System Calls Continued & C++ Intro

CSE 333

Instructor: Alex Sanchez-Stern

Teaching Assistants:

Justin Tysdal

Sayuj Shahi

Nicholas Batchelder

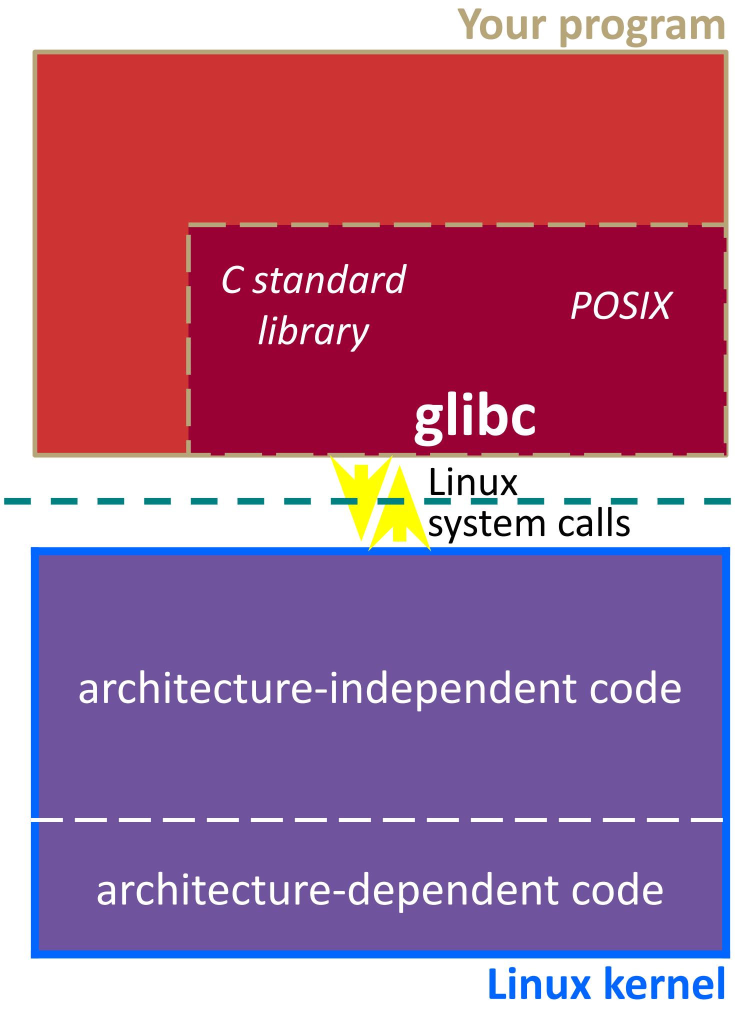
Leanna Mi Nguyen

Administrivia

- Homework 1 is due **tonight at 11pm**
- Exercise 7 was due this morning
- Exercise 8 is posted this morning, but not due until **Wednesday**
  - It's on C++, and we'll be finishing our C++ intro on Monday
- Don't forget to use cpplint on all your assignments!
  - Linter errors are correctness errors in this course
- Homework 2 starter code is being pushed **tomorrow**

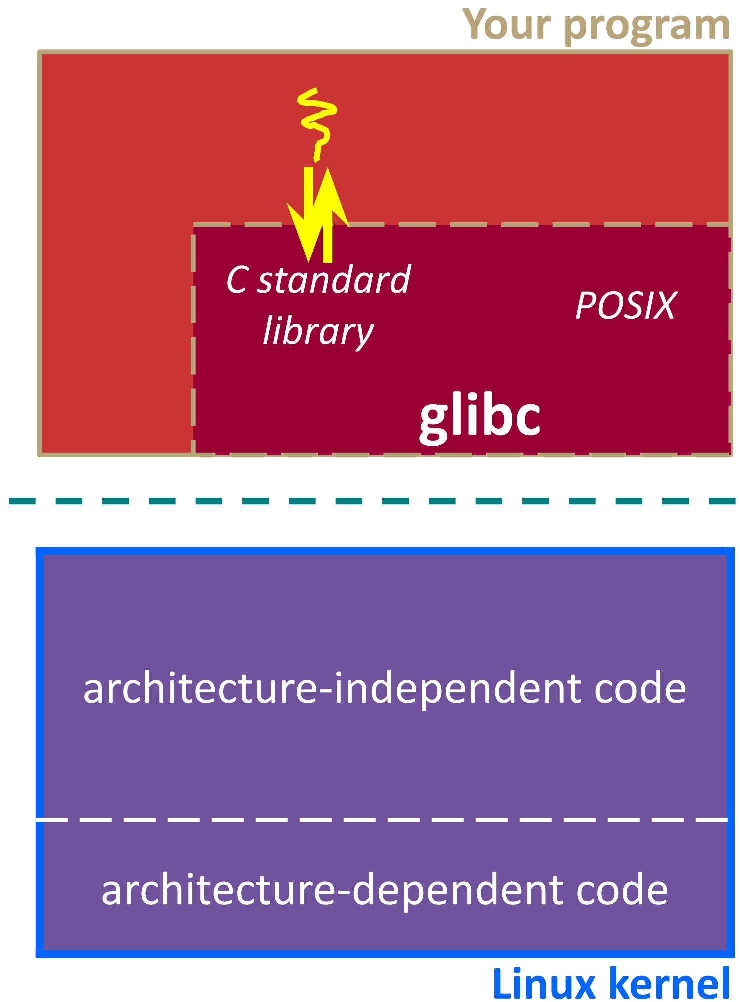
Details on x86/Linux

- A more accurate picture:
  - Consider a typical Linux process
  - Its thread of execution can be in one of 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 x86-64 code



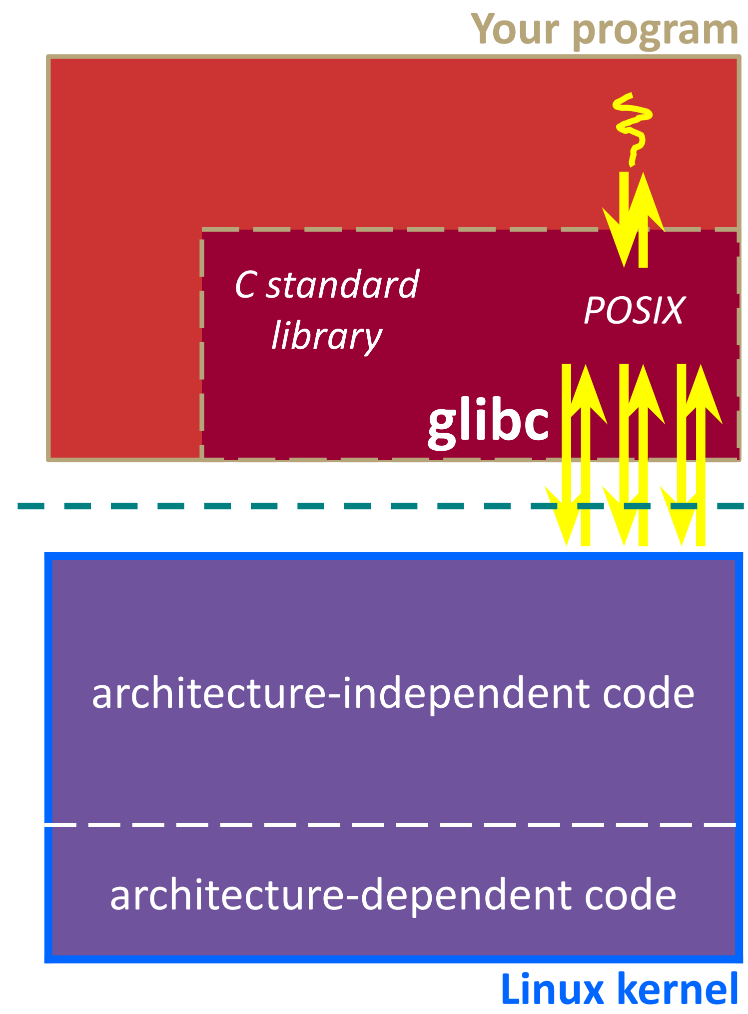
Details on x86/Linux

- Some routines your program invokes may be entirely handled by `glibc` without involving the kernel
  - *e.g.* `strcmp()` from `stdio.h`
  - There is some initial overhead when invoking functions in dynamically linked libraries (during loading)
    - But after symbols are resolved, invoking `glibc` routines is basically as fast as a function call within your program itself!



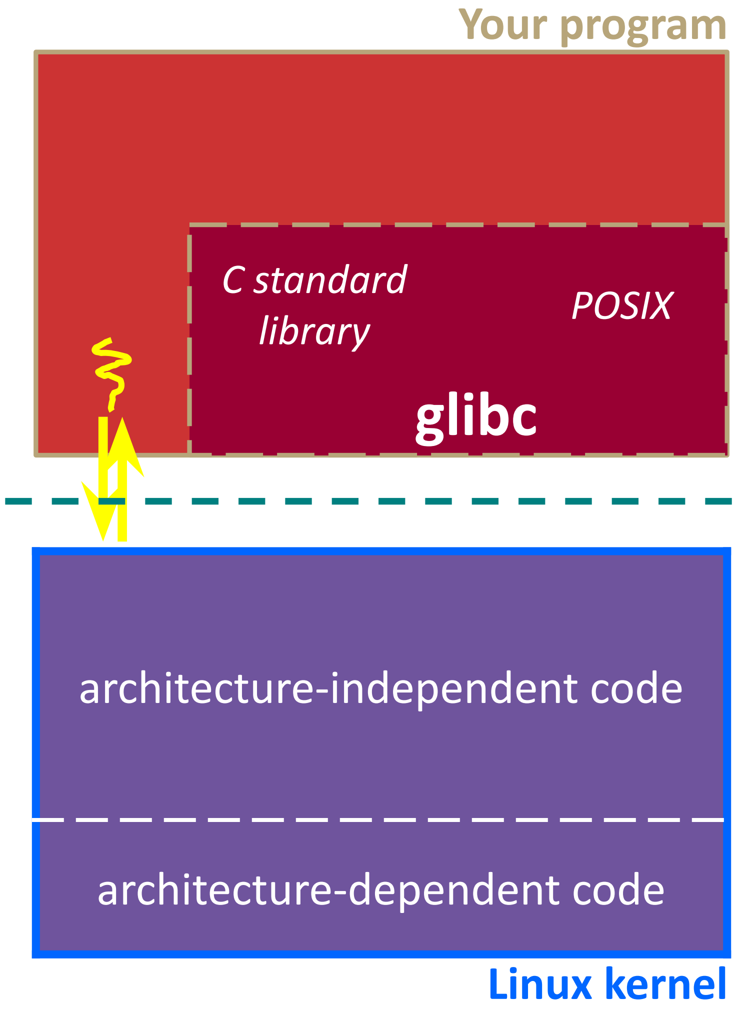
Details on x86/Linux

- Some routines may be handled by `glibc`, but they in turn invoke Linux system calls
  - *e.g.* POSIX wrappers around Linux `syscall`s
    - 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()`, `close()`, `write()`, etc.



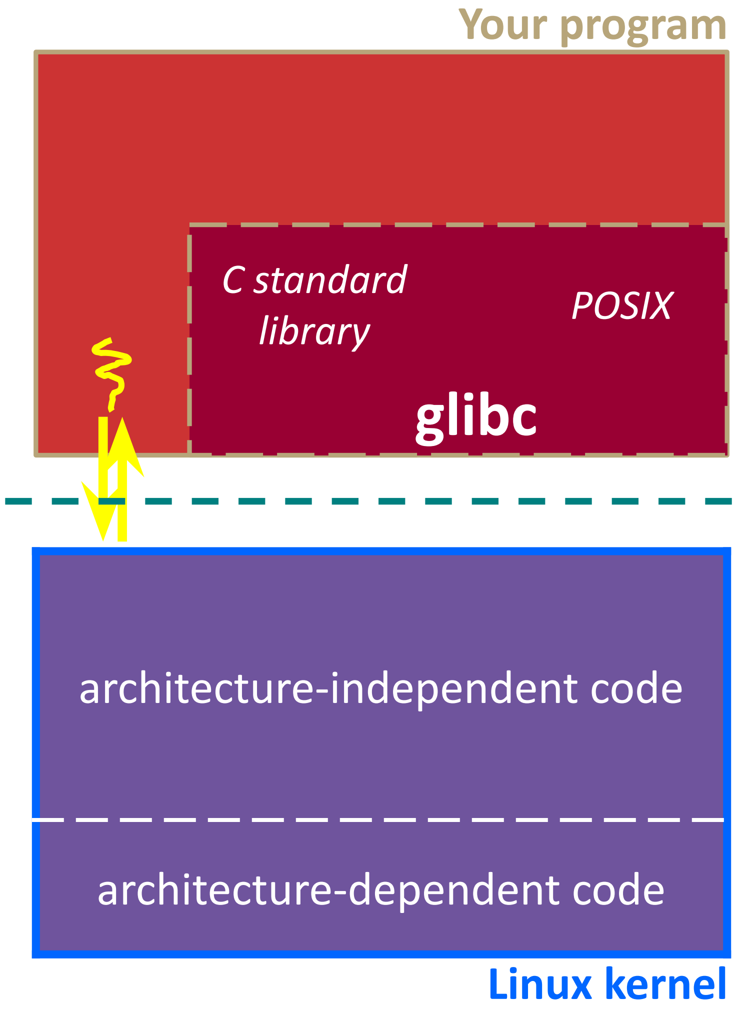
Details on x86/Linux

- Your program can choose to directly invoke Linux system calls as well
  - Nothing is forcing 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
    - (And won't be portable to non-Unix systems like Windows that run standard C on top of their own, different syscalls)



Details on x86/Linux

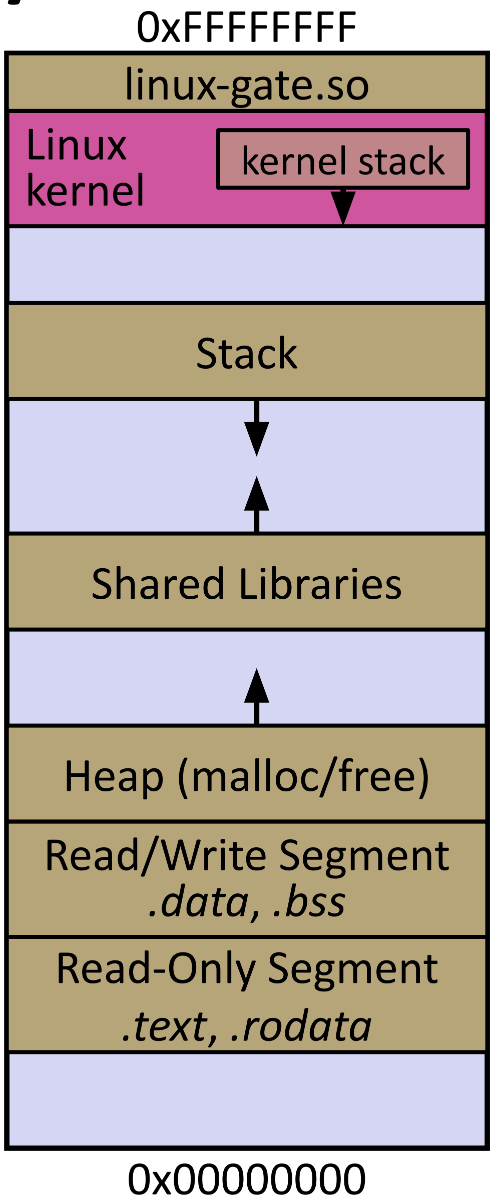
- Let's walk through how a Linux system call actually works
  - We'll assume *32-bit x86* using the modern `SYSENTER` / `SYSEXIT` x86 instructions
    - x86-64 code is similar, though details always change over time, so take this as an example – not a debugging guide

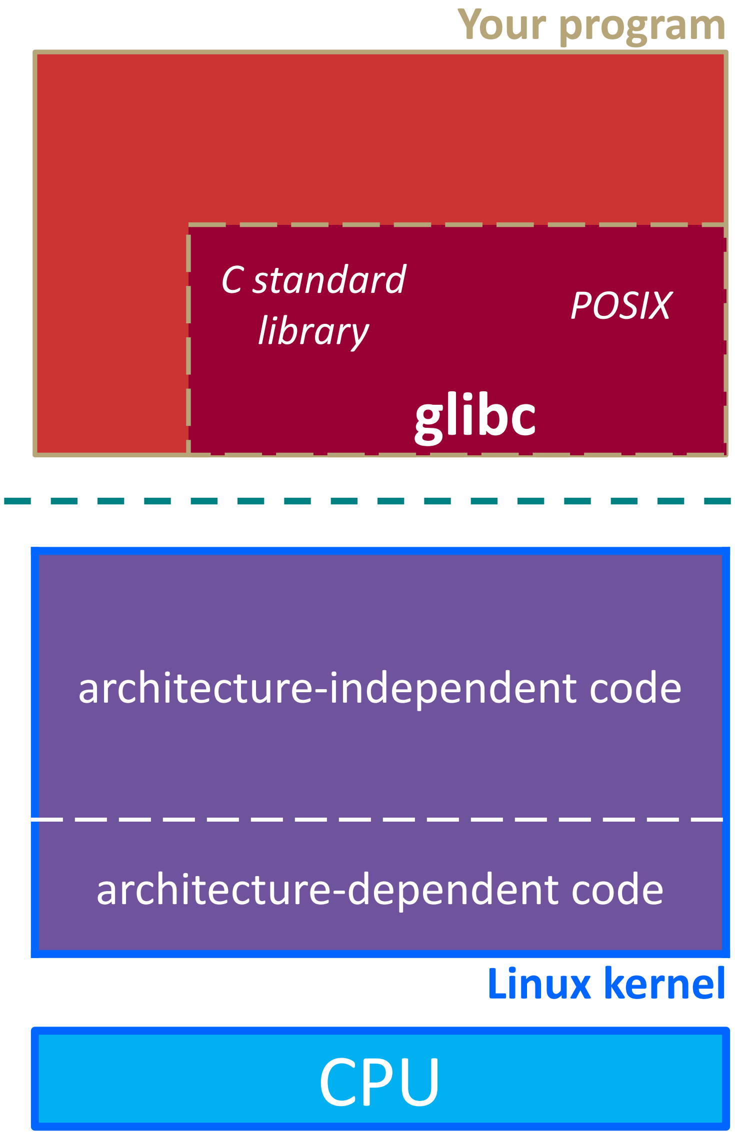


Details on x86/Linux

Remember our process address space picture?

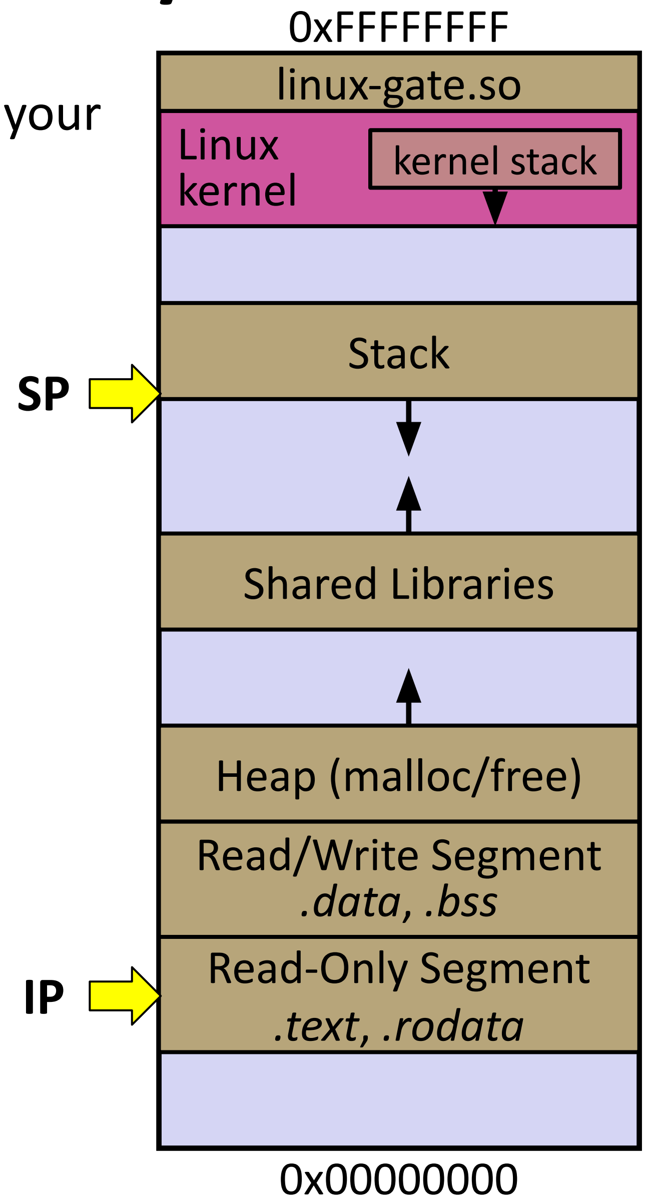
- Let's add some details:

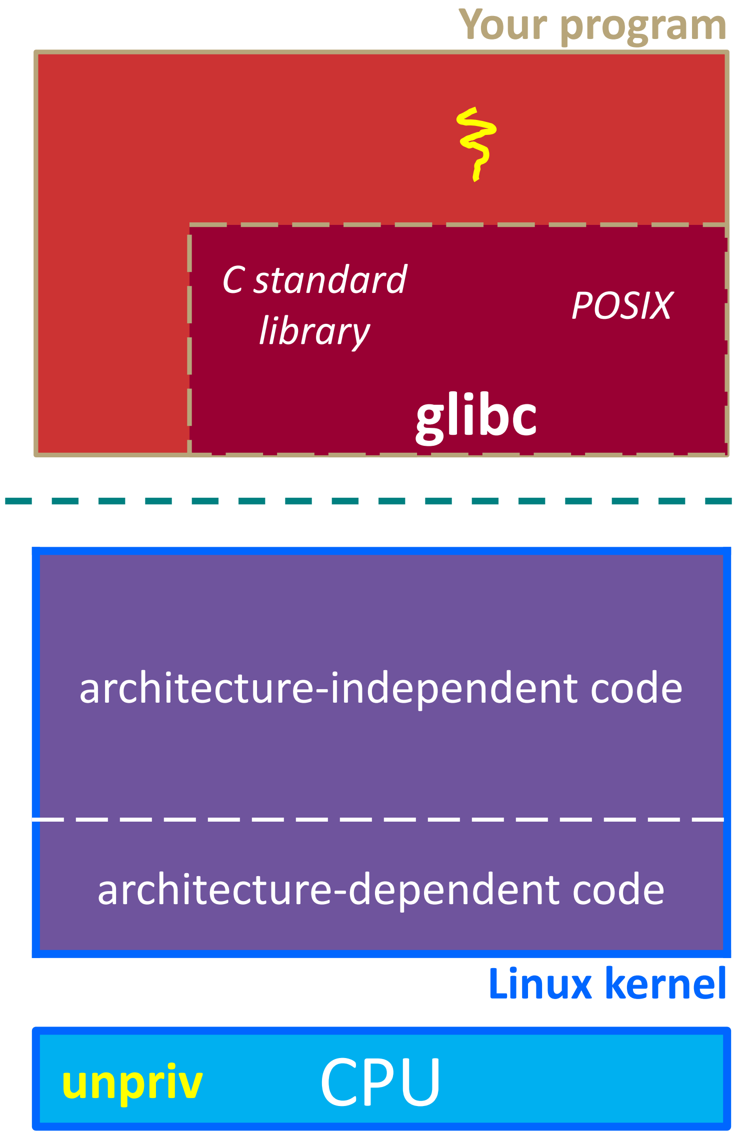




Details on x86/Linux

Process is executing your program code

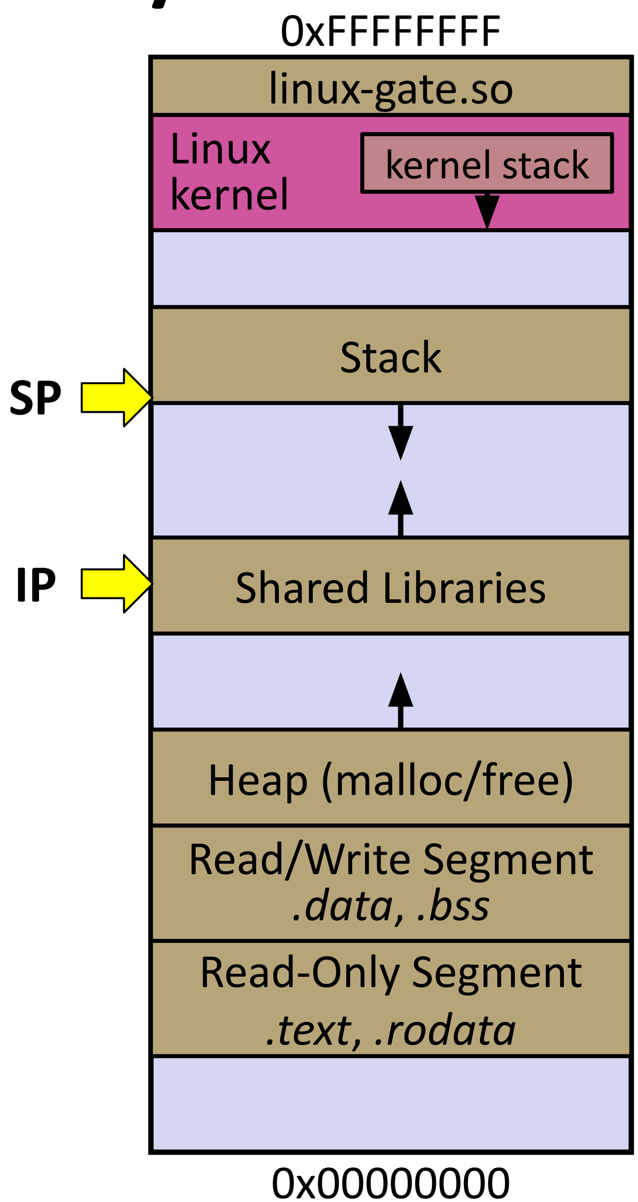


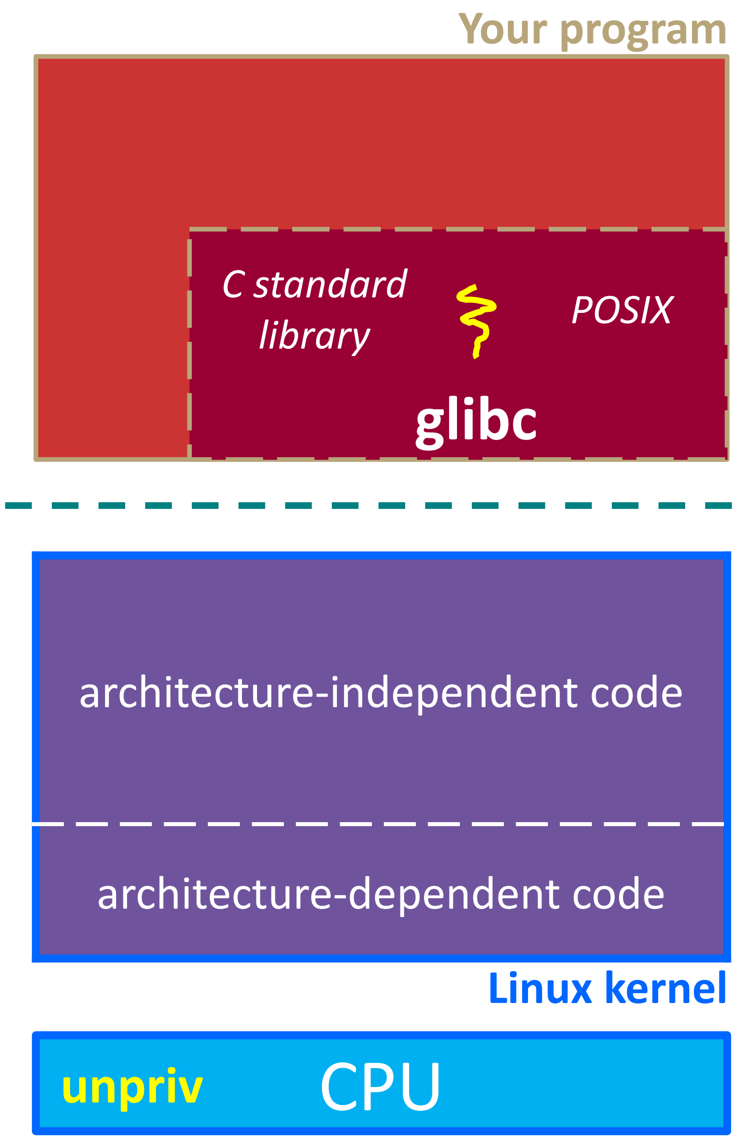


Details on x86/Linux

Process calls into a `glibc` function

- *e.g.* `fopen()`
- We'll ignore the messy details of loading/linking shared libraries

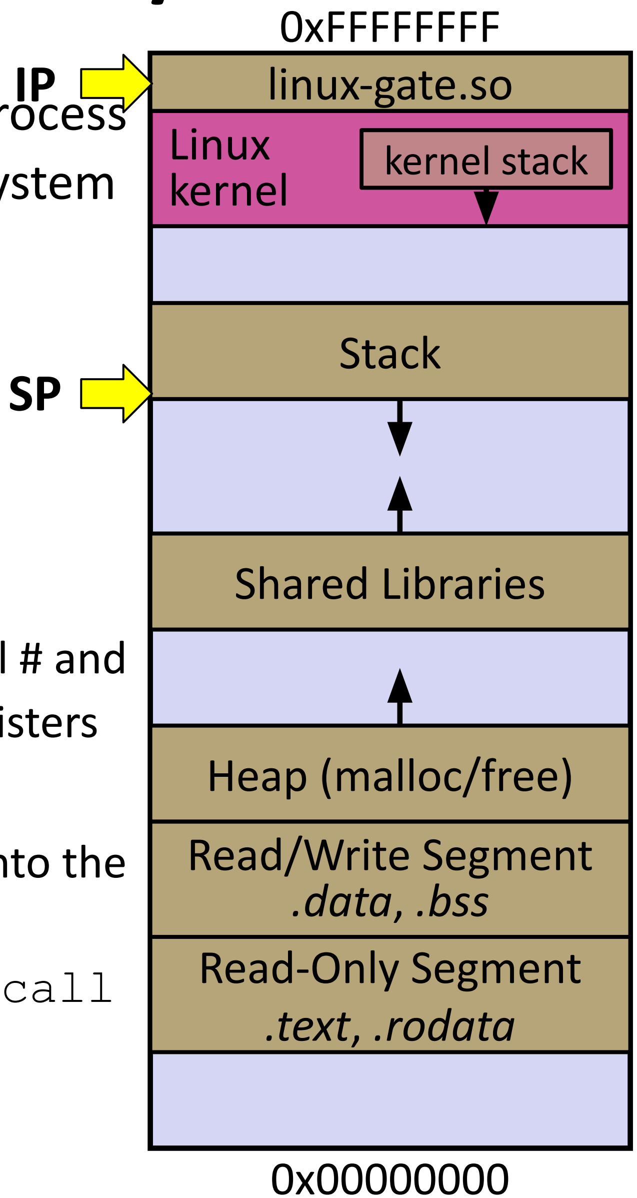


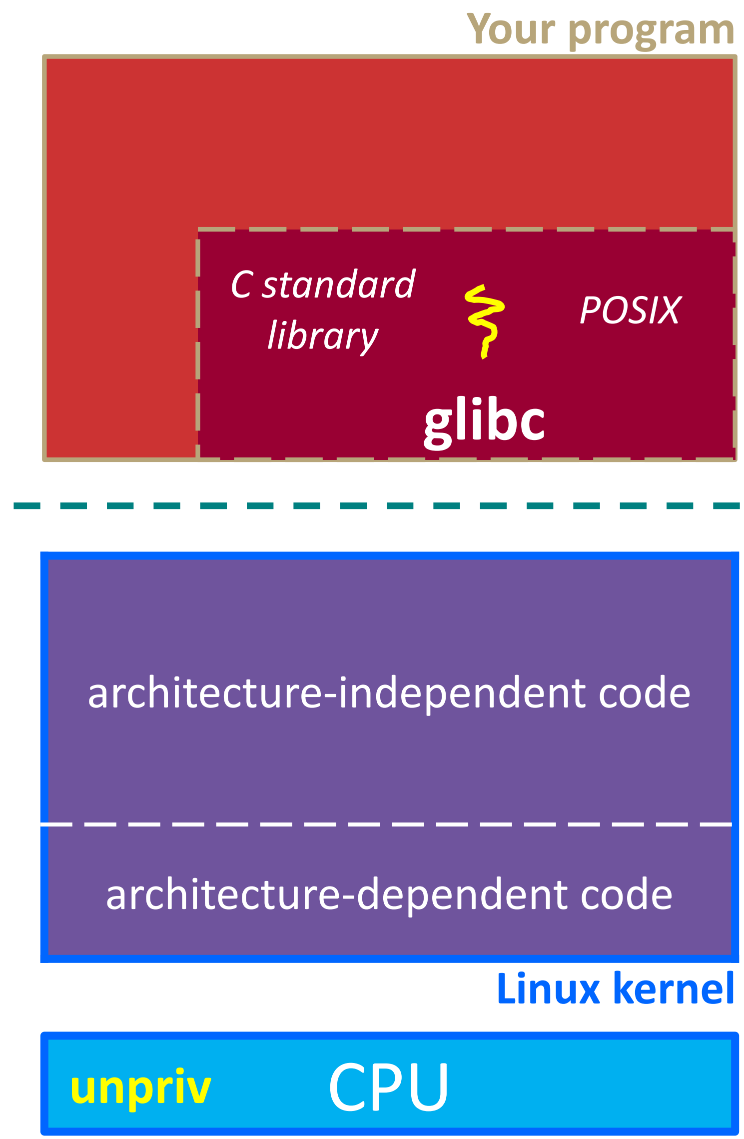


Details on x86/Linux

`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`

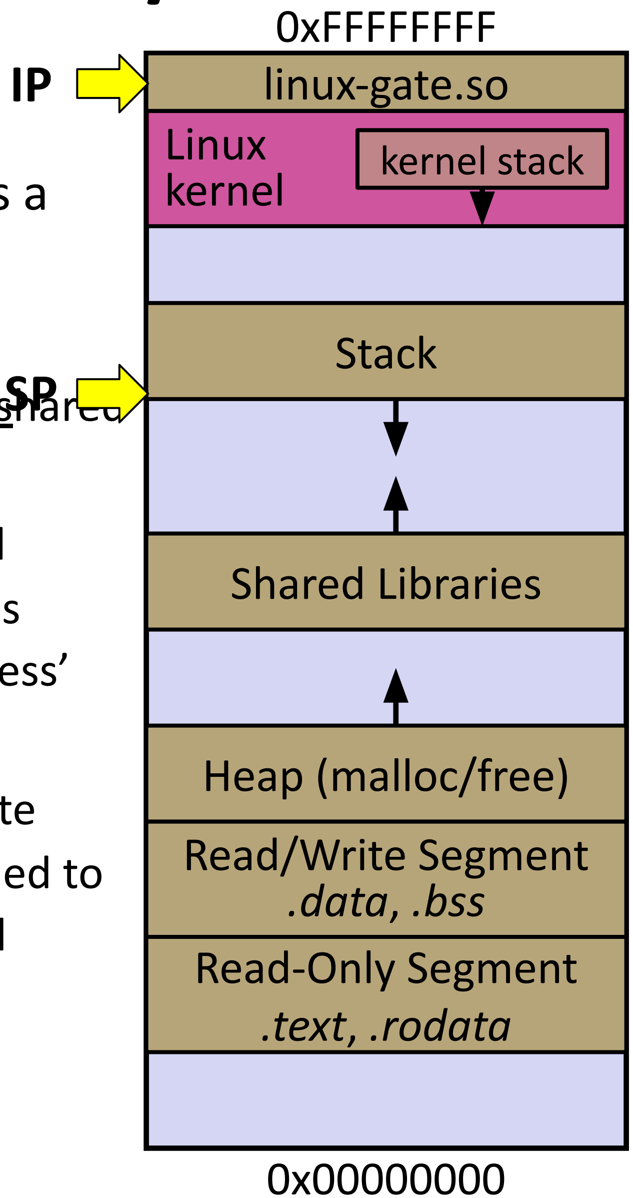


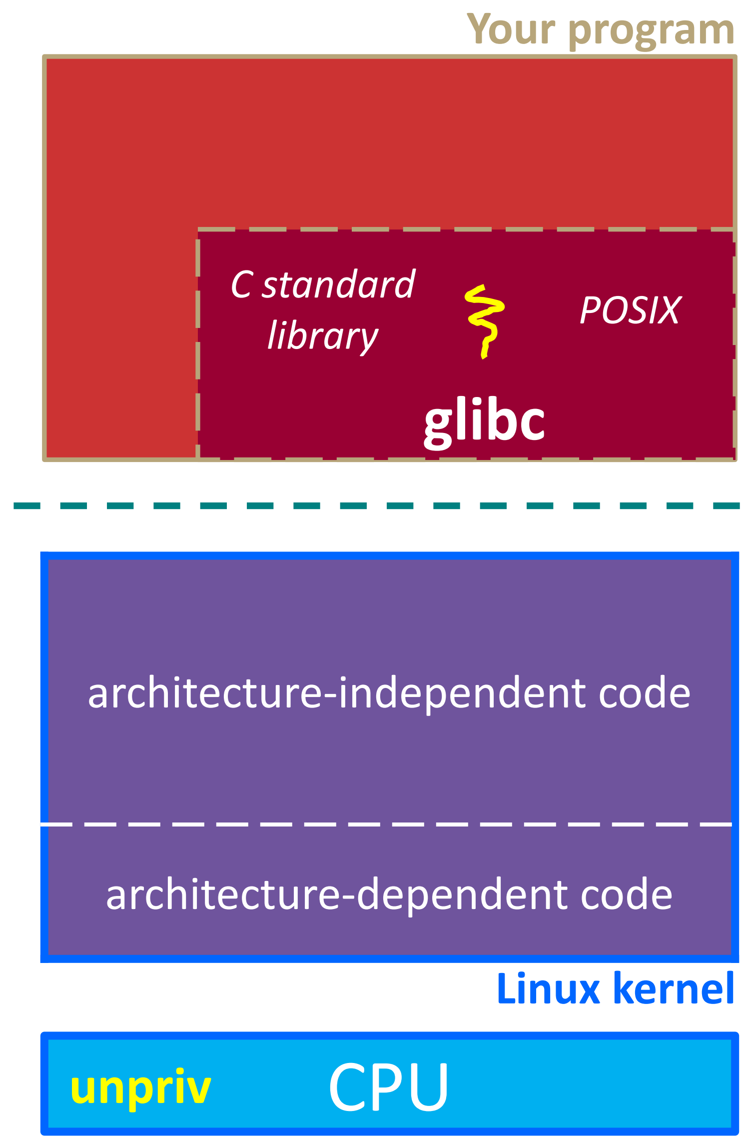


Details on x86/Linux

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

- A virtual dynamically-linked shared object
- Is a kernel-provided shared library that is plunked into a process' address space
- Provides the intricate machine code needed to trigger a system call

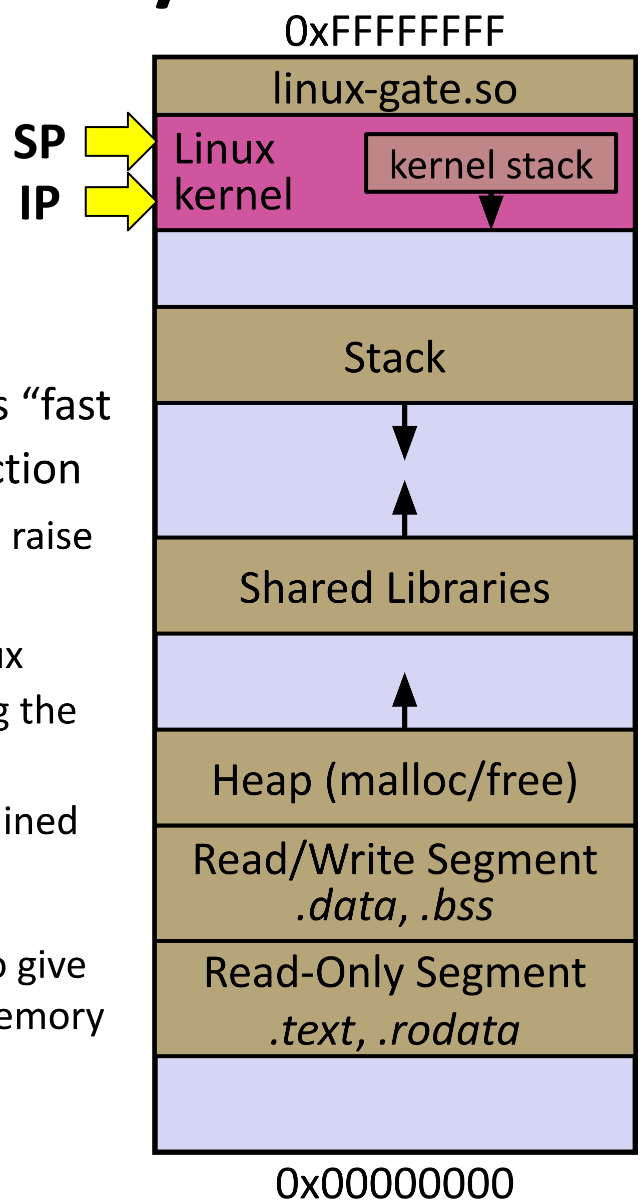


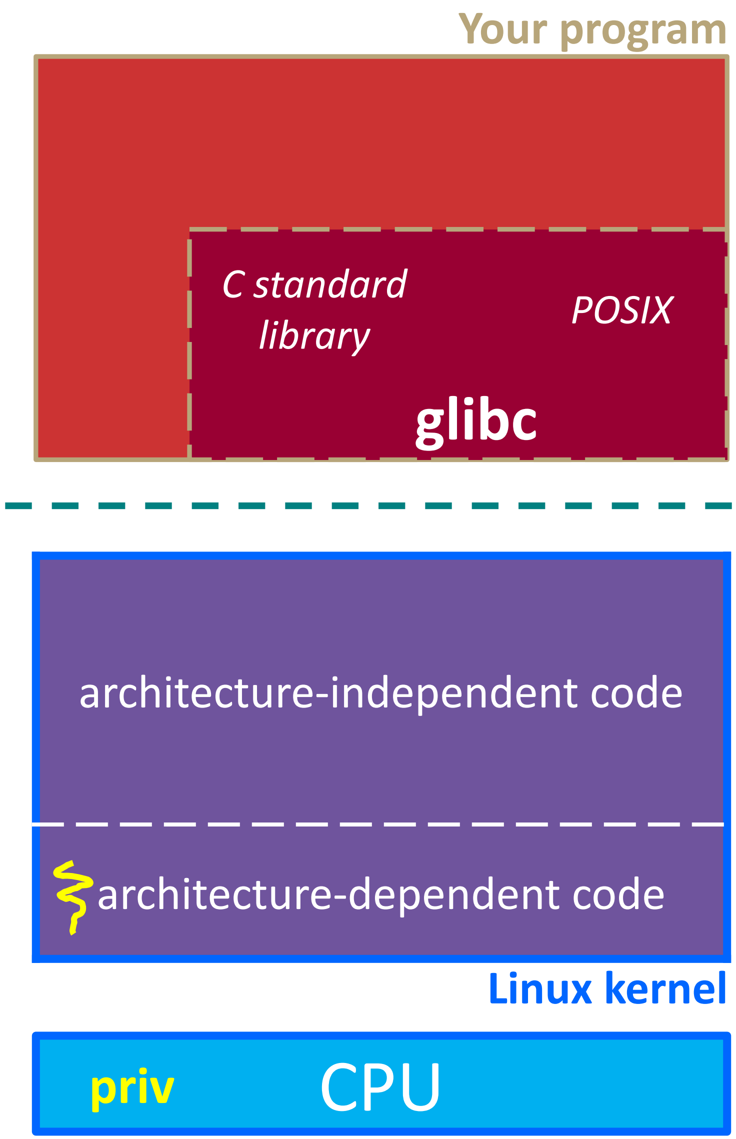


Details on x86/Linux

`linux-gate.so` eventually invokes the `SYSENTER` x86 instruction

- `SYSENTER` is x86's "fast system call" instruction
  - Causes the CPU to raise its privilege level
  - Traps into the Linux kernel by changing the SP, IP to a previously-determined location
- Changes page table to give kernel access to all memory

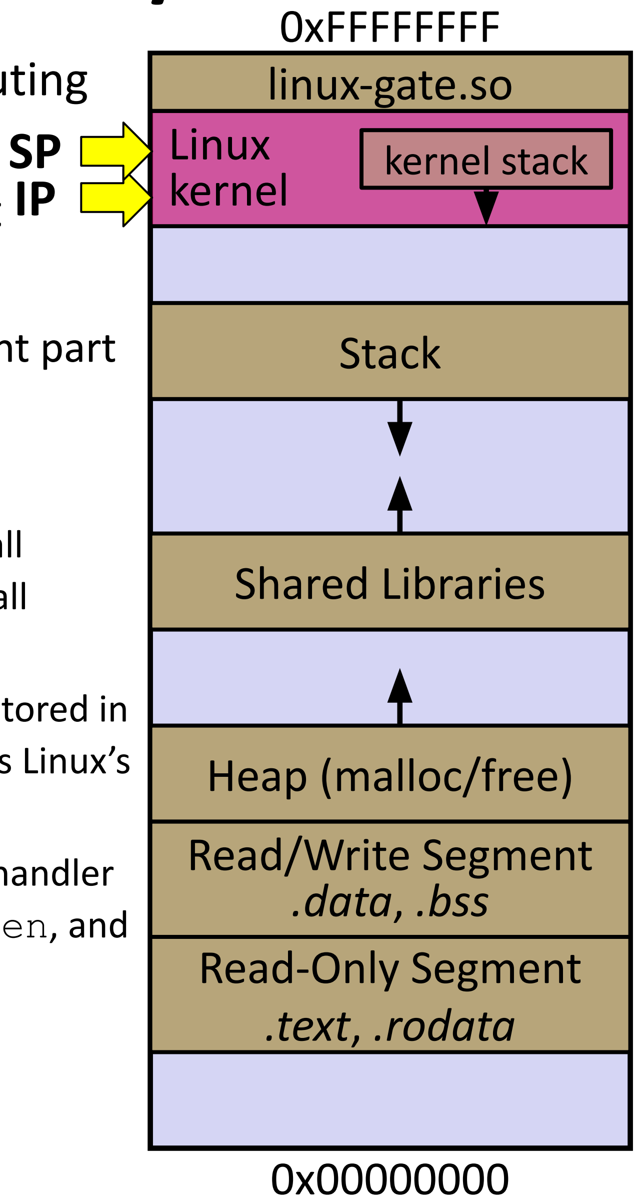


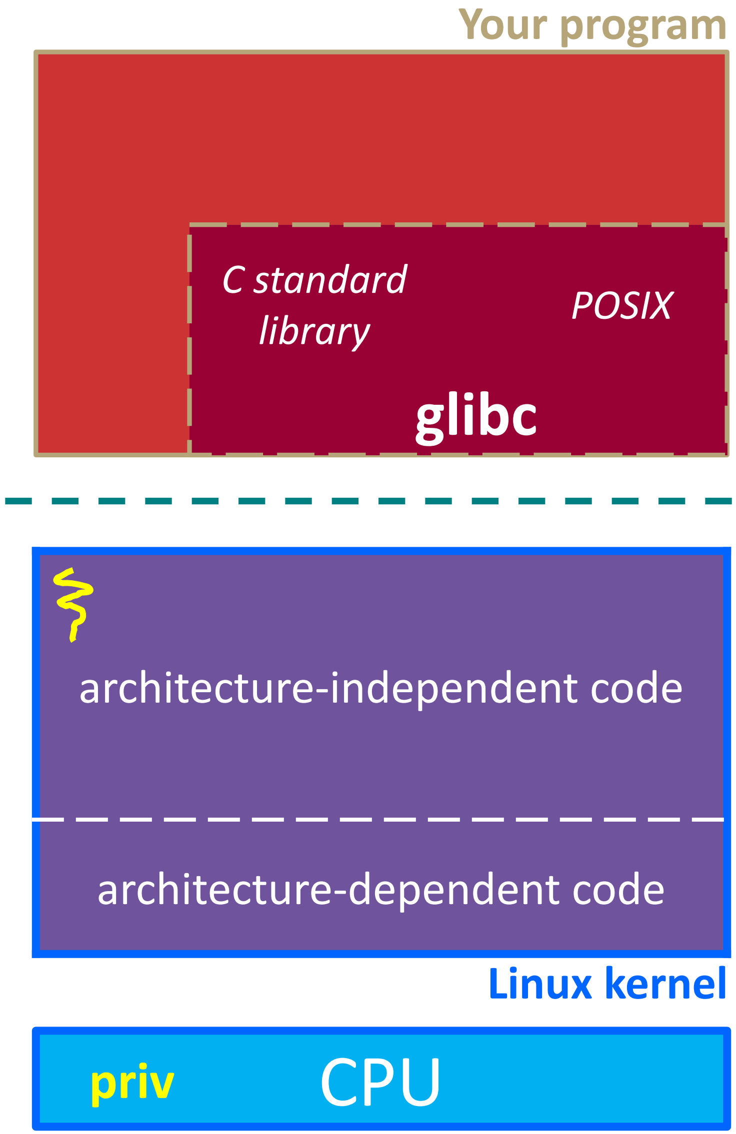


Details on x86/Linux

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



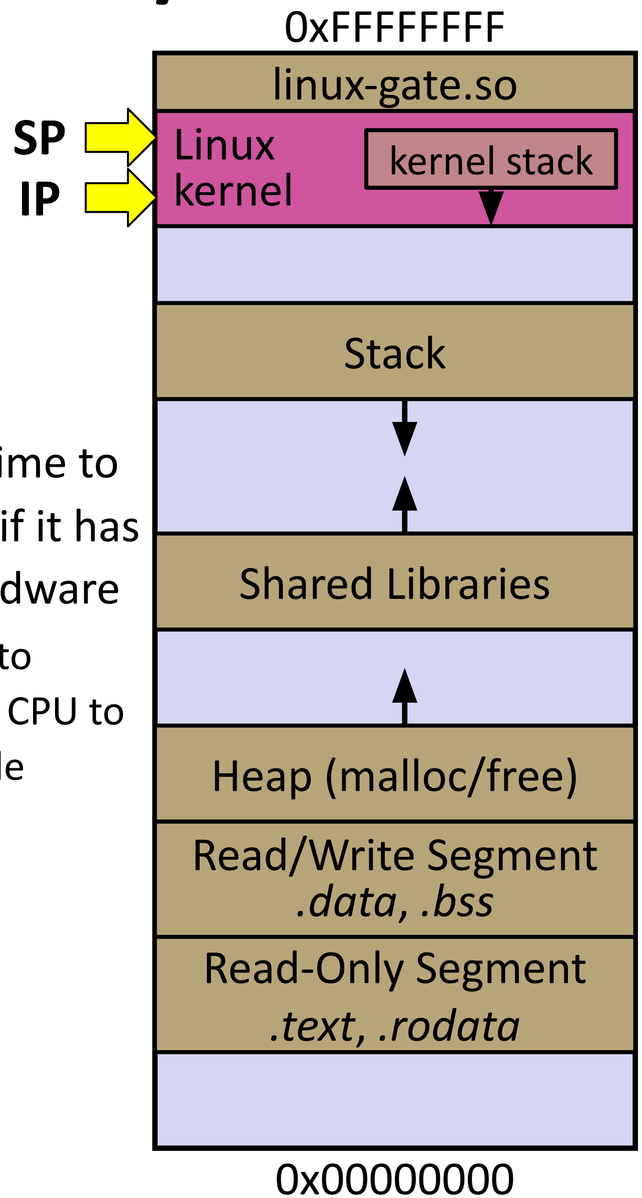


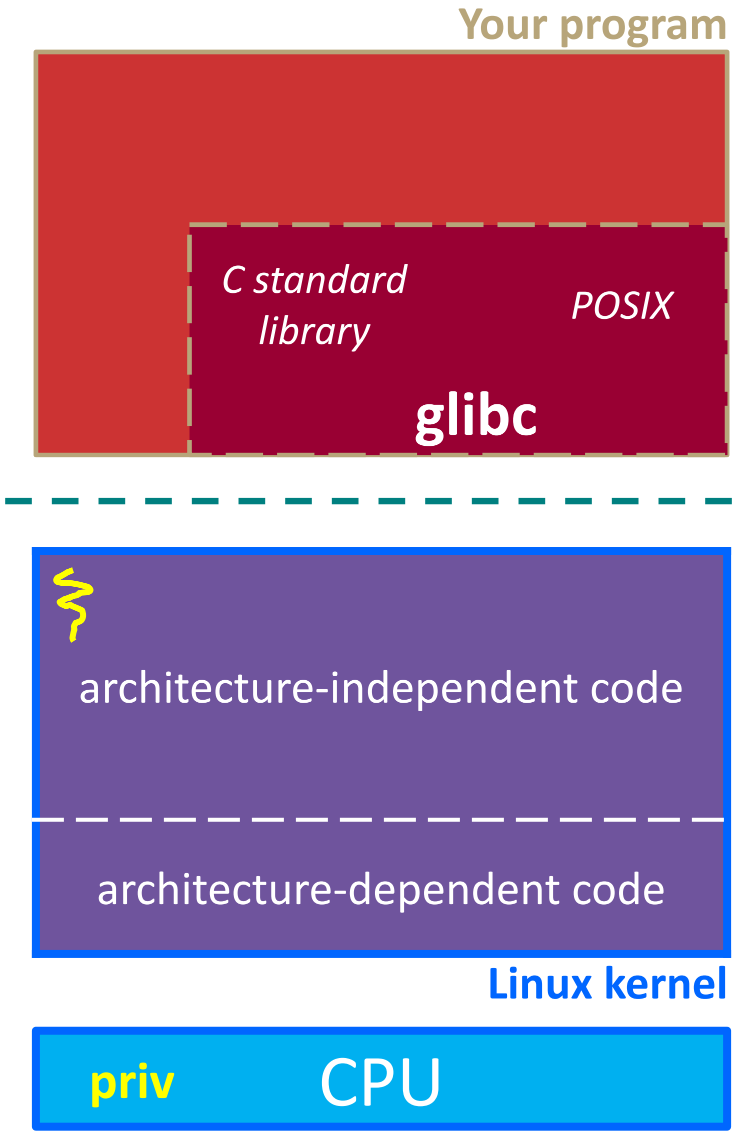
Details on x86/Linux

The system call handler executes

- What it does is system-call specific

- 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

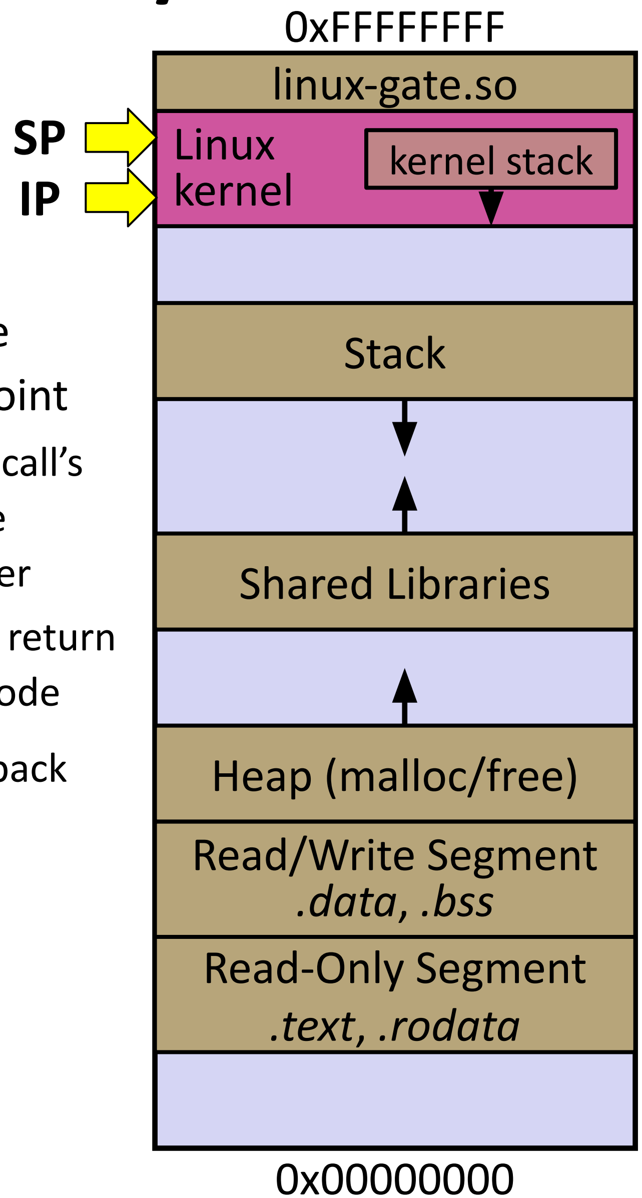


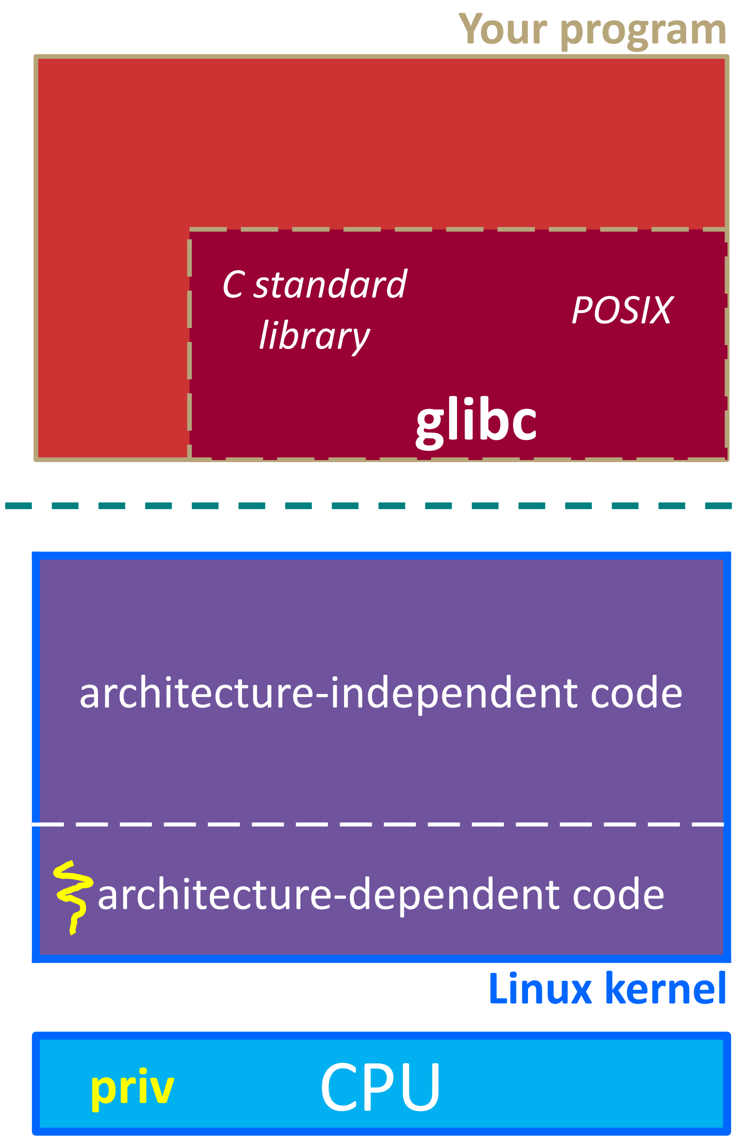


Details on x86/Linux

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
- Changes page table back

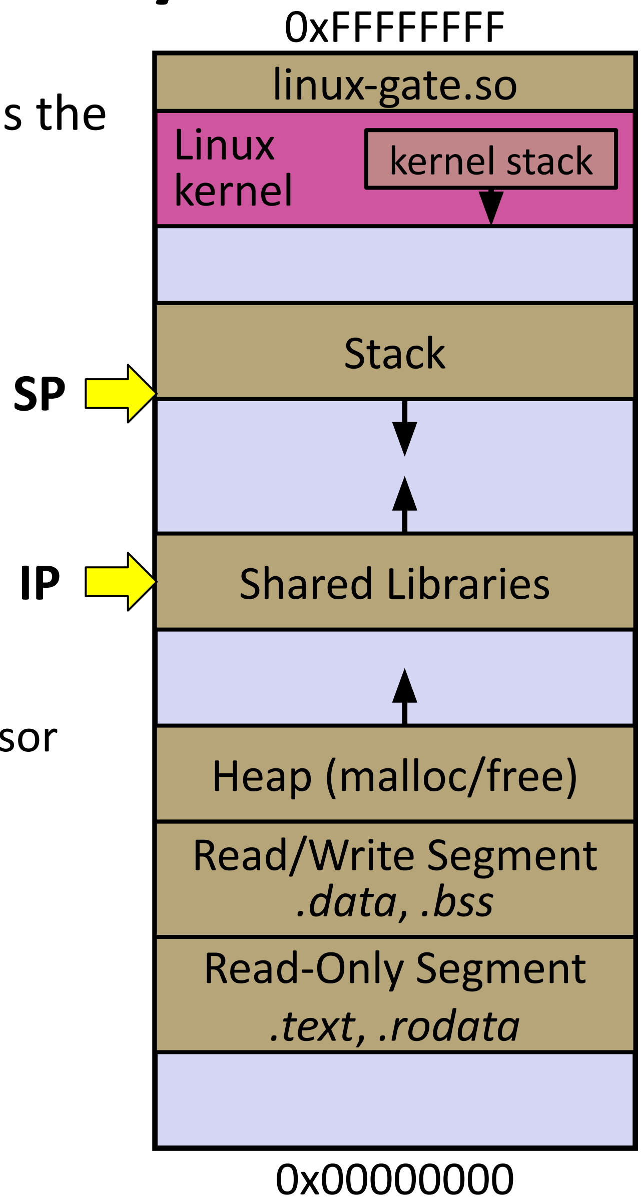


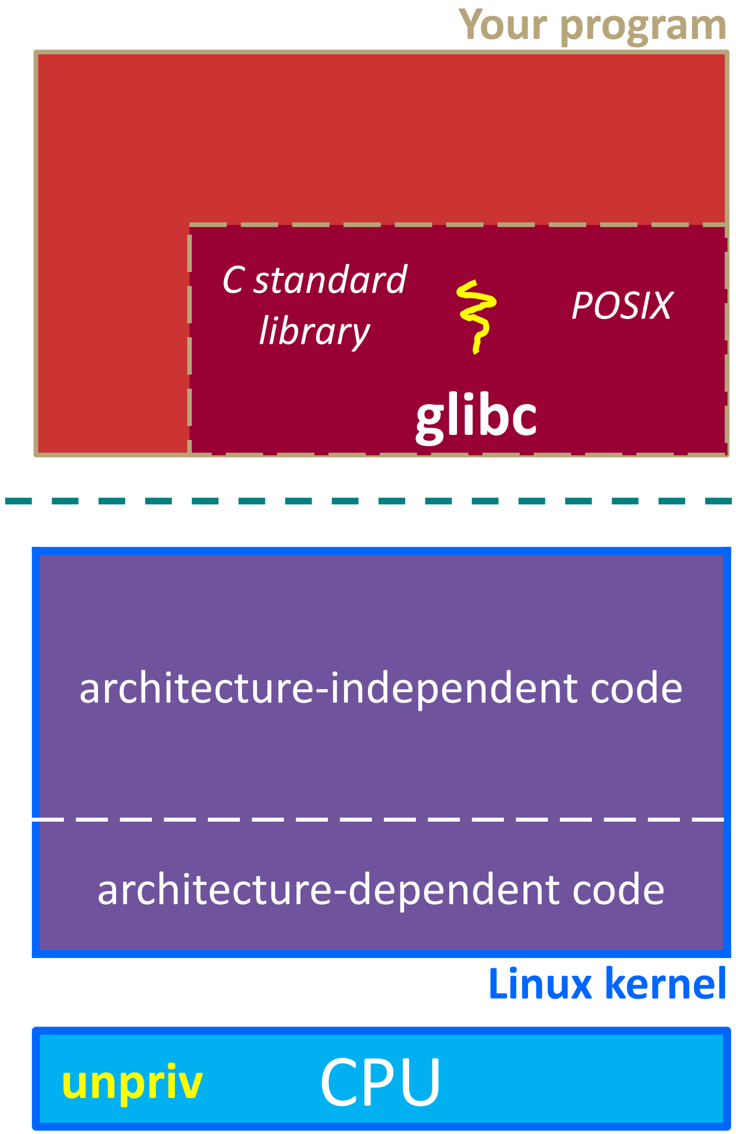


Details on x86/Linux

`SYSEXIT` transitions the processor back to user-mode code

- Restores the IP, SP to user-land values
- Sets the CPU back to unprivileged mode
- Returns the processor back to `glibc`

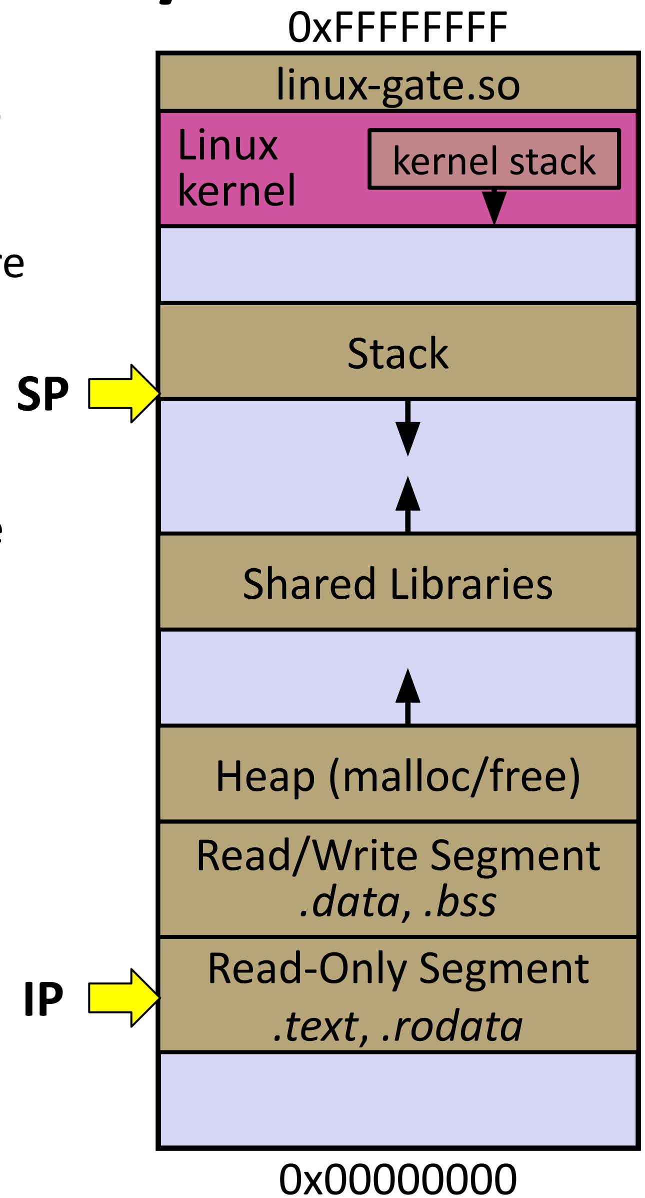


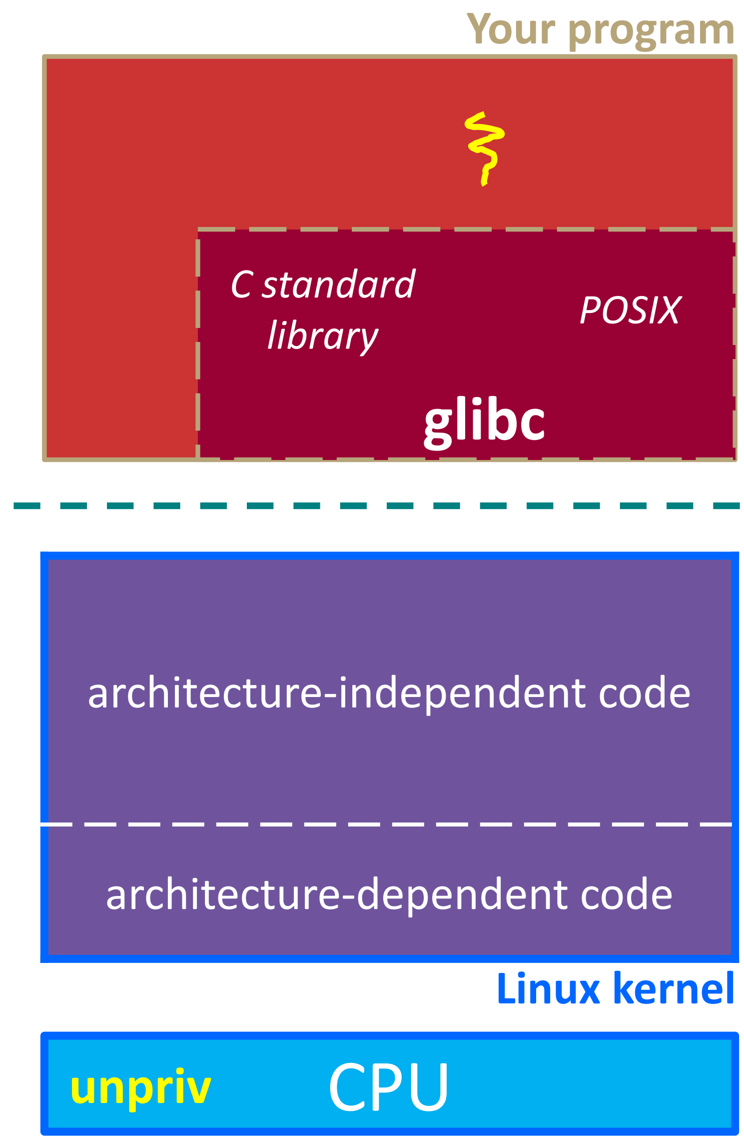


Details on x86/Linux

`glibc` continues to execute

- Might execute more system calls
- Eventually returns back to your program code





strace

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

```
bash$ strace ls 2>&1 | less
execve("/usr/bin/ls", ["ls"], [/* 41 vars */]) = 0
brk(NULL)                                       = 0x15aa000
mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) =
   0x7f03bb741000
access("/etc/ld.so.preload", R_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=126570, ...}) = 0
mmap(NULL, 126570, PROT_READ, MAP_PRIVATE, 3, 0) = 0x7f03bb722000
close(3)                                        = 0
open("/lib64/libselinux.so.1", O_RDONLY|O_CLOEXEC) = 3
read(3, "\177ELF\2\1\1\0\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0\300j\0\0\0\0\0\0"...,
   832) = 832
fstat(3, {st_mode=S_IFREG|0755, st_size=155744, ...}) = 0
mmap(NULL, 2255216, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) =
   0x7f03bb2fa000
mprotect(0x7f03bb31e000, 2093056, PROT_NONE) = 0
mmap(0x7f03bb51d000, 8192, PROT_READ|PROT_WRITE,
   MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x23000) = 0x7f03bb51d000
... etc ...
```

If You're Curious

- Download the Linux kernel source code
  - Available from http://www.kernel.org/
- `man`, section 2: Linux system calls
  - `man 2 intro`
  - `man 2 syscalls`
- `man`, section 3: `glibc/libc` library functions
  - `man 3 intro`
- *The* book: *The Linux Programming Interface* by Michael Kerrisk (keeper of the Linux man pages)

Today's Goals

- An introduction to C++
  - Some comparisons to C and shortcomings that C++ addresses
  - Give you a perspective on how to learn C++
  - Kick the tires and look at some code
  - Not trying to explain all the details, just an introduction.
- **Advice:** Read related sections in the *C++ Primer*!
  - It's hard to learn the "why is it done this way" from reference docs, and even harder to learn from random stuff on the web
  - Lectures and examples will introduce the main ideas, but aren't everything you'll ~~want~~ need to understand
  - And *free* access through UW libraries (O'Reilly books online)

C

- We had to work hard to mimic encapsulation, abstraction
  - **Encapsulation:** hiding implementation details
    - Used header file conventions and the "static" specifier to separate private functions from public functions
    - Cast structure pointers to (void*) to hide details
  - **Operational Abstraction:** associating behavior with encapsulated state
    - Function that operate on a LinkedList were not really tied to the linked list structure
    - We passed a linked list to a function, rather than invoking a method on a linked list instance

C++

- A major addition is support for classes and objects!
  - Classes
    - Public, private, and protected **methods** and **instance variables**
    - (multiple!) inheritance
  - Polymorphism
    - Static polymorphism: multiple functions or methods with the same name, but different argument types (overloading)
      - Works for all functions, not just class members
    - Dynamic (subtype) polymorphism: derived classes can override methods of parents, and methods will be dispatched correctly

C

- We had to emulate generic data structures
  - Generic linked list using `void*` payload
  - Pass function pointers to generalize different "methods" for data structures
    - Comparisons, deallocation, pickling up state, etc.

C++

- Supports templates to facilitate generic data types
  - Parametric polymorphism – same idea as Java generics, but different in details, particularly implementation
  - To declare that x is a vector of ints: `vector<int> x;`
  - To declare that x is a vector of strings: `vector<string> x;`
  - To declare that x is a vector of [vectors of floats]: `vector<vector<float>> x;`

C

- We had to be careful about namespace collisions
  - C distinguishes between external and internal linkage
    - Use `static` to prevent a name from being visible outside a source file (as close as C gets to "private")
    - Otherwise, name is global and visible everywhere
  - We used naming conventions to help avoid collisions in the global namespace
    - *e.g.* **<u>`LL`</u>**`IteratorNext` vs. **<u>`HT`</u>**`IteratorNext`, etc.

C++

- Permits a module to define its own namespace!
  - The linked list module could define an "`LL`" namespace while the hash table module could define an "`HT`" namespace
  - Both modules could define an Iterator class
    - One would be globally named `LL::Iterator`
    - The other would be globally named `HT::Iterator`
  - Entire C++ standard library is in a namespace `std` (more later…)

- Classes also allow duplicate names without collisions
  - Namespaces group and isolate names in collections of classes and other "global" things (somewhat like Java packages)

C

- C does not provide any standard data structures
  - We had to implement our own linked list and hash table
  - As a C programmer, you often reinvent the wheel… poorly
    - Maybe if you're clever you'll use somebody else's libraries
    - But C's lack of abstraction, encapsulation, and generics means you'll probably end up tinkering with them or tweak your code to use them

C++

- The C++ standard library is huge!
  - **Generic containers:** bitset, queue, list, associative array (including hash table), deque, set, stack, and vector
    - And iterators for most of these
  - **A `string` class:** hides the implementation of strings
  - **Streams:** allows you to stream data to and from objects, consoles, files, strings, and so on
  - And more…

C

- Error handling is a pain
  - Have to define error codes and return them
  - Customers have to understand error code conventions and need to constantly test return values
  - *e.g.* if `a()` calls `b()`, which calls `c()`
    - `a` depends on `b` to propagate an error in `c` back to it

C++

- Error handling is STILL a pain, but now we have exceptions
  - `try` / `throw` / `catch`
  - If used with discipline, can simplify error processing
    - But, if used carelessly, can complicate memory management
    - Consider: `a()` calls `b()`, which calls `c()`
      - If `c()` throws an exception that `b()` doesn't catch, you might not get a chance to clean up resources allocated inside `b()`
  - But much C++ code still needs to work with C & old C++ libraries that are not exception-safe, so still uses return codes, exit(), etc.
    - We won't use (and Google style guide doesn't use either)

Some Tasks Still Hurt in C++

- Memory management
  - C++ has no garbage collector
    - You have to manage memory allocation and deallocation and track ownership of memory
    - It's still possible to have leaks, double frees, and so on
  - But there are some things that help
    - "Smart pointers"
      - Classes that encapsulate pointers and track reference counts
      - Deallocate memory when the reference count goes to zero
    - C++'s destructors permit a pattern known as "Resource Allocation Is Initialization" (RAII) (terrible name but super useful idea)
      - Useful for releasing memory, locks, database transactions, and more

Some Tasks Still Hurt in C++

- C++ doesn’t guarantee type or memory safety
  - You can still:
    - Forcibly cast pointers between incompatible types
    - Walk off the end of an array and smash memory
    - Have dangling pointers
    - Conjure up a pointer to an arbitrary address of your choosing

C++ Has Many, Many Features

- Operator overloading
  - Your class can define methods for handling "+", "->", etc.
- Object constructors, destructors
  - Particularly handy for stack-allocated objects
- Reference types
  - True call-by-reference instead of always call-by-value
- Advanced Objects
  - Multiple inheritance, virtual base classes, dynamic dispatch