Last Time
- Stack Frames
- Syscalls (kind of)

Today
- Syscalls (briefly)
- OS view of processes
- Shell
- Threads
- Concurrency

Memory

User
Unmapped
User
Unmapped
Kernel
User

 syscall & arguments
- syscall number (1) (rax)
- 1 (rdi)
- Pointer to "Yes
in" (rsi)

while

trap

while

write (1, "Yes
in", 4);
OS State for Processes

Process Control Block ← Lives in the kernel space

- Process ID
- State ← USED BY SCHEDULER (RUNNABLE/RUNNING/BLOCK)
- USER/GROUP ID
- FILE TABLE
- VM STRUCT CONTEXT

Kernel maintains a PCB array

fork in a tiny bit more detail

\[ A \xrightarrow{fork} A_c \]

Files:
- stdin (0)
- stdout (1)
- stderr (2)
Shell

How to start and compose applications.

Our focus:

- Talk about good abstractions

You have all already used a shell

```
cs202-user@....:~/cs202-labs$
```

A way to launch programs

Core loop:

```
while (---) {
  Get user input // "vim foo"

  Split into argv array // ["vim", "foo"]
  int pid = fork();
  if (pid == 0) {
    //
    execv(argv[0], argv...)
  } else {
    // to be done
  }
}
```
Composing Programs

$ ./foo > file  # Clean out file's content & write foo's output to file
$ ./foo >> file  # Append foo's output to file

$ ./foo < file    # Use file as input to foo

$ ./foo <./bar    # Use foo's output as input to bar.

All are implemented using the same mechanism
- Remember execve preserved file table

0 - stdin - where program input comes from
1 - stdout - where normal program output goes
            (printf)
2 - stderr - where error output goes
            (fprintf(stderr, ... ))

- Shell simply sets up files appropriately

  $ ./foo > file =
  # fork()
3

- Pipes (yes | head -5)

**ONLY SLIGHTLY MORE COMPLICATED**

L> NEED A SPECIAL TYPE OF FILE

See handout 2 posted on the class webpage!

**THREADS & WHAT**

```
// File example

close (1);
openat (1, "file", O_WRONLY, O_TRUNC);
```

```
// File example

proc
memory

fork

proc
memory

processes
```
**Problem: Concurrency**

How to prevent two threads from stepping on each other?

How to reason about interaction between threads?

More precisely:

- Each thread gets its own set of registers
- All threads in a process share the same memory.
\[ x = 1 \]
\[ y = 2 \]