CS 202 (-002): Lecture 3

Processes (contd.)

Last Class

- Process
  - Why
  - What
    - Abst. Machine - Reg., Memory, Execution Unit
    - Memory
- Function Calls & Stack Frames
  - Prolog
    - → Run on Entry
    - → Creates Stack Frame
      - → The portion of the stack belonging to the function
    - → Changes some Reg.
      - → %rsp, %rbp
  - Epilog
    - → Run Before Exit
    - → Undoes what the Prolog Did

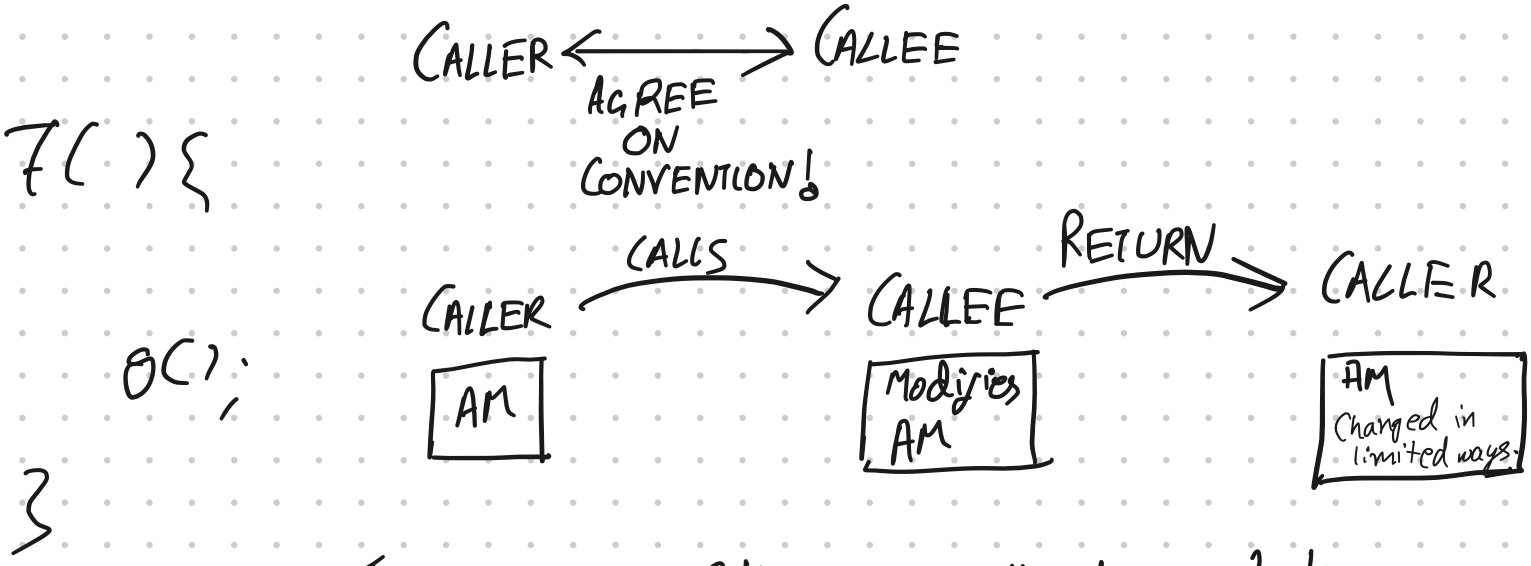
Today

- More Functions
  - Calling Convention
- Syscalls
  - → What
  - → How
- Shell

- Quiz.

□ Functions

Core Idea: A way to virtualize the process's Abstract Machine



Important → Callee generally does not know the caller's code
↦ Many possible callers
→ Conventions encapsulate all assumptions.

Last class ↔ How the stack is virtualized

Other Parts Of The AM:

- Registers

AKA
- Call preserved
- Callee saved:

Caller can expect that their values are unchanged

%RBP, %RSP ← Restored by EPILOG

Others include

caller → f() { → callee

g()

%RBX, %R12 – %R15

3

AKA
- Call clobbered

Caller saved: The callee can change these, the caller should not depend on their values

INCLUDE

- %RAX – Used for return values
- %RDI, %RSI, %RDX, %RCX, %R8, %R9
  ↳ See below
- %R10, %R11

DO NOT NEED TO REMEMBER WHAT REGISTERS ARE CALLEE– (CALLER-) SAVED

JUST THE IDEA!

- EVERYTHING ELSE
  1. EXEC. UNIT ← STATELESS !!
  2. REST OF MEMORY

CONVENTION ALSO DETERMINES HOW ARGUMENTS ARE PASSED & RETURNED.

- ARGUMENTS

ARG	0	1	2	3	4	5

%RDI, %RSI, %RDX, %RCX, %R8, %R9

```
void f(uint64_t a, uint64_t b);
void g() {
    ...
    f(42, 78);
    ...
}
```

```
g:
    pushq %rbx
    ...
    popq
```

RETURN: IN %RAX

↑ SAW THIS LAST CLASS IN HANDOUT!

- But really,
  A WAY TO VIRTUALIZE THE PROCESS'S ABSTRACT MACHINE

☐ Sometimes, code running in the process needs to call into the OPERATING SYSTEM.

Why?

- To open a file

```
int fd = open("hello.txt", ...)
```

- To read or write from file

```
int rd = read(fd, arr, len)
int wr = write(fd, arr, len)
```

- To create a process

int out = fork()

↑ Reading

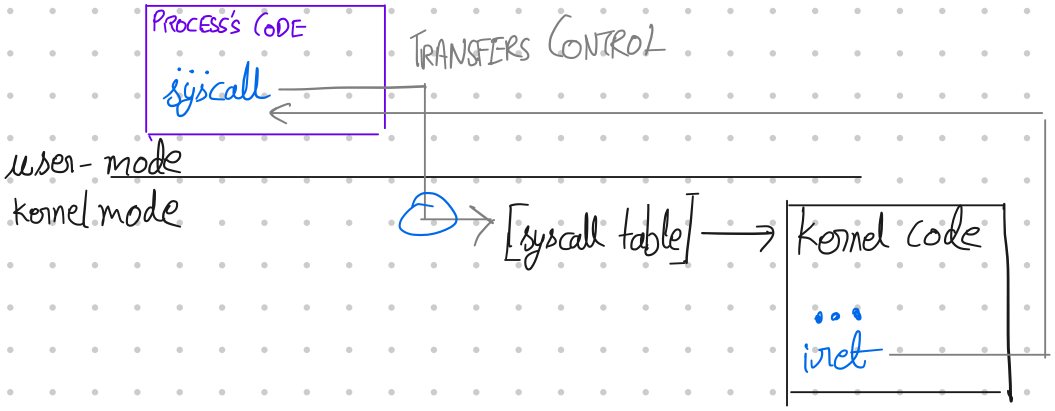
...

Small changes from the processes perspective.

① call ⟶ syscall

② Only %RAX is caller saved.

Big change from the computer's perspective



KERNEL/SUPERVISOR MODE

- MORE INSTRUCTIONS
- ACCESS TO DIFFERENT MEMORY
- ACCESS TO MORE REGISTERS

...

INTEL: RING 0 — 1 — 2 — RING 3

KERNEL — USER

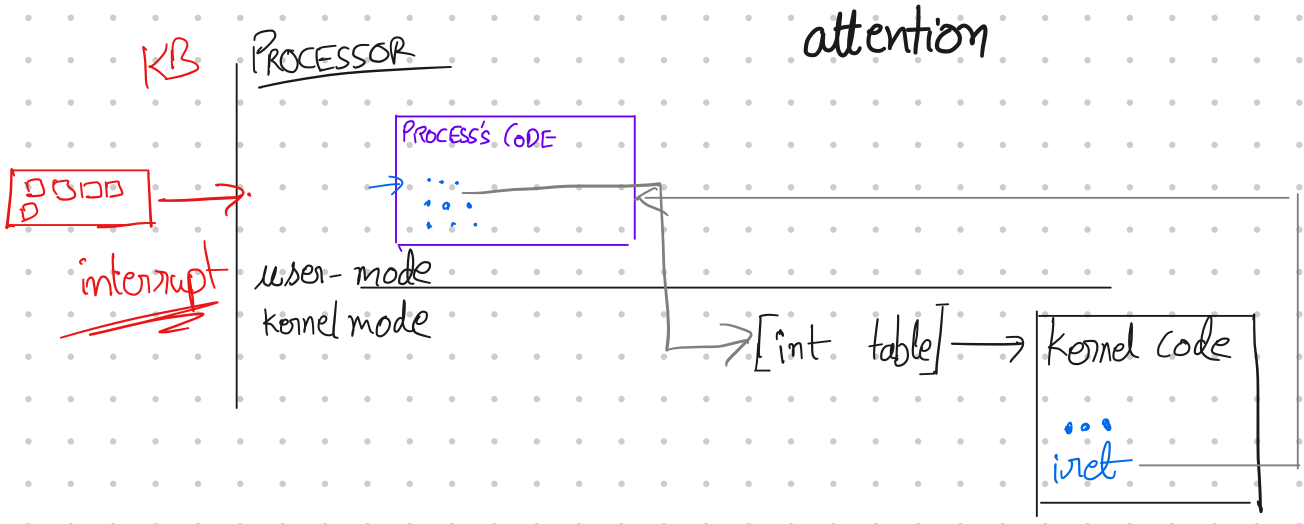
Ignore these for now.

syscall is only one way to enter the kernel

↳ TRAPS

- Syscall
- Interrupts

→ Device (e.g., keyboard) needs attention



- Exception
  - PROBLEM EXECUTING INSTRUCTION
    ↳ + Accessing an invalid memory location
    + Dividing by 0
    +...
  - KERNEL CAN HANDLE IN DIFF WAYS
    - KILL PROCESS
    - Notify (signal) the program

- Fix the problem

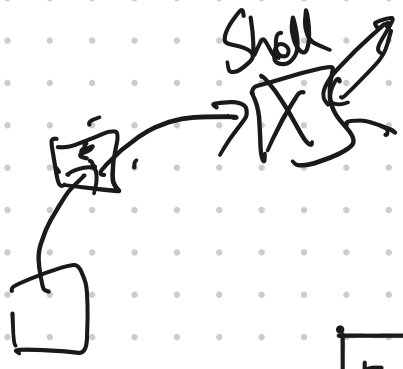
WILL LOOK AT THE KERNEL IN MORE DETAIL LATER, FOR NOW, JUST REMEMBER

- Two modes
  - ↳ kernel/supervisor mode can 'do more'
  - → User mode is where your prog. logic runs
- TRAPS go user → kernel
  - syscall/int/exception
- IRET returns to user

SHELL

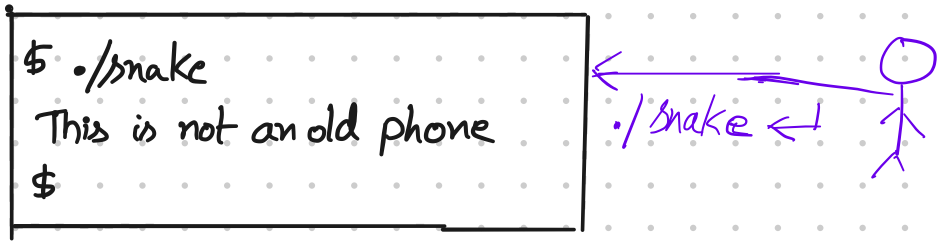
- Hopefully, all of you have used a shell by now
  - Required for setup!

- The Human (🧍) ↔ OS interface



↳ Start a program
→ Kill it
→ Pause it
→ etc.

} The GUI you are used to is basically another type of shell



What is going on

```
while (1) {
    printf ("$ ");
    readcommand (cmd, args);
    int pid = fork();
    if (pid==0) {
        execve (cmd, args, 0); printf("p");
    }else if (pid>0){
        wait(0);
    } else {
        ...
    }
}
```

FROM THE READING!

- Sometimes, you want output to go to a file

$ ./snake > snake-out

OR HOOK PROGRAMS TOGETHER

$ ls -al | more

Abstraction: 'Everything' is a file.

- Output to screen ← file

↳ stdout

→ File descriptor 1 (STDOUT_FILENO)

- Input ← file

↳ stdin

→ FD: 0 (STDIN_FILENO)

- Errors ← file

↳ stderr

→ FD: 2 ...

read, write, close ← Take fd as input

printf(...) ⟹ (eventually) write(1, "...")