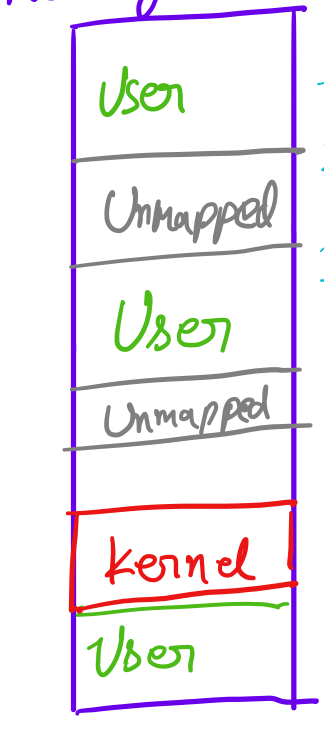
Last Time

- Stack Frames
- Syscalls (kind of)

Today

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



1 ~~if~~ (...) {

2 → write(1, "Yes\n", 4);

3 }

stdout

call f

PROCESSOR MODE

User

- Exception
- Interrupts

trap

write → syscall

: arguments

Calling Convention:
- syscall number (1) (rax)
- 1 (rdi)
- Pointer to "Yes\n" (rsi)
- ...

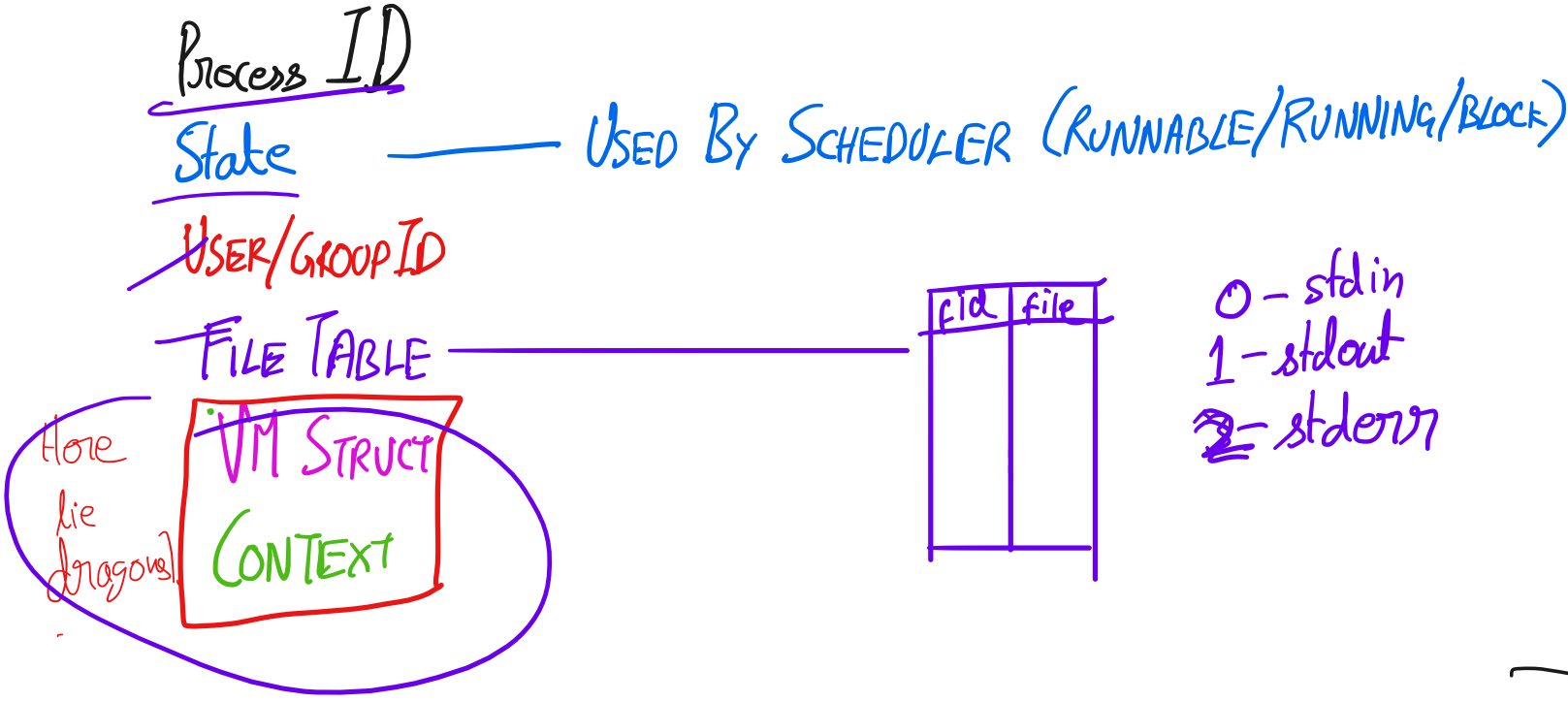
Syscall handler

...

iret: return in %rax

OS State for Processes

Process Control Block ← Lives in the kernel space



Kernel maintains a PCB array

fork in a tiny bit more detail

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 (__) {
        join(pid); //wait for process pid
                   // to be done
```

}
}

Composing Programs

```
$ ./foo > file
```
Clear 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 preserves 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 ≡

```
int pid = fork();
if (pid == 0) {
```

```
        close(1);
        openat(1, "file", O_WRONLY|
                          O_TRUNC);
        execv...
    }
```

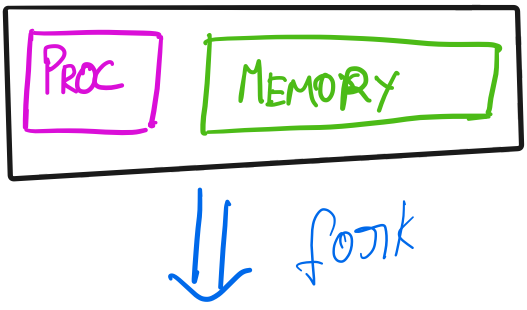
- Pipes (yes | head -5)

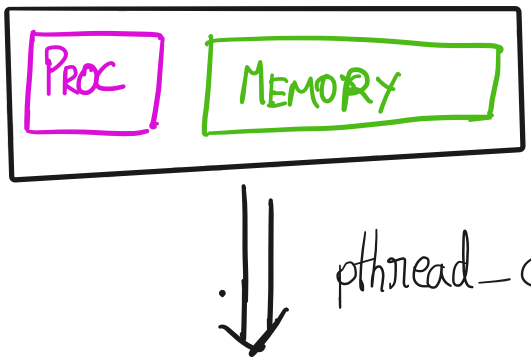
ONLY SLIGHTLY MORE COMPLICATED

↳ NEED A SPECIAL TYPE OF FILE

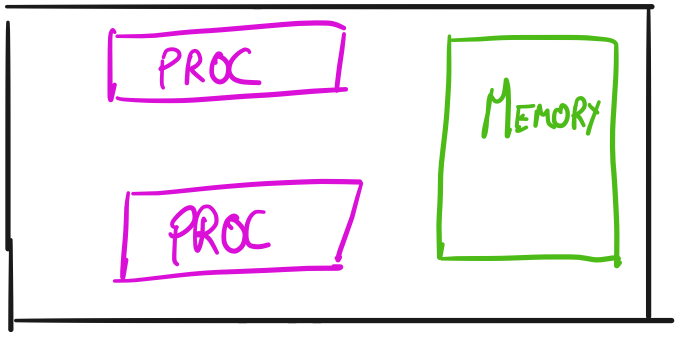
> See handout 2 posted on the class webpage!

THREADS: WHAT





pthread_create



More precisely

- Each thread gets its own set of registers
- All threads in a process share the same memory.

PROBLEM: CONCURRENCY

How to prevent two threads from stepping on each other?

How to reason about interaction between threads?

t1	t2

$x = 1$

$y = 2$