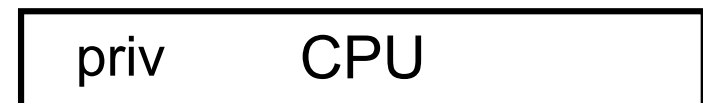
causes the CPU to raise its privilege level

traps into the Linux kernel by changing the SP, IP to a previously determined location

changes some segmentation related registers (see cse451)



Linux kernel



# Details on x86 / Linux

The kernel begins executing code at the SYSENTER entry point

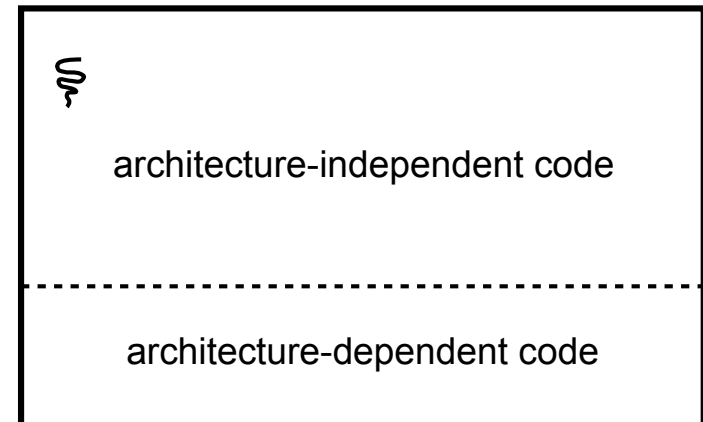
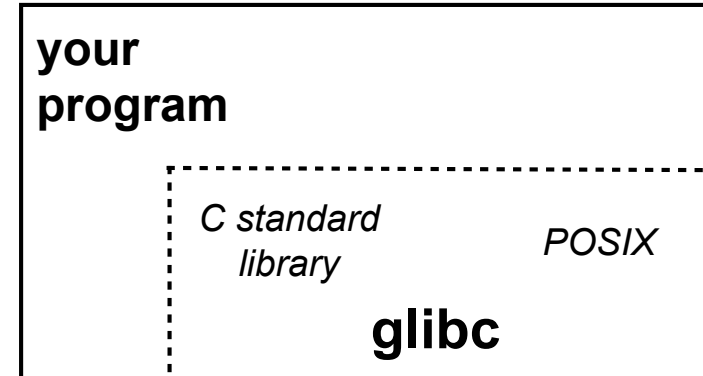
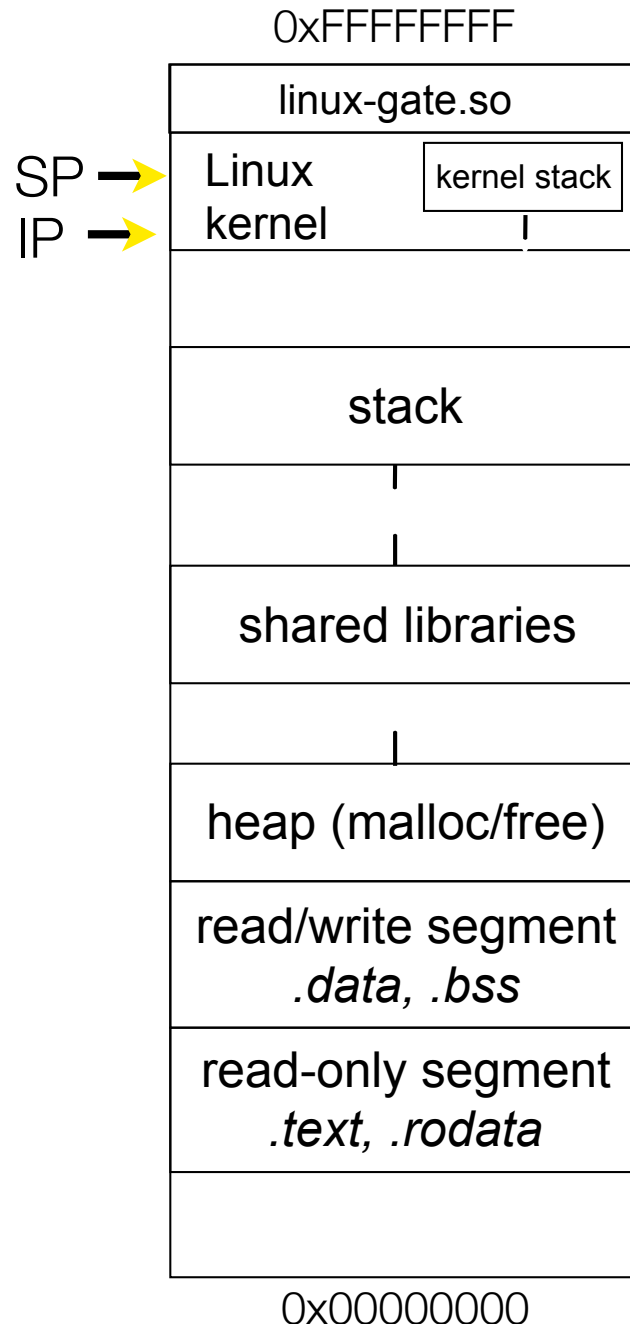
is in the architecture-dependent part of Linux

it's job is to:

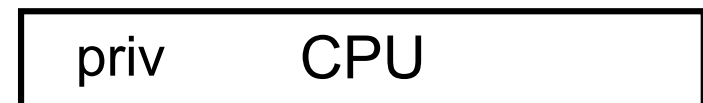
look up the system call number in a system call dispatch table

call into the address stored in that table entry; this is Linux's system call handler

for open, the handler is named sys\_open, and is system call #5



Linux kernel



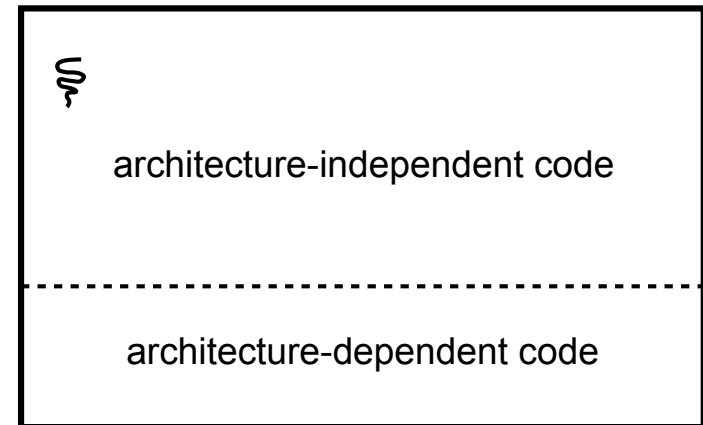
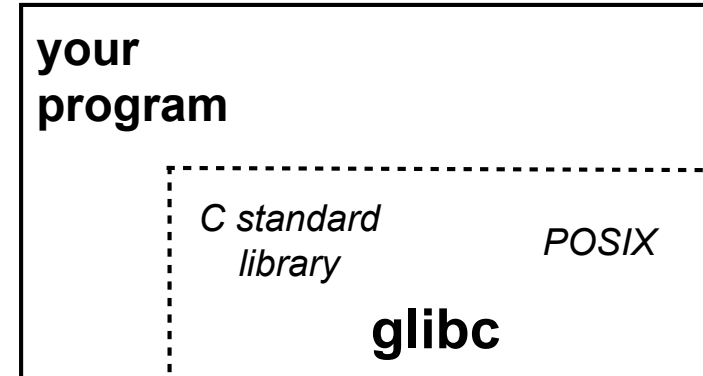
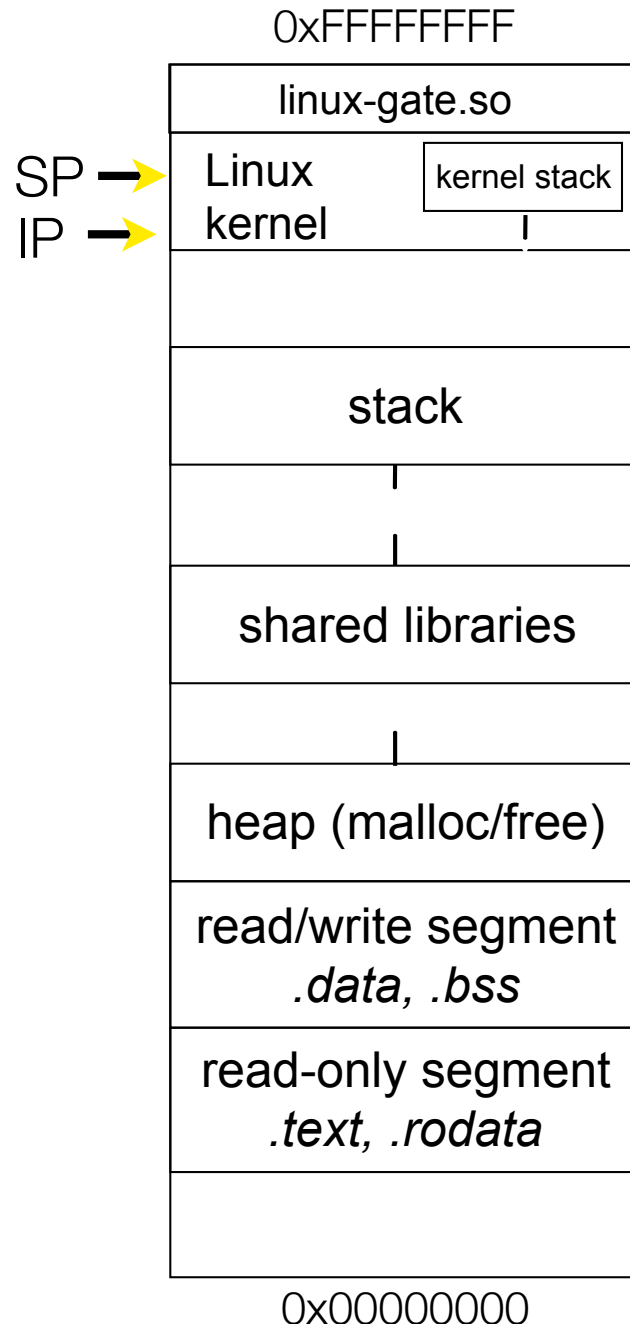
# Details on x86 / Linux

The system call handler executes

what it does is system-call specific, of course

it may take a long time to execute, especially if it has to interact with hardware

Linux may choose to context switch the CPU to a different runnable process

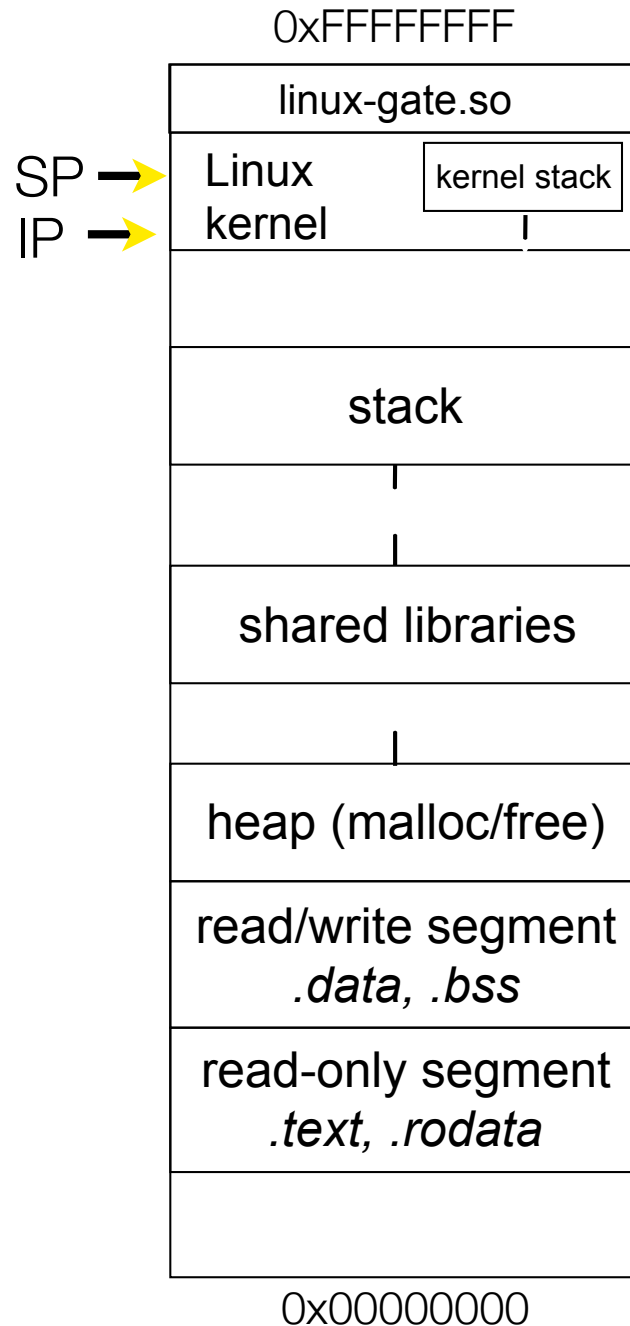


Linux kernel





# Details on x86 / Linux

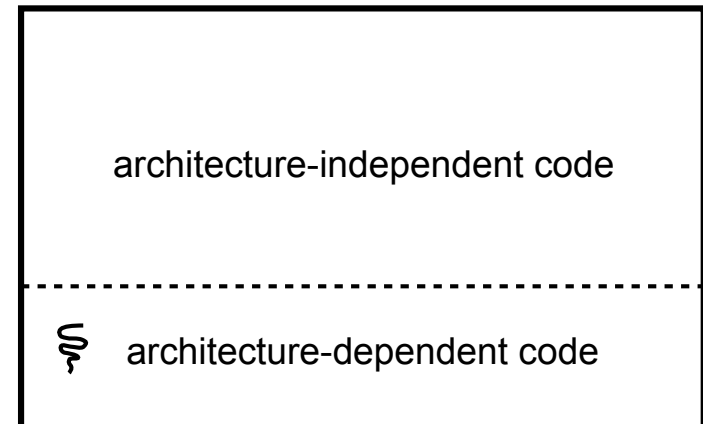
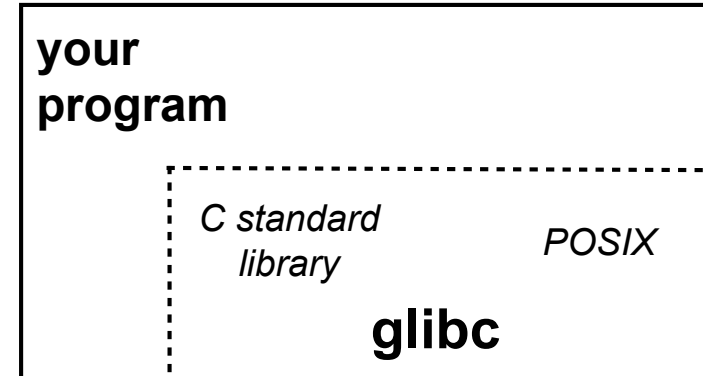


Eventually, the system call handler finishes

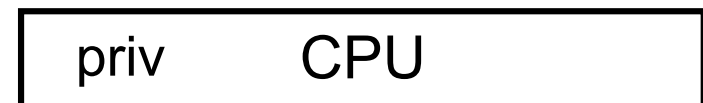
returns back to the system call entry point

places the system call's return value in the appropriate register

calls SYSEXIT to return to the user-level code



Linux kernel



# Details on x86 / Linux

SYSEXIT transitions the processor back to user-mode code

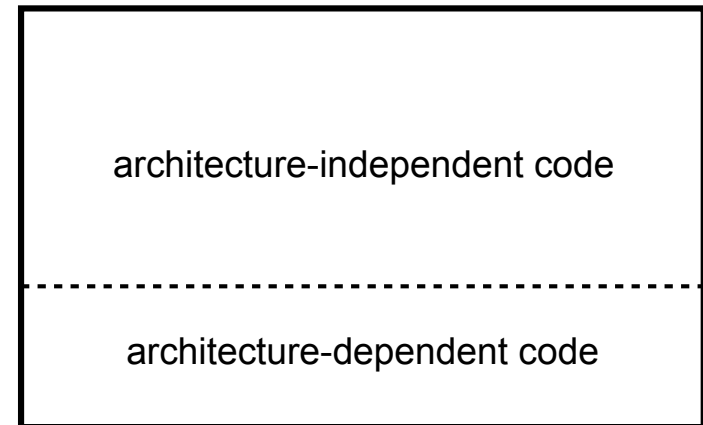
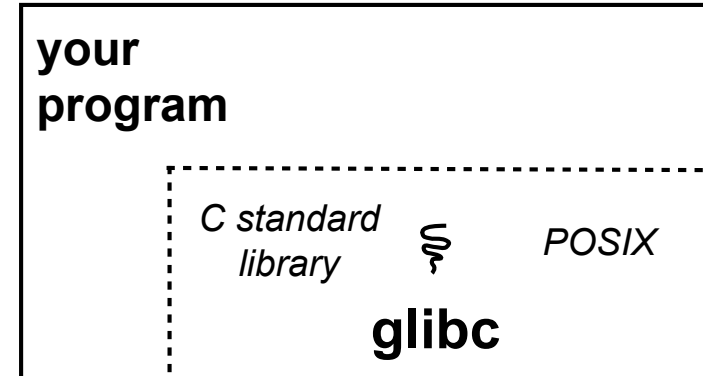
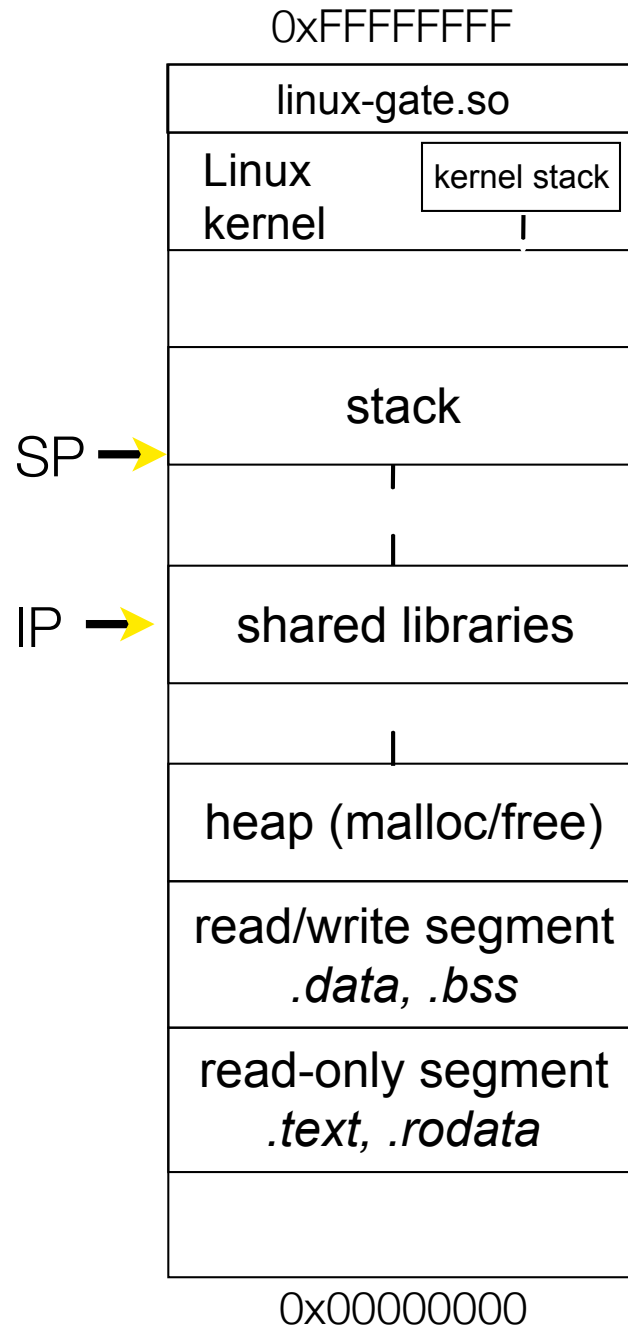
has several side-effects

restores the IP, SP to user-land values

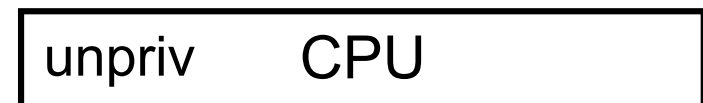
sets the CPU back to unprivileged mode

changes some segmentation related registers (see cse451)

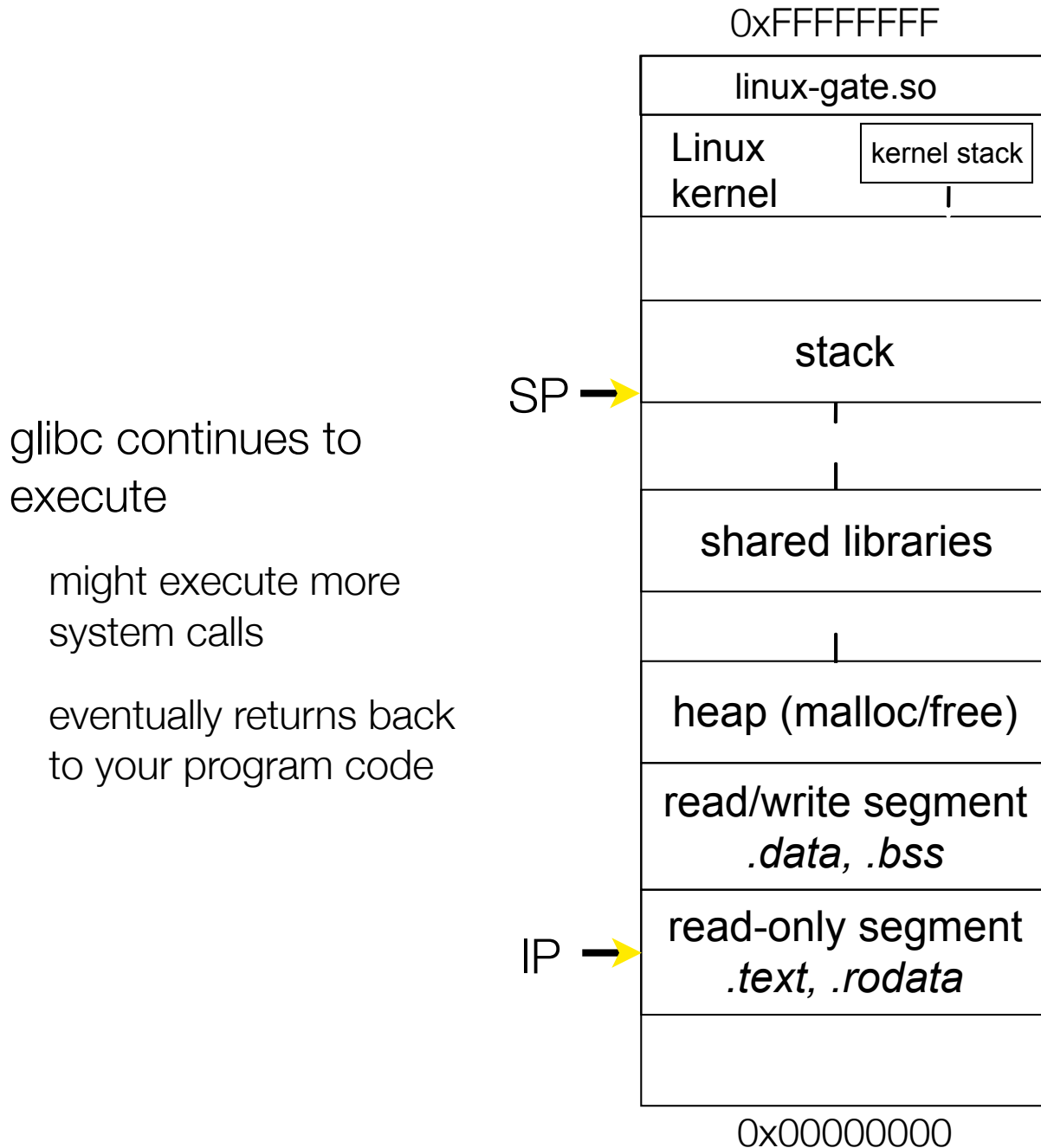
returns the processor back to glibc



Linux kernel



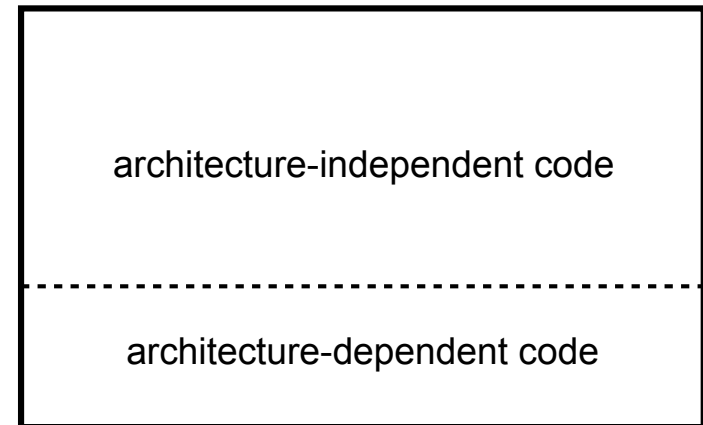
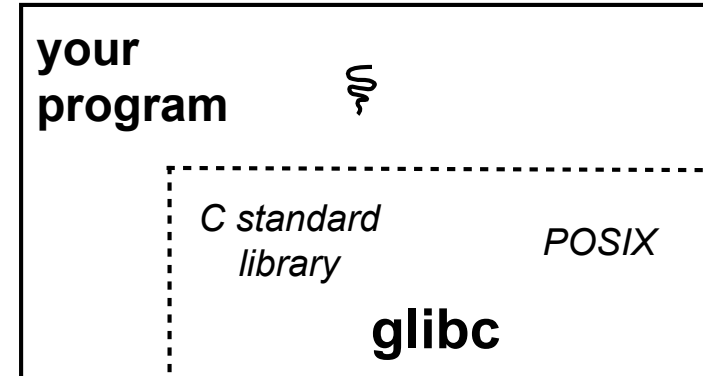
# Details on x86 / Linux



glibc continues to execute

might execute more system calls

eventually returns back to your program code



Linux kernel



# strace

A useful Linux utility that shows the sequence of system calls that a process makes:

```
bash$ strace ls 2>&1 | less
```

```
[005c7424] execve("/bin/ls", ["ls"], [/* 47 vars */]) = 0
[003caffd] brk(0) = 0x9376000
[003cc3c3] mmap2(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) =
0xb7800000
[003cc2c1] access("/etc/ld.so.preload", R_OK) = -1 ENOENT (No such file or directory)
[003cc184] open("/etc/ld.so.cache", O_RDONLY) = 3
[003cc14e] fstat64(3, {st_mode=S_IFREG|0644, st_size=92504, ...}) = 0
[003cc3c3] mmap2(NULL, 92504, PROT_READ, MAP_PRIVATE, 3, 0) = 0xb77e9000
[003cc1bd] close(3) = 0
[003cc184] open("/lib/libselinux.so.1", O_RDONLY) = 3
[003cc204] read(3, "\177ELF\1\1\1\0\0\0\0\0\0\0\0\3\0\3\0\1\0\0"... , 512) = 512
[003cc14e] fstat64(3, {st_mode=S_IFREG|0755, st_size=122420, ...}) = 0
[003cc3c3] mmap2(0x6d6000, 125948, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) =
0x6d6000
[003cc3c3] mmap2(0x6f3000, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP
_DENYWRITE, 3, 0x1c) = 0x6f3000
[003cc1bd] close(3) = 0
[003cc184] open("/lib/librt.so.1", O_RDONLY) = 3
[003cc204] read(3, "\177ELF\1\1\1\0\0\0\0\0\0\0\0\3\0\3\0\1\0\0\0\200X[\0004\0\0\0"... ,
512) = 512
... etc.
```

# strace

A useful Linux utility that shows the sequence of system calls that a process makes:

```
bash$ strace ls 2>&1 | less
```

```
...
```

```
[00110424] open(".", O_RDONLY|O_NONBLOCK|O_LARGEFILE|O_DIRECTORY|O_CLOEXEC) = 3
[00110424] fcntl64(3, F_GETFD) = 0x1 (flags FD_CLOEXEC)
[00110424] getdents64(3, /* 6 entries */, 32768) = 184
[00110424] getdents64(3, /* 0 entries */, 32768) = 0
[00110424] close(3) = 0
[00110424] fstat64(1, {st_mode=S_IFIFO|0600, st_size=0, ...}) = 0
[00110424] mmap2(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) =
0xb77ff000
[00110424] write(1, "bomstrip.py\nmountlaptop.sh\nteste"..., 43
bomstrip.py
mountlaptop.sh
tester
tester.c
) = 43
[00110424] close(1) = 0
[00110424] munmap(0xb77ff000, 4096) = 0
[00110424] close(2) = 0
[00110424] exit_group(0) = ?
```

# If you're curious

Download the Linux kernel source code

available from <http://www.kernel.org/>

Take a look at:

`arch/x86/kernel/syscall_table_32.S` [system call table]

`arch/x86/syscalls/syscall_32.tbl` in more recent versions

`arch/x86/kernel/entry_32.S` [SYSENTER entry point and more]

`arch/x86/vdso/vdso32/sysenter.S` [user-land vdso]

And: [http://articles.manugarg.com/systemcallinlinux2\\_6.html](http://articles.manugarg.com/systemcallinlinux2_6.html)

# Also...

man, section 2: Linux system calls

man 2 intro

man 2 syscalls (or [look online here](#))

man, section 3: glibc / libc library functions

man 3 intro (or [look online here](#))

*The book: [The Linux Programming Interface](#) by Michael Kerrisk (keeper of the Linux man pages)*

If you want a copy: go to the book web site ([man7.org/tlpl](http://man7.org/tlpl)), get discount code there, then order from the publisher

Book + ebook for cost of printed copy from Amazon

# Let's do some file I/O...

We'll start by using C's standard library

these functions are implemented in glibc on Linux

they are implemented using Linux system calls

C's stdio defines the notion of a **stream**

a stream is a way of reading or writing a sequence of characters from/to a device

a stream can be either *text* or *binary*; Linux does not distinguish

a stream is *buffered* by default; libc reads ahead of you

three streams are provided by default: **stdin**, **stdout**, **stderr**

you can open additional streams to read/write to files



# Using C streams

```
read_example.c
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>

#define READBUFSIZE 128
int main(int argc, char **argv) {
    FILE *f;
    char readbuf[READBUFSIZE];
    size_t readlen;

    if (argc != 2) {
        fprintf(stderr, "usage: ./fread_example filename\n");
        return EXIT_FAILURE; // defined in stdlib.h
    }

    // Open, read, and print the file
    f = fopen(argv[1], "rb"); // "rb" --> read, binary mode
    if (f == NULL) {
        fprintf(stderr, "%s -- ", argv[1]);
        perror("fopen failed -- ");
        return EXIT_FAILURE;
    }

    // Read from the file, write to stdout.
    while ((readlen = fread(readbuf, 1, READBUFSIZE, f)) > 0)
        fwrite(readbuf, 1, readlen, stdout);
    fclose(f);
    return EXIT_SUCCESS; // defined in stdlib.h
}
```

printf(...) is equivalent to fprintf(stdout, ...)

stderr is a stream for printing error output to a console

fopen opens a stream to read or write a file

perror writes a string describing the last error to stderr

stdout is for printing non-error output to the console

# Writing is easy too

*see cp\_example.c*

# A gotcha

By default, stdio turns on **buffering** for streams

data written by `fwrite( )` is copied into a buffer allocated by stdio inside your process's address space

at some point, the buffer will be drained into the destination

when you call `fflush( )` on the stream

when the buffer size is exceeded (*often 1024 or 4096 bytes*)

for `stdout` to a console, when a newline is written (*"line buffered"*)

when you call `fclose( )` on the stream

when your process exits gracefully (*`exit( )` or `return from main( )`*)

# Why is this a gotcha?

What happens if...

your computer loses power before the buffer is flushed?

your program assumes data is written to a file, and it signals another program to read it?

What are the performance implications?

data is ***copied*** into the stdio buffer

consumes CPU cycles and memory bandwidth

can potentially slow down high performance applications, like a web server or database (“zero copy”)

# What to do about it

Turn off buffering with **setbuf( )**

this, too, may cause performance problems

e.g., if your program does many small `fwrite( )`'s, each of which will now trigger a system call into the Linux kernel

Use a different set of system calls

POSIX provides `open( )`, `read( )`, `write( )`, `close( )`, and others  
no buffering is done at the user level

but...what about the layers below?

the OS caches disk reads and writes in the FS *buffer cache*  
disk controllers have caches too!

# Exercise 1

Write a program that:

uses `argc/argv` to receive the name of a text file

reads the contents of the file a line at a time

parses each line, converting text into a `uint32_t`

builds an array of the parsed `uint32_t`'s

sorts the array

prints the sorted array to `stdout`

hints: use “man” to read about `getline`, `sscanf`, `realloc`, and `qsort`

```
bash$ cat in.txt
1213
3231
000005
52
bash$ ex1 in.txt
5
52
1213
3231
bash$
```

# Exercise 2

Write a program that:

loops forever; in each loop, it:

prompts the user to input a filename

reads from stdin to receive a filename

opens and reads the file, and prints its contents to stdout, in the format shown on the right

hints:

use “man” to read about `fgets`

or if you’re more courageous, try “man 3 readline” to learn about `libreadline.a`, and google to learn how to link to it

```
0000000 50 4b 03 04 14 00 00 00 00 00 9c 45 26 3c f1 d5
0000010 68 95 25 1b 00 00 25 1b 00 00 0d 00 00 00 43 53
0000020 45 6c 6f 67 6f 2d 31 2e 70 6e 67 89 50 4e 47 0d
0000030 0a 1a 0a 00 00 00 0d 49 48 44 52 00 00 00 91 00
0000040 00 00 91 08 06 00 00 00 c3 d8 5a 23 00 00 00 09
0000050 70 48 59 73 00 00 0b 13 00 00 0b 13 01 00 9a 9c
0000060 18 00 00 0a 4f 69 43 43 50 50 68 6f 74 6f 73 68
0000070 6f 70 20 49 43 43 20 70 72 6f 66 69 6c 65 00 00
0000080 78 da 9d 53 67 54 53 e9 16 3d f7 de f4 42 4b 88
0000090 80 94 4b 6f 52 15 08 20 52 42 8b 80 14 91 26 2a
00000a0 21 09 10 4a 88 21 a1 d9 15 51 c1 11 45 45 04 1b
00000b0 c8 a0 88 03 8e 8e 80 8c 15 51 2c 0c 8a 0a d8 07
00000c0 e4 21 a2 8e 83 a3 88 8a ca fb e1 7b a3 6b d6 bc
...etc.
```

See you on Wednesday!