What is lab 1 all about?

Three major tasks:

- Initialize KOS
- Implement the read()/write() system calls
- Set up argc and argv for the user program

Your code will mostly be waiting for other things to happen!

- OSes are **asynchronous** and **event-driven**

What you're turning in:

- kos.c (.h)
- exception.c
- console_buf.c (.h)
- scheduler.c (.h)
- syscall.c (.h)

The simulator

- The simulator (or any PC) is a mishmash of different hardware devices that *do things on their own time*
  - The simulator calls your code 1) at startup, 2) upon trap/exception, 3) upon interrupt
  - You give control back to the simulator with run_user_code()
- KOS is based on ULTRIX (a type of UNIX)
- Shipped with the DECstation (1993)
  - Based on the 32-bit MIPS architecture
- The simulator we provide you will call KOS() (among other things)
- Your interface with the simulator is in ***simulator.h***

simulator.h

- ***Do not modify this file!***
- int examine_registers(int buf[40])
  - Take a snapshot of the registers and put it in buf
- Also contains some useful stuff you need
  - NumTotalRegs (it’s 40 btw)
  - extern char *main_memory
  - Also system call numbers, exception/interrupt types, etc.

kos.c

In a real OS, KOS() would be *bootstrapped* in the hardware

The basic steps for loading a user program in KOS is:

- Load the program's executable file into memory
- Zero all the registers except for StackReg, which should be set to at least 12 bytes from the highest memory address for the program
- Call run_user_code() with these registers

Your OS runs in **kernel space** and the user program will run in **user space**

- Can only access some ISA instructions, memory, ...
- Controlled by a bit (or multiple) somewhere on the CPU!

syscall.c

- This is where you implement the read() and write() system calls
- buf[5] ⇛ file descriptor
- buf[6] ⇛ pointer to read/write buffer
- buf[7] ⇛ size

Trap: a process asks the OS to do something

- **Exception**: a *TRAP* generated by the CPU (div by 0, system call, ...)
- The terminology is very fuzzy here...

Interrupt: a device asks the OS to do something

"Our OS isn't running in parallel! Why do we need threads?"

scheduler.c

PCB struct contains an array of ints representing your register values

How to initialize argc and argv?

- MoveArgsToStack(int *registers, char *argv[], int mem_base)
- InitCRuntime(int *user_args, *registers, char *argv[], int mem_base)
    - (These are defined in simulator.h)
    - mem_base is the lowest address in main_memory[] that the process will use (0 for now...)
    - int *user_args is generated by MoveArgsToStack() and deallocated by InitCRuntime()

All about PCBs and the ready queue

- You need to maintain a *doubly-linked list* of PCBs, where each PCB represents a process
  - *In this lab*, the ready queue will be max 1 element long
- What's in a PCB?
  - Just a snapshot of the register set
  - This will change...
- Append a PCB to the ready queue when it's ready to run
- Pop a PCB off the ready queue before calling run_user_code()
- ***And yes! You can finally use malloc()!!!***

exception.c

When you encounter a read/write *exception*:

- kt_fork(), invoking your code that handles read/write
- kt_joinall() # BLOCK
- Execute the waiting process

In later labs, you will add to exception.c to handle other types of interrupts!