Lecture 04: Understanding System Calls

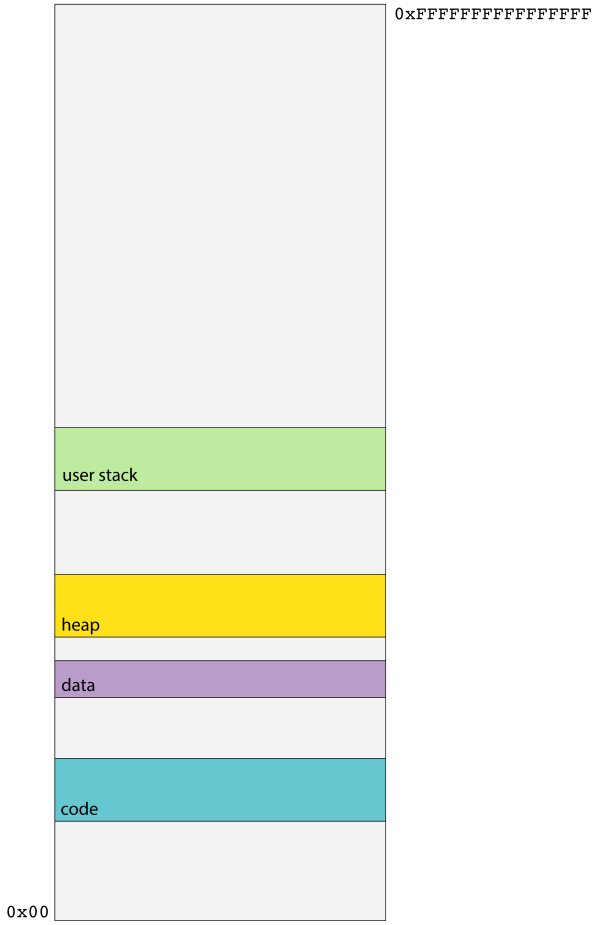
- **System calls** are functions that user programs use to interact with the OS and request some core service be executed on their behalf.
  - Examples of system calls we've already seen this quarter: `open`, `read`, `write`, `close`, `stat`, and `lstat`. We'll see many others in the coming weeks.
  - Functions like `printf`, `malloc`, `fopen`, and `opendir` are not system calls. They're C library functions that themselves rely on system calls to get their jobs done.
- Unlike traditional user functions (the ones we write ourselves, `libc` and `libstdc++` library functions), system calls need to execute in some privileged mode so they can access data structures, system information, and other OS resources intentionally hidden from user code.
- The implementation of `open`, for instance, needs to access all of the filesystem data structures for existence and permissioning. Filesystem implementation details should be hidden from the user, and permission information should be respected as private.
- The information loaded and manipulated by `open` musn’t be visible to the user functions that call `open`. Restated, privileged information shouldn't be discoverable.
- That means we need a **different call and return model** for system calls than we have for traditional functions.

Lecture 04: Understanding System Calls

- Recall that each process operates as if it owns all of main memory.
- The diagram on the right presents a 64-bit process's general memory playground that stretches from address 0 up through and including 2^64 - 1.
- CS107 and CS107-like intro-to-architecture courses present the diagram on the right, and discuss how various portions of the address space are cordoned off to manage traditional function call and return, dynamically allocated memory, global data, and machine code storage and execution.
- No process actually uses all 2^64 bytes of its address space. In fact, the vast majority of processes use a miniscule fraction of what they otherwise think they own.

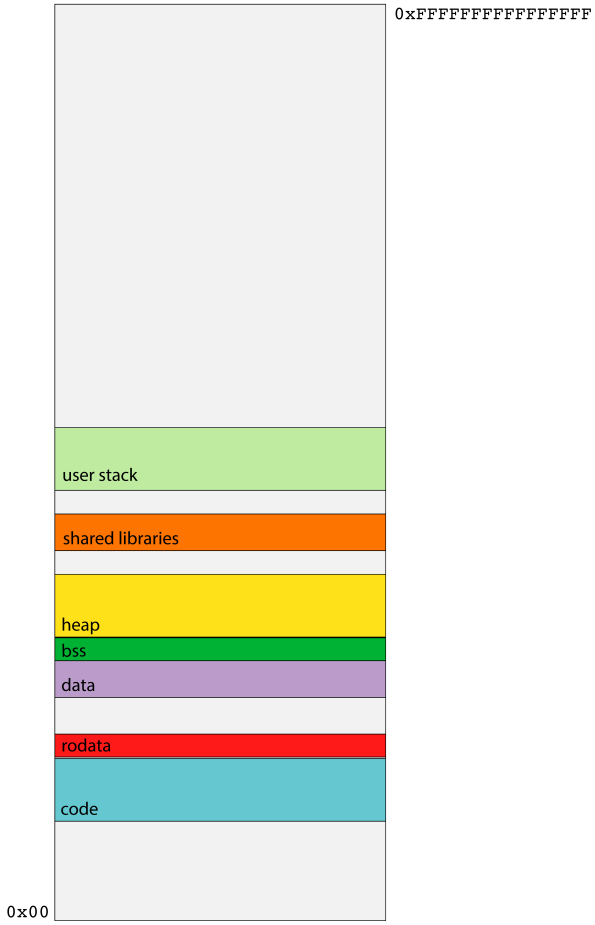
Lecture 04: Understanding System Calls

- The code segment stores all of the assembly code instructions specific to your process. The address of the currently executing instruction is stored in the %rip register, and that address is typically drawn from the range of addresses managed by the code segment.
- The data segment intentionally rests directly on top of the code segment, and it houses all of the explicitly initialized global variables that can be modified by the program.
- The heap is a software-managed segment used to support the implementation of `malloc`, `realloc`, `free`, and their C++ equivalents. It's initially very small, but grows as needed for processes requiring a good amount of dynamically allocated memory.
- The user stack segment provides the memory needed to manage user function call and return along with the scratch space needed by function parameters and local variables.



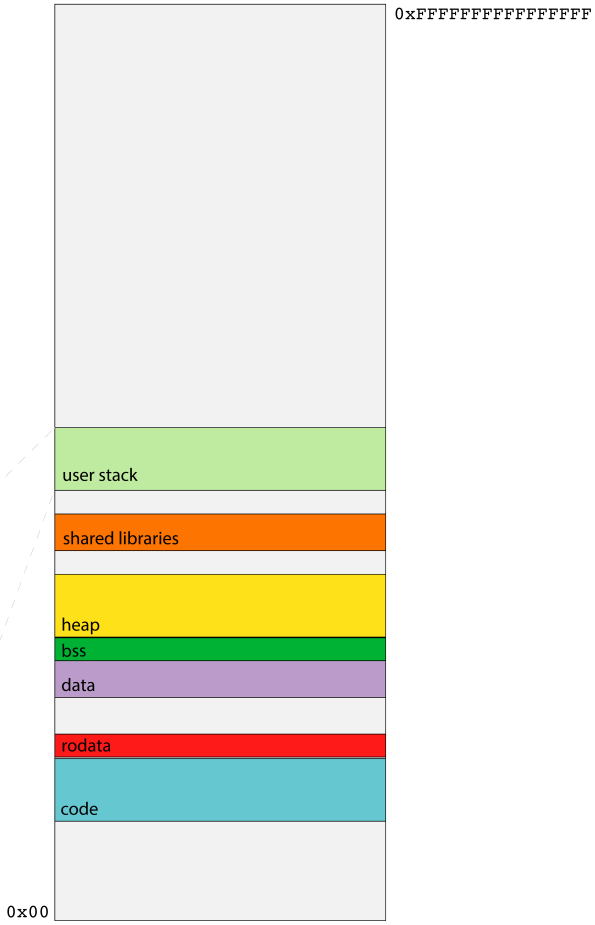
Lecture 04: Understanding System Calls

- There are other relevant segments that haven't been called out in prior classes–at least not in CS107 here.
- The **rodata** segment also stores global variables, but only those which are immutable–i.e. constants. As the runtime isn't supposed to change anything read-only, the segment housing constants can be protected so any attempts to modify it are blocked by the OS.
- The **bss** segment houses the uninitialized global variables, which are defaulted to be zero (one of the few situations where pure C provides default values).
- The shared library segment links to shared libraries like `libc` and `libstdc++` with code for routines like C's `printf`, C's `malloc`, or C++'s `getline`. Shared libraries get their own segment so all processes can trampoline through some glue code–that is, the minimum amount of code necessary–to jump into the one copy of the library code that exists for all processes.



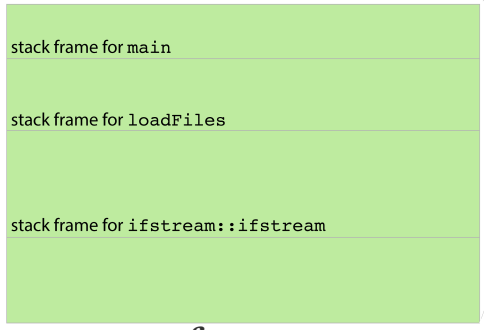
Lecture 04: Understanding System Calls

- The user stack maintains a collection of stack frames for the trail of currently executing user functions.
- 64-bit process runtimes rely on `%rsp` to track the address boundary between the in-use portion of the user stack and the portion that's on deck to be used should the currently executing function invoke a subroutine.
- The x86-64 runtime relies on `callq` and `retq` instructions for user function call and return.
- The first six parameters are passed through `%rdi`, `%rsi`, `%rdx`, `%rcx`, `%r8`, and `%r9`. The stack frame is used as general storage for partial results that need to be stored somewhere other than a register (e.g. a seventh incoming parameter)



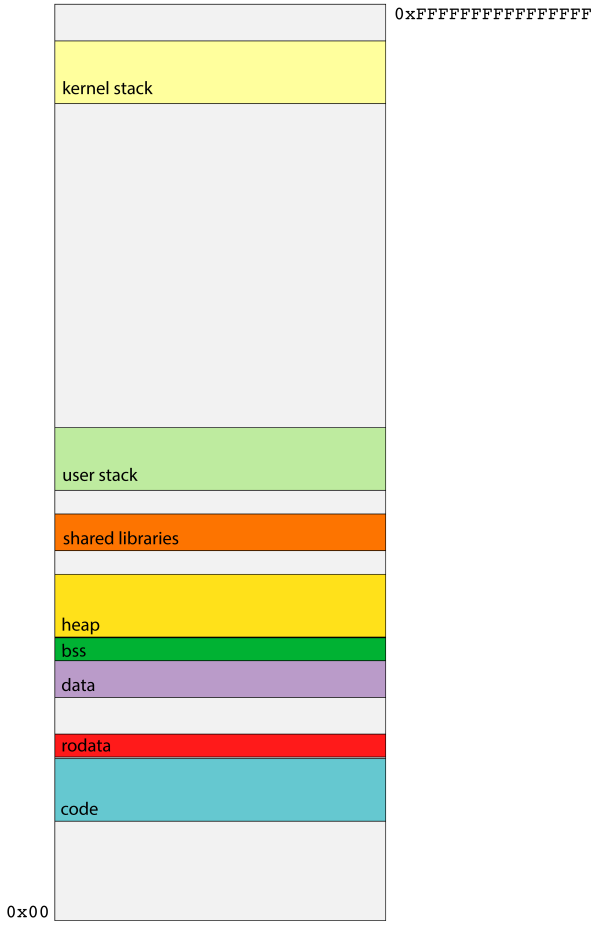
Lecture 04: Understanding System Calls

- The user function call and return protocol, however, does little to encapsulate and privatize the memory used by a function.
- Consider, for instance, the execution of `loadFiles` as per the diagram below. Because `loadFiles`'s stack frame is directly below that of its caller, it can use pointer arithmetic to advance beyond its frame and examine–or even update–the stack frame above it.
- After `loadFiles` returns, `main` could use pointer arithmetic to descend into the ghost of `loadFiles`'s stack frame and access data `loadFiles` never intended to expose.
- Functions are supposed to be modular, but the function call and return protocol's support for modularity and privacy is pretty soft.



Lecture 04: Understanding System Calls

- System calls like `open` and `stat` need access to OS implementation detail that should not be exposed or otherwise accessible to the user program.
- That means the activation records for system calls need to be stored in a region of memory that users can't touch, and the system call implementations need to be executed in a privileged, superuser mode so that it has access to information and resources that traditional functions shouldn't have.
- The upper half of a process's address space is **kernel space**, and none of it is visible to traditional user code.
- Housed within kernel space is a kernel stack segment, itself used to organize stack frames for system calls.
- `callq` is used for user function call, but `callq` would dereference a function pointer we're **not permitted** to dereference, since it resides in kernel space.
- We need a different call and return model for system calls—one that doesn't rely on `callq`.



Lecture 04: Understanding System Calls

- The relevant opcode is placed in `%rax`. Each system call has its own opcode (e.g. 0 for `read`, 1 for `write`, 2 for `open`, 3 for `close`, 4 for `stat`, and so forth).
- The system call arguments–there are at most 6–are evaluated and placed in `%rdi`, `%rsi`, `%rdx`, `%r10`, `%r8`, and `%r9`. Note the fourth parameter is `%r10`, not `%rcx`.
- The system issues a software interrupt (otherwise known as a **trap**) by executing `syscall`, which prompts an interrupt **handler** to execute in superuser mode.
- The interrupt handler builds a frame in the kernel stack, executes the relevant code, places any return value in `%rax`, and then executes `iretq` to return from the interrupt handler, revert from superuser mode, and execute the instruction following the `syscall`.
- If `%rax` is negative, `errno` is set to abs(`%rax`) and `%rax` is updated to contain a -1. If `%rax` is nonnegative, it's left as is. The value in `%rax` is then extracted by the caller as any return value would be.