Operating Systems!

OS Interaction (Part 1)

How to Get Stuff Done Using the OS, Processes, Threads, Etc.

Prof. Travis Peters

Montana State University

CS 460 - Operating Systems

Fall 2020

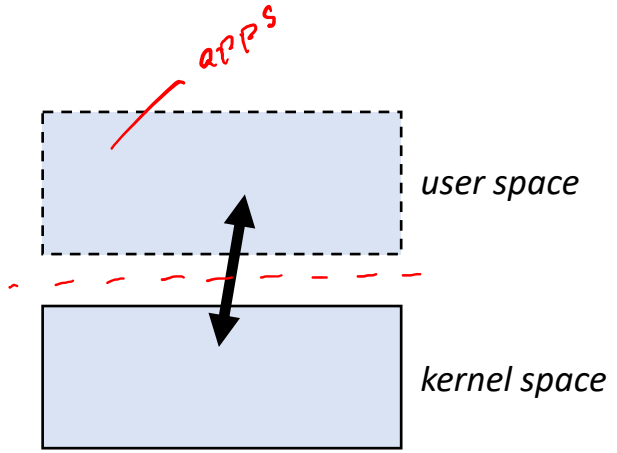
https://www.cs.montana.edu/cs460

Today

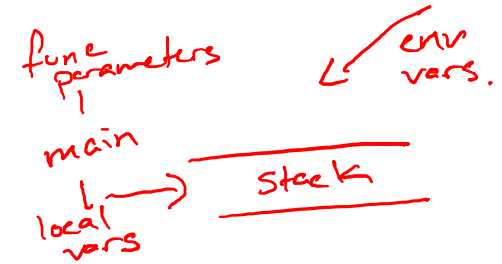
- Announcements
  - *Heads up.... PA1 has been released!*
    Sunday [09/20/2020] @ 11:59 PM (MST)

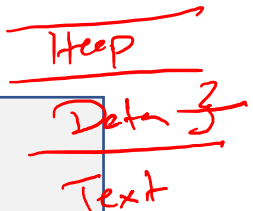
    NO D2L! *Code pushed to GitHub == submitted!*

- Learning Objectives
  - Understand the big ideas behind the "OS API"
    - **user vs. kernel, modes, syscalls, libraries**
  - Understand the big ideas behind process control
    - *[theory]* **control info, creation, termination, states, etc.**
    - *[reality]* **fork, exec, getpid, waitpid, etc.**



Warm-up *(C Programming)*



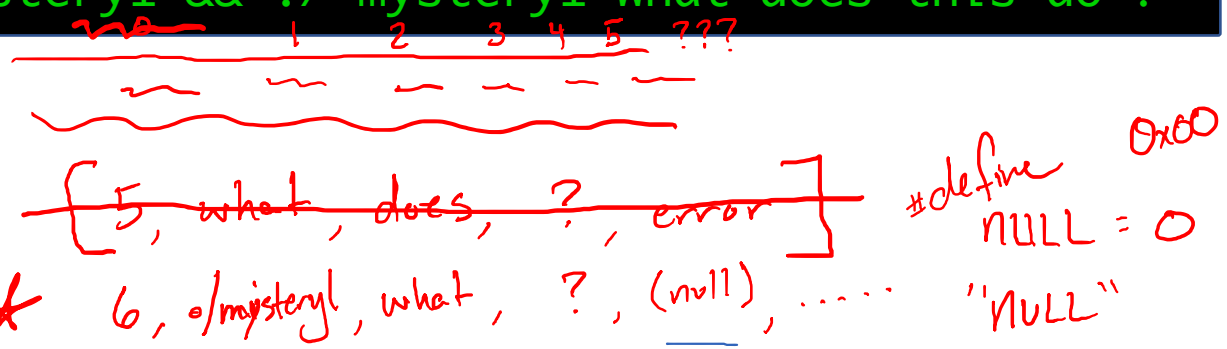



```c
#include <stdio.h>

int main(int argc, char *argv[])
{
    printf("%d, %s, %s, %s, %s\n", argc, argv[0], argv[1], argv[5], argv[6]);
    return 0;
}
```

```
# Run prog. What does this program do?
$ gcc mystery1.c -o mystery1 && ./ mystery1 what does this do ?
```

args — command line args



Re-Orienting

An OS is the thing between the machine and the "user" that makes it easier for the user to get some work done.

Some Key Ideas:

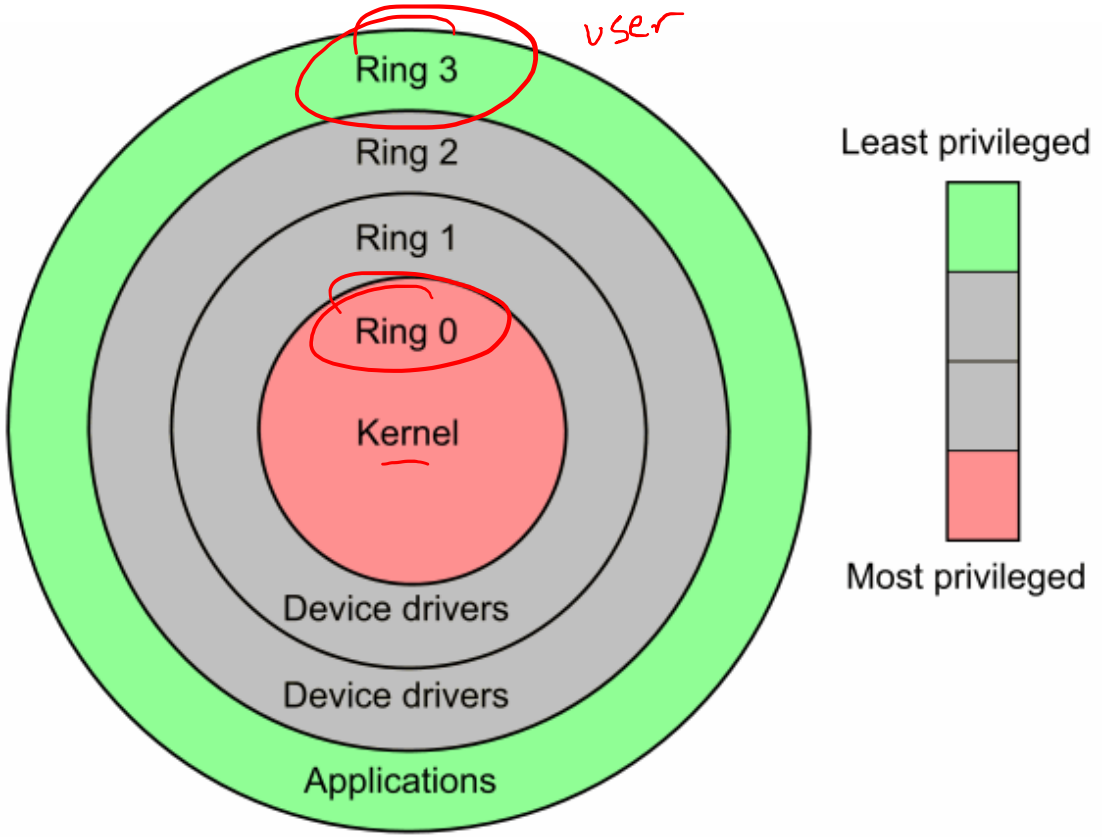
- **multiprogramming:** the same processor can switch back and forth between different "processes" --- each process *thinks* it has the system to itself!
- ***interrupt:*** mechanism to stop a process and execute an "interrupt handler"
- ***switching:*** the actual mechanics of taking one process off of the processor and putting another one on.
- **modes:** an OS is typically divided into two *(or more)* modes of operation
  - user *("unprivileged")*
  - kernel *("privileged" / "supervisor" / "monitor")*

Modes (Kernel Mode vs. User Mode)

An OS is typically divided into two *(or more)* modes of operation

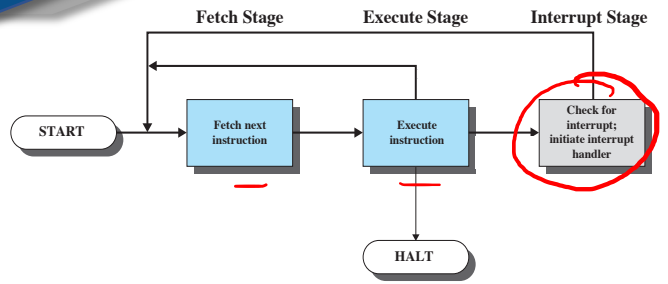
- user *("unprivileged")*
  - restricted instruction set
  - restricted access to memory
- kernel *("privileged" / "supervisor" / "monitor")*
  - unrestricted instruction set
  - full access to memory and I/O

CPU must check whether instruction is permissible before executing it



Credit: https://blog.codinghorror.com/understanding-user-and-kernel-mode/

(Revisiting) Interrupts & Traps



Interrupt

- A hardware event that sets flag in the CPU; e.g.,
  - Timer event
  - I/O complete

} types

Trap

- A software interrupt that is run under predefined conditions; e.g.,
  - Exceptions
  - System Calls

} types

A Simple Flow

1. Set flag in CPU
   (A bit; also some info about the type of interrupt)
2. CPU cycle detects flag
   (Recall fetch-decode-execute cycle w/ "check")
3. CPU pauses typical execution sequence
   a) stash current processor state
      (By pushing some things onto the stack)
   b) switch to kernel mode
      *(Flip the **"mode" bit**)*
   c) start executing at predefined location in memory
      *(A "handler" - look in the **Interrupt Vector**; which handler to run depends on the interrupt type)*

While there are subtle distinctions between definitions for interrupts, traps, and exceptions, they really are all just types of *interrupts*... They all *interrupt* the processor and cause it to take some other action.

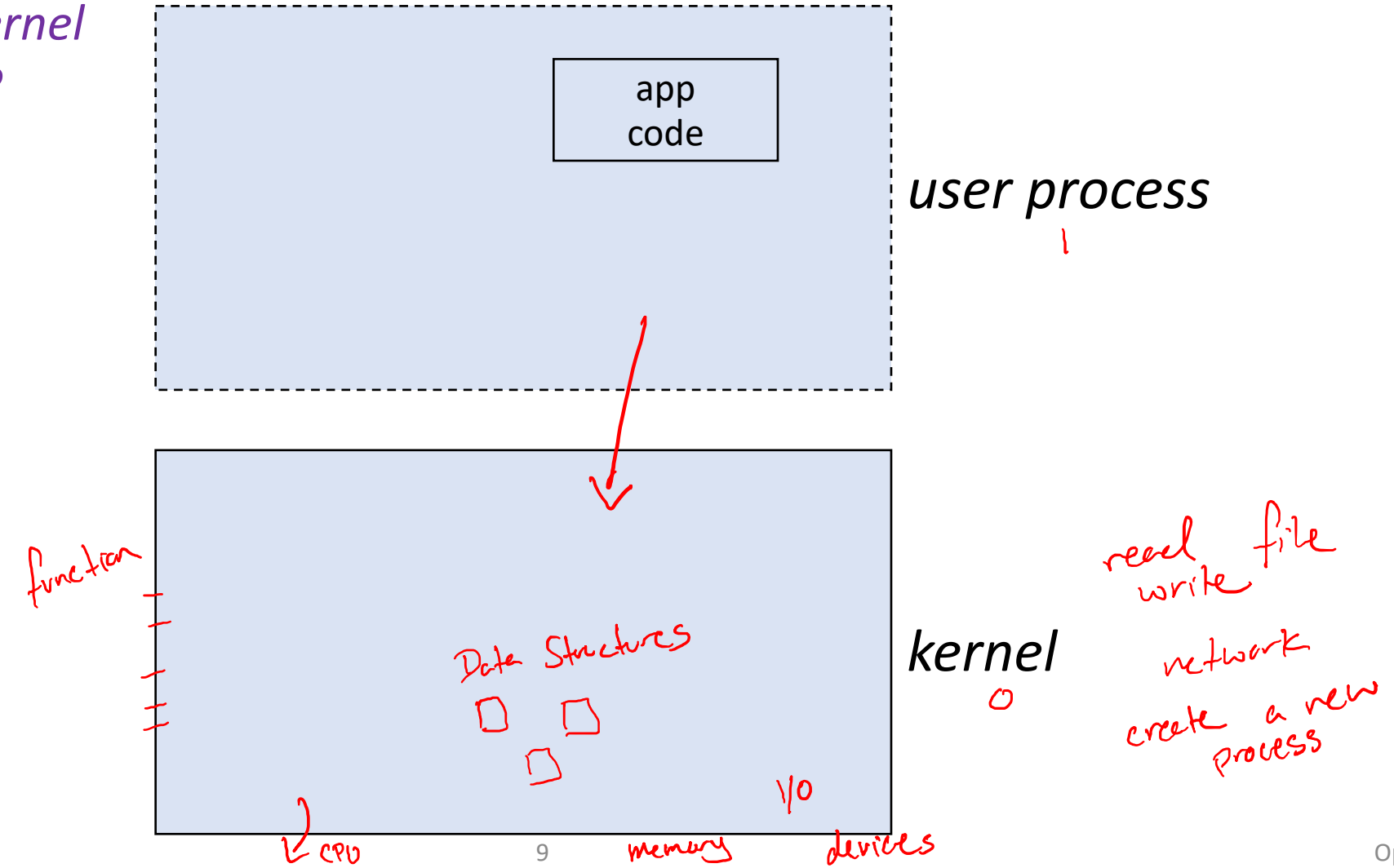
(Go read this SO post: https://stackoverflow.com/a/3149217)

Interacting With Your OS

System Calls ("Syscalls")

An application program explicitly requests an OS service by making a ***system call ("syscall").***

HOW do I use the kernel and its services?



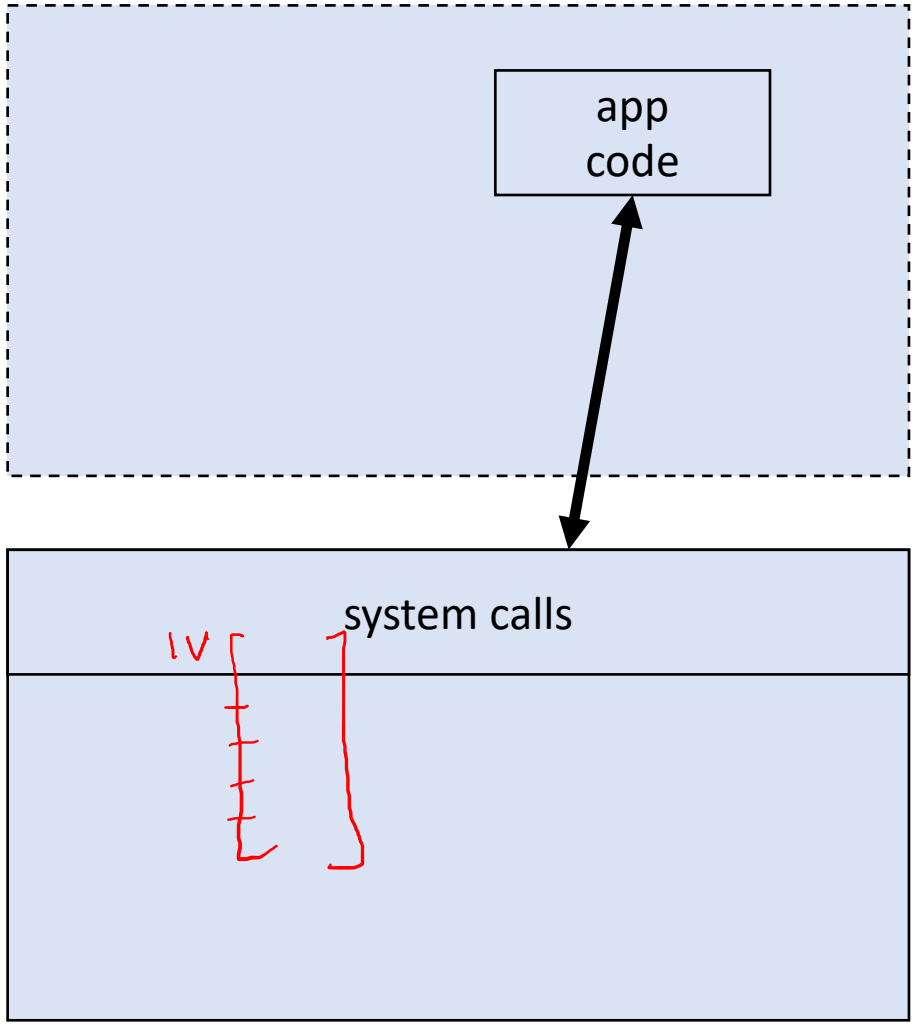
System Calls ("Syscalls")

An application program explicitly requests an OS service by making a ***system call ("syscall").***

A ***syscall*** *is invoked with a* ***software interrupt***

after

putting values ***into registers*** *or on the stack*



user process

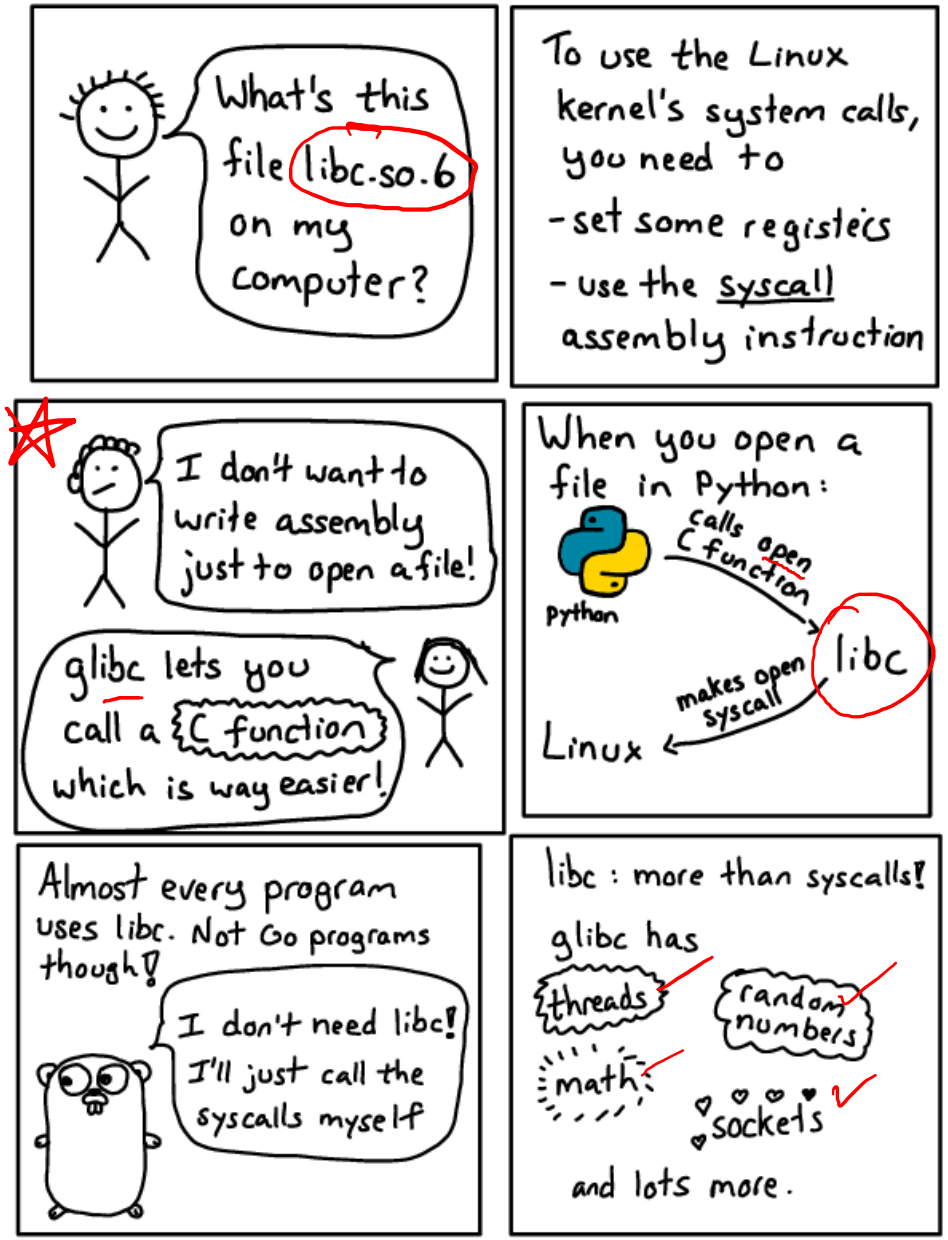
kernel

issuing instructions? interrupts?

x86 Instructions:
https://www.felixcloutier.com/x86/

♥ libc ♥

Julia Evans
@b0rk



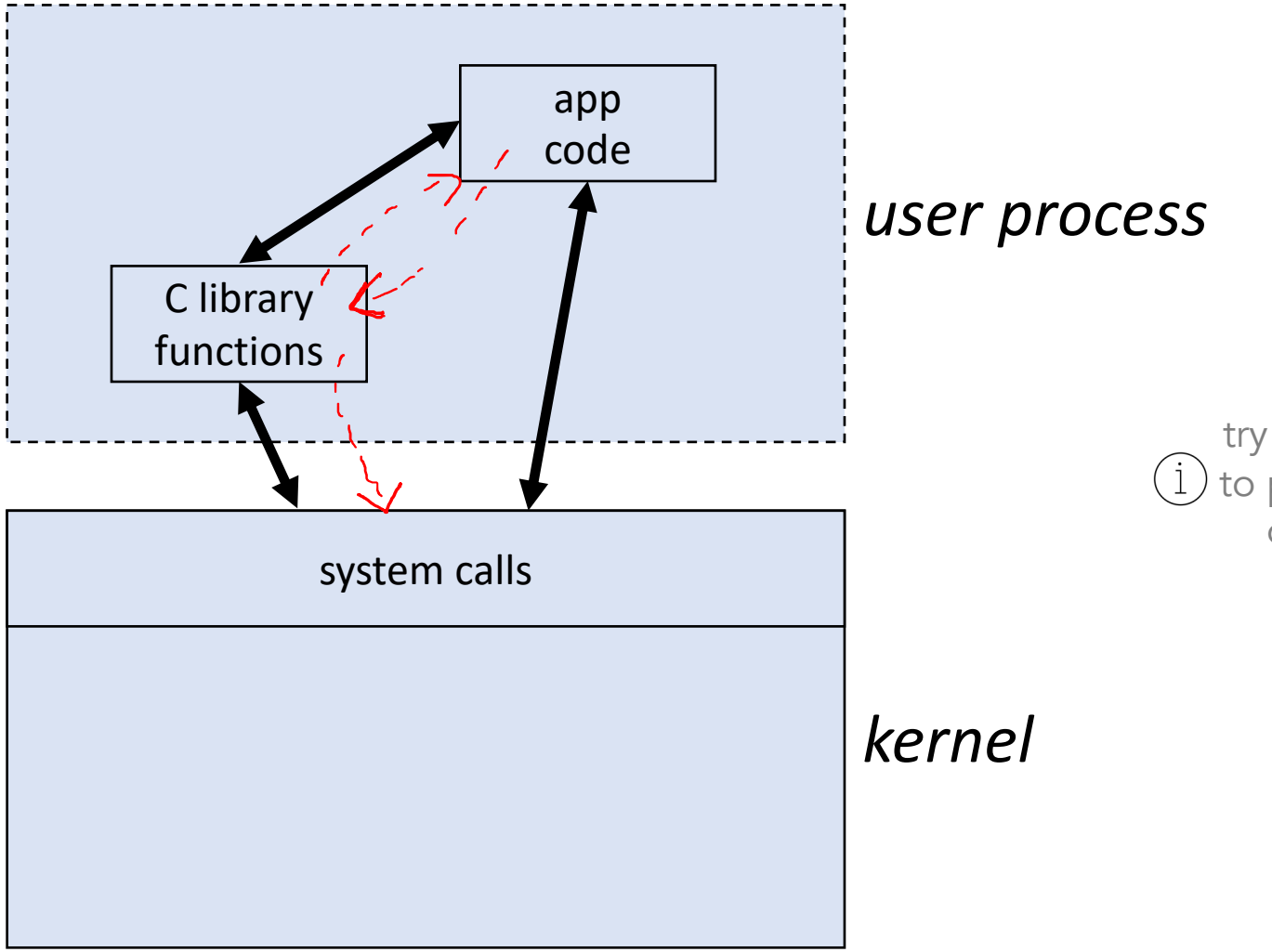
System Calls ("Syscalls")

An application program explicitly requests an OS service by making a ***system call ("syscall").***

*A **syscall** is invoked with a **software interrupt***

after

*putting values **into registers** or on the stack*



try week03/test.c to play with ideas of lib vs. OS

Assembler Fragment

A syscall is often realized in assembly, which adheres to the processor's conventions (this fragment is actually from Yalnix!)

```
// basic macro: move the 4 args and the 'code' into registers
// and throw a software trap (picked up in support/exception.c).
//
// ideally, we should have a different version for each argument count.

#define YSYSCALL(code,A,B,C,D) \
  int rc; \
  __asm__ __volatile__ ( "int $04\n\t"          \
                         : "=eax" (rc)          \
                         : "eax"(A), "b"(B), "c"(C),"d"(D), "D" (code) ); \
  return rc
```