Systems 1 – CS 6013

Computer Architecture and Operating Systems

Lecture 6: System Calls in ASM

MASTER OF SOFTWARE DEVELOPMENT (MSD) PROGRAM

J. DAVISON DE ST. GERMAIN

SPRING 2024

*Adapted from slides by Ben Jones, Matthew Flatt, Varun Shankar, and Others

Miscellaneous

- Have you started the assembly code project?
- Have you finished Week 2 readings?
  - 4 – Processes
  - 5 – Process API
  - 6 – Direct Execution
- Begin Week 3 readings…
- Questions on Makefiles or Name Mangling?

Lecture 5 – Topics

- System Calls
- Interrupts

Some More Assembly Instructions

- `call func` – pushes the current instruction pointer (+1) onto the stack, then jumps to the label (in this case, `func:`) specified.
- Note, `call` and `syscall` are similar, but not the same…
- `pop <reg>` – places the value at the top of the stack into `reg`
- `pop rax ; eg: call placed the IP onto stack, so pop retrieves it`
  `; Copy that value into rax`
- `jump rax ;Now jump to the address stored in rax`
  - What *register* does jump modify?
    - Places the value in `rax` into the `rip` (Instruction Pointer)
    - Note, the IP is often called the PC (Program Counter)
- `ret` – Combination of the `pop rax` and `jump rax` instructions

System Calls – Why?

- User programs need to do "dangerous" things (like I/O, creating new processes, checking the time, etc)
- OS's don't (and shouldn't!) trust user programs to have free access to the resources necessary for these things
  - e.g. If a program could create new processes – how would the OS manage them?
  - If a user writes to a directory structure wrong, it can break the filesystem!
- How do we let user programs do what they need in a (reasonably) safe way?

System Calls – What

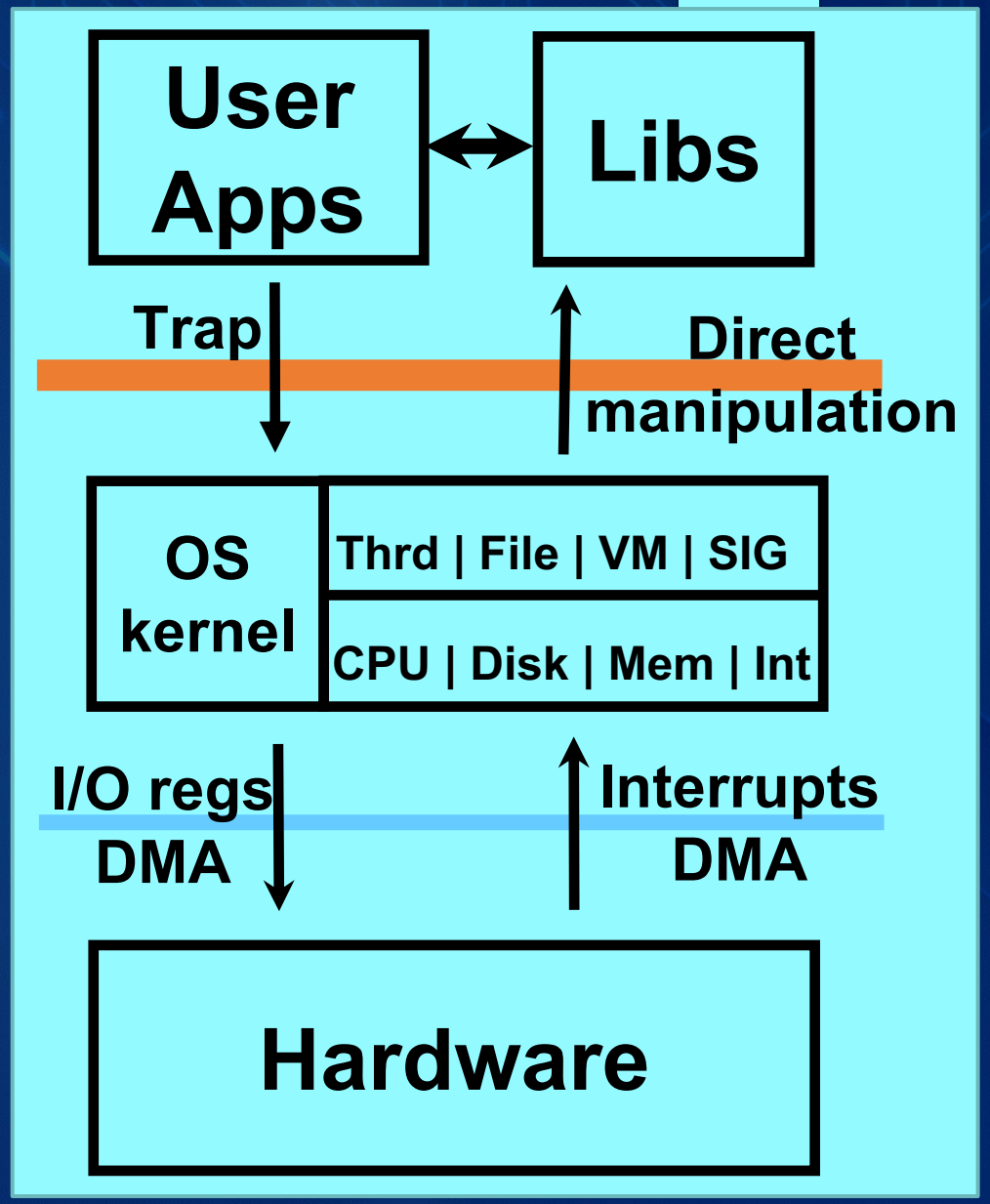
- A set of functions (basically) that user programs can call that request the kernel to do something for them
- The API looks like a normal function, but the ABI/implementation is subtly different.
- Recall that the CPU runs user programs in "user" mode, which does not allow some instructions (like the ones necessary for I/O)
- The code that actually performs I/O must be performed in *kernel* mode on the CPU
- When a syscall is made, the CPU switches to kernel mode, performs the task, then switches back to user mode when the syscall function returns

System Calls – Examples

- exit()
- fork()
- read() / write() / open() / close()
- chdir() / chmod()
- getpid()
- sbrk()
- sleep()

Standard OS Structure

- User-level
  - Applications
  - Libraries: many common "OS" functions
  - Example: malloc() vs sbrk()
- Kernel-level
  - Top-level: machine independent
  - Bottom-level: machine dependent
  - Runs in protected mode
  - Need a way to switch (user <–> kernel)
- Hardware-level
  - Device maker exports known interface
  - Device driver initiates operations by writing to device registers
  - Devices use interrupts to signal completion
  - DMA – Direct Memory Access (offloads work, but has restrictions)



How Are Sys Calls Implemented?

- Like normal function calls (ABI describes where parameters go, where the return value goes, etc)
- However, CANNOT be exactly the same because we MUST somehow get the CPU in kernel mode.
- How do we get the CPU into kernel mode?
- *interrupts*, *exceptions*, and *traps* — basically wakeup calls for the CPU
  - Interrupts: Some are hardware generated: key pressed, mouse moved, disk finished copying data, a timer went off, etc
  - Exceptions: Some are error reports: divide by 0, out of bounds memory access, etc (directly from the CPU or OS itself)
  - Traps: Some are caused by user mode programs calling privileged instructions: halt (the shutdown instruction), I/O type stuff, or the `syscall` instruction

Traps vs Interrupts vs Exceptions

- The main difference between a trap and an interrupt is…
  - A ***trap*** is triggered by a user program to invoke OS functionality while…
  - An ***interrupt*** is triggered by a hardware device to allow the processor to execute the corresponding interrupt handler routine.
- ***Exceptions*** are triggered inside the CPU itself (memory violations, divide by 0, etc) when running code.

The "Interrupt Vector"

- Each different type of ***interrupt/trap*** has a number associated with it. Some are defined by the hardware, some the OS decides what to do with
- When the computer boots up the OS, it will make/fill an array (vector). Each entry stores a pointer (address) to the code to run when that event occurs
- When an interrupt occurs, the CPU is switched into kernel mode and jumps to the appropriate ***interrupt handler*** code
- The interrupt handler has a special return-like instruction for setting the CPU back into user mode and returning.
- To make a system call, we set up the parameters we want, then trigger the interrupt that the OS knows is for syscalls. On x86_64, this is conveniently known as the `syscall` instruction.
  - Note older code uses ***int 0x80*** instead of ***syscall***

How to make a Syscall

- Put the `syscall` number in the appropriate registers (again, part of the ABI)
- Put the `syscall` parameters in the appropriate registers (there's an ABI for system calls)
- Execute the `syscall` ASM instruction which will switch the CPU to kernel mode and execute the interrupt handler
- The kernel actually executes the system call then returns to the next line
- Do whatever else is necessary (possibly moving the return value to the right spot)

- `call function1`

  vs

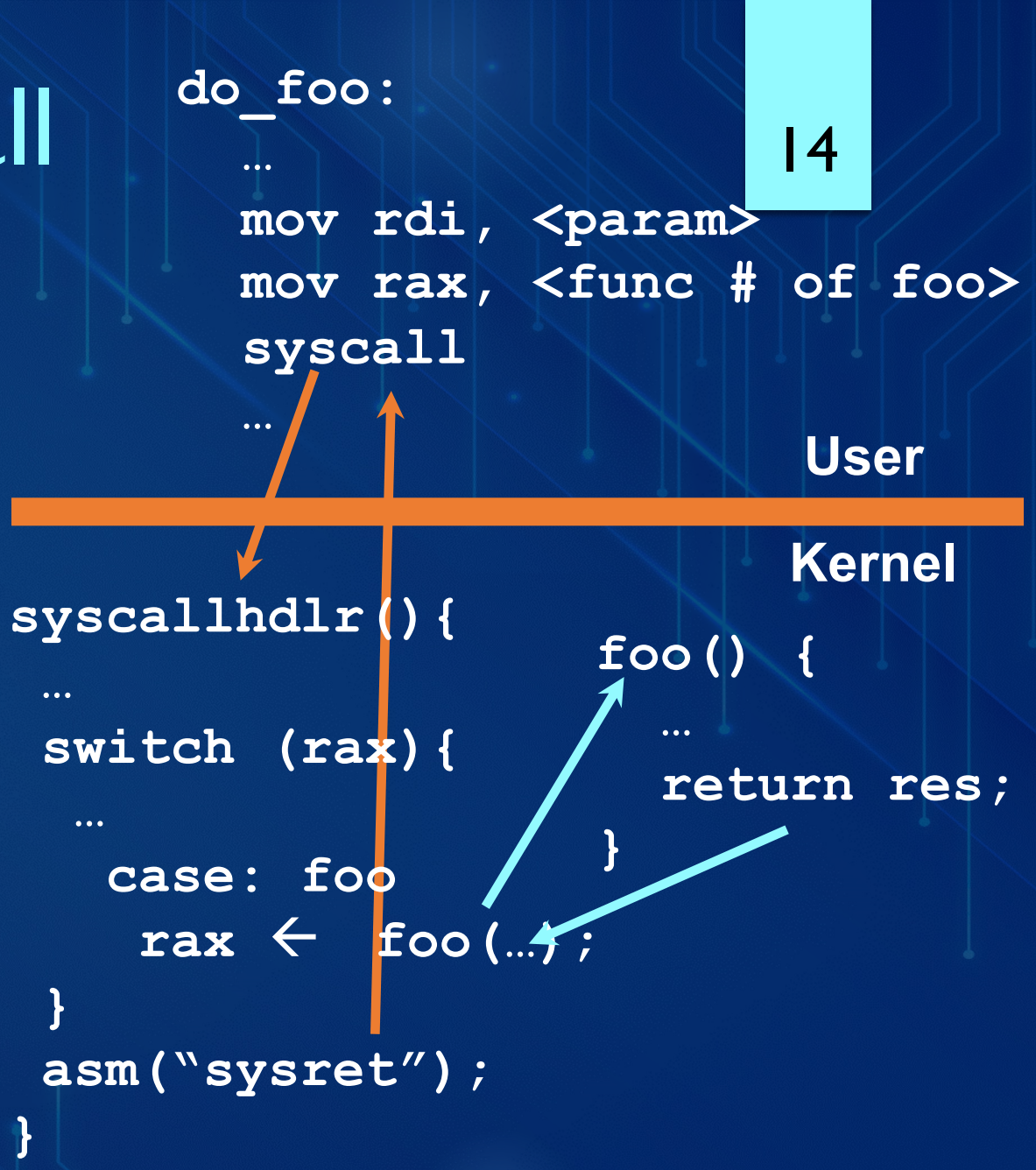
- `syscall`
- How does `call` know what to set the `IP` to?
- How does `syscall`?

System Calls

- Arguments passed in registers:
  - Integer constants (e.g., buffer size, file offset, file id)
  - Pointers to blocks of memory (e.g., strings, file buffers, …)
  - Handles (e.g., file handle, socket handle, …)
- OS typically returns:
  - Return code (value of `-1` often denotes error)
  - Other results written into buffers in user space
  - You should always (always!) check syscall return values
- Principle: Dialogue between user-mode and kernel should be semantically simple
  - Simpler interfaces easier to work with
  - Simpler interfaces easier to implement correctly
  - Simpler interfaces tend to be easier to make efficient

Anatomy of a System Call

- User applications make system calls to execute privileged instructions
- Anatomy of a system call:
  - Program puts syscall params in registers
  - Program executes a trap:
    - Minimal processor state (PC, PSW) pushed on stack
    - CPU switches mode to KERNEL
    - CPU vectors to registered trap handler in the OS kernel
- Trap handler uses param to jump to desired handler (e.g., fork, exec, open, write, …)
- When complete, reverse operation
  - Place return code in register
  - Return from exception



~ Fin ~