System Calls, Signals, & Interrupts

CS 3410: Computer System Organization and Programming

Spring 2025

[K. Bala, A. Bracy, G. Guidi, A. Sampson, E. Sirer, Z. Susag, and H. Weatherspoon]

“Spring” Break!

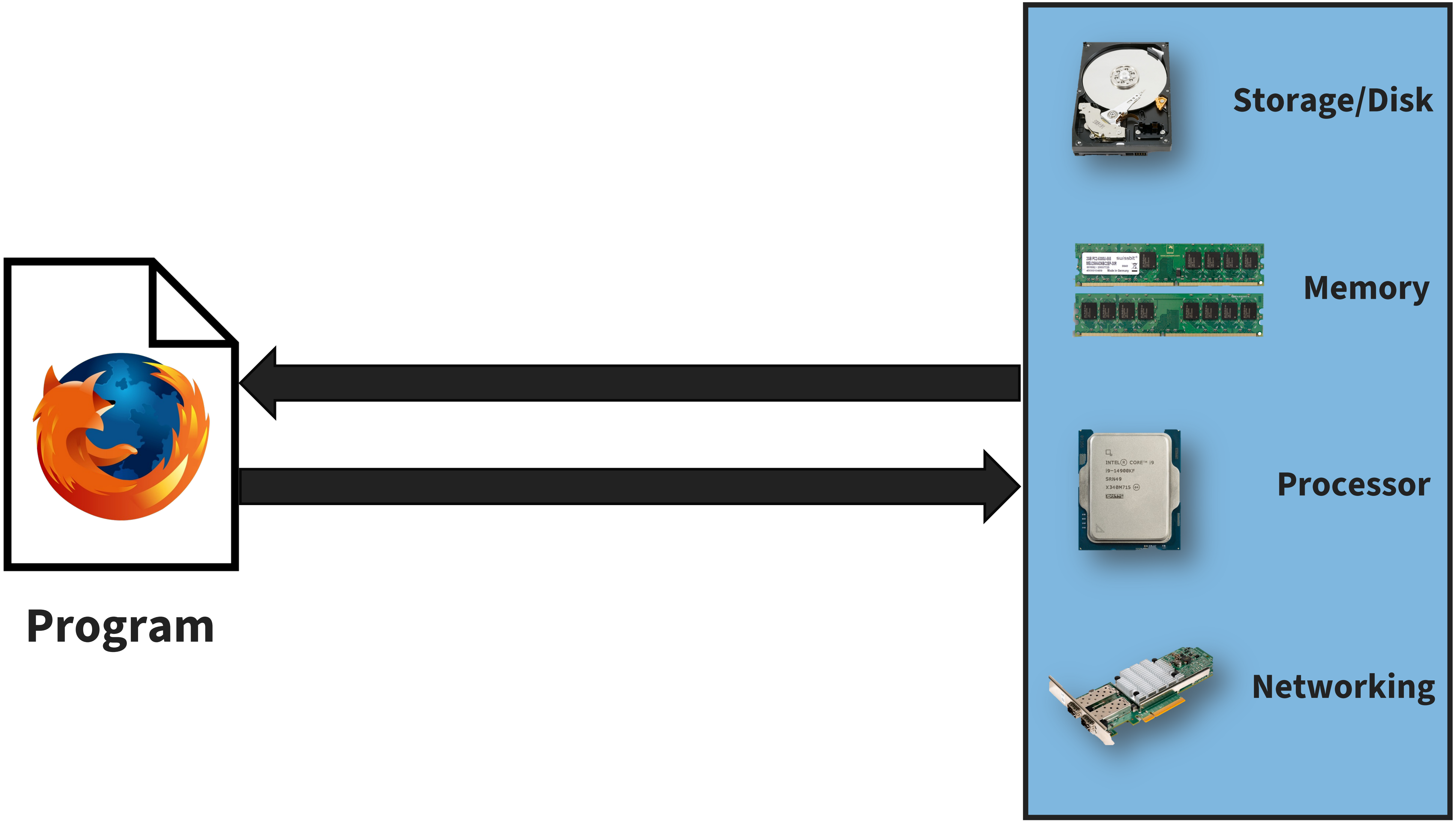
- No lab this week!
- No work over break…except studying for Prelim 2
- Have fun, be safe!

Where are you going?

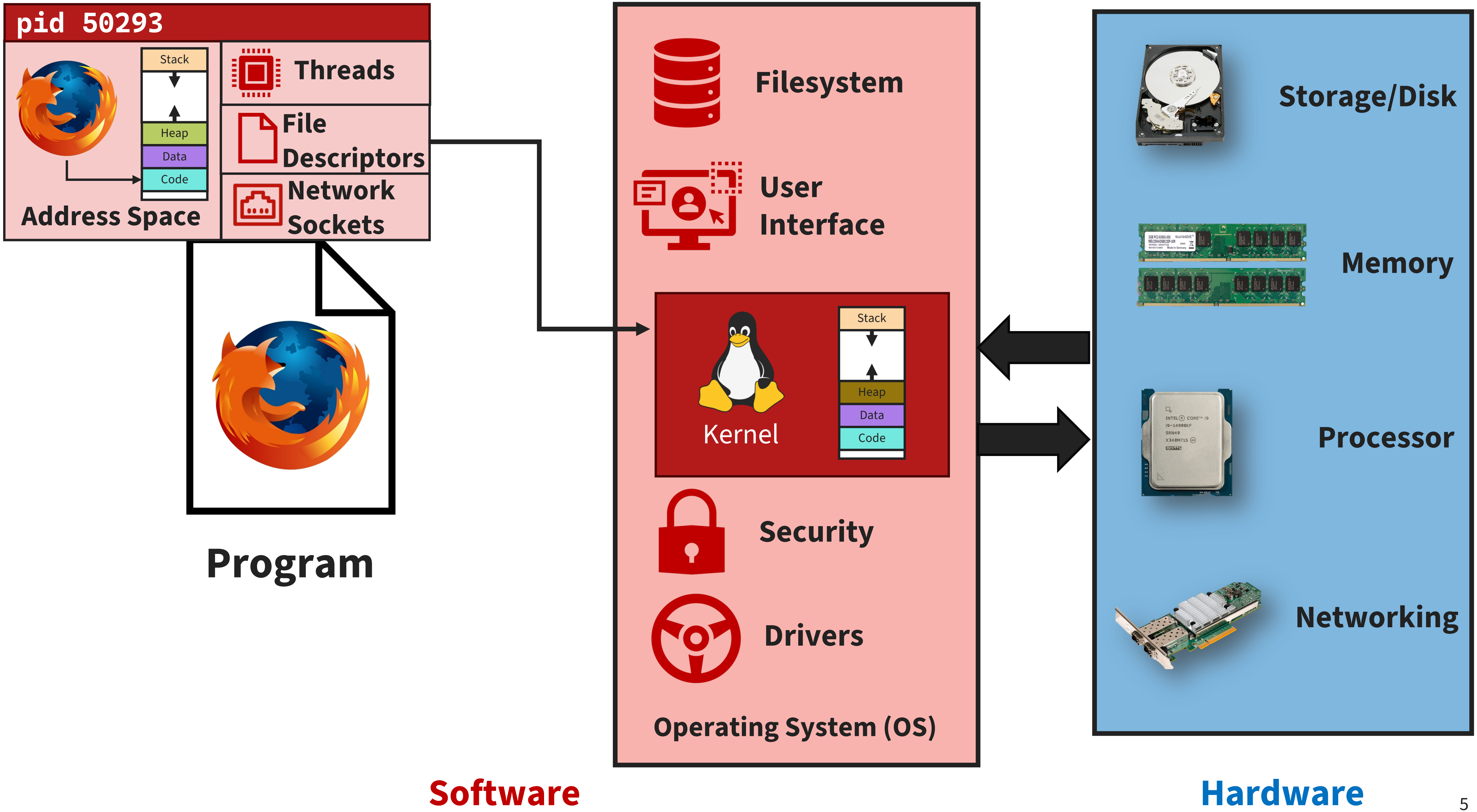
PollEv.com/zjs

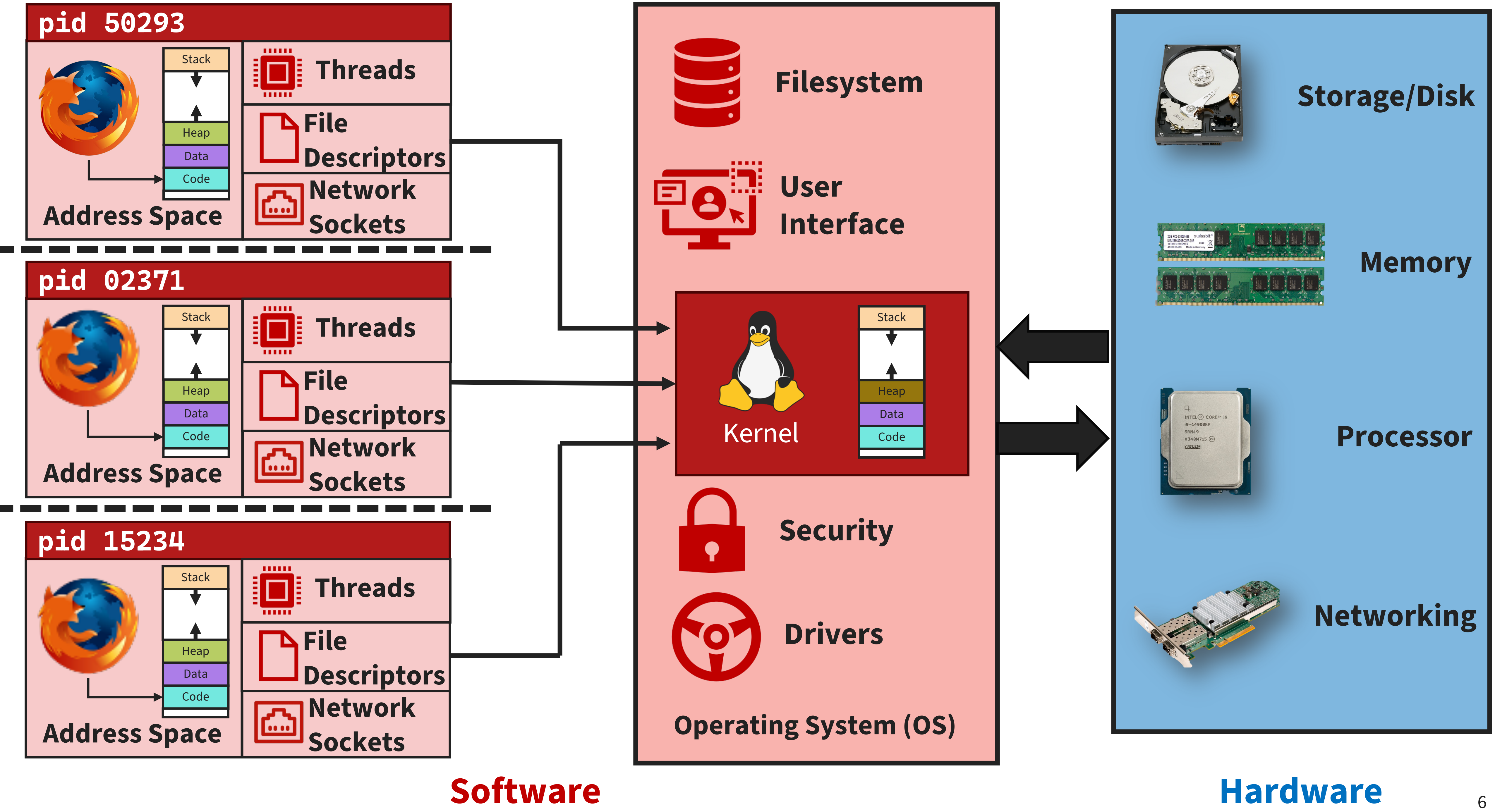
Today's Goals

- Review: Processes
- System Calls: A C Perspective
- Exceptional Control Flow
  - Signals
  - Interrupts
- System Calls: How They Actually Work

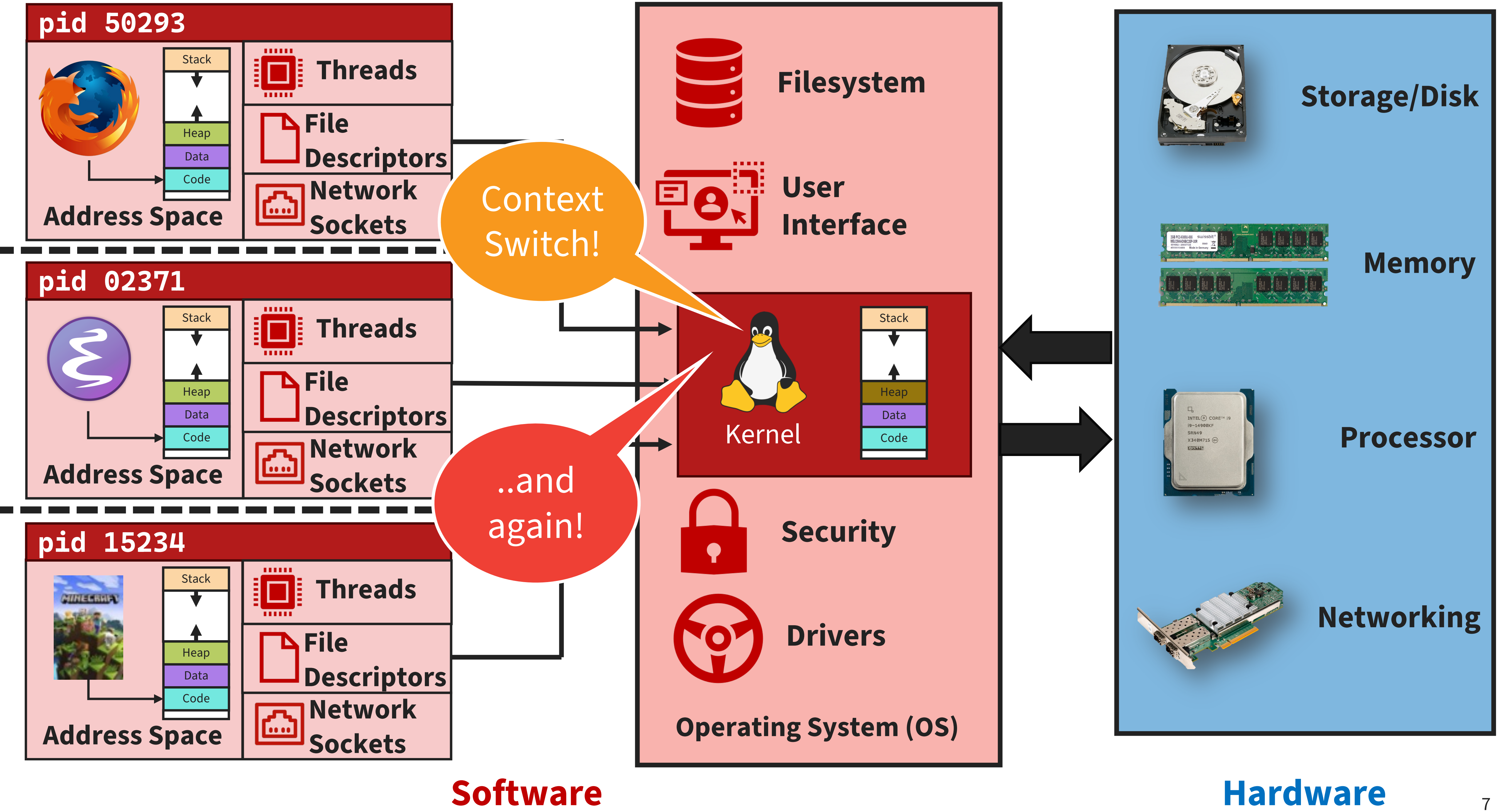


Software **Hardware**


pid 50293
Stack
Heap
Data
Code
Address Space
Threads
File Descriptors
Network Sockets
Program
Filesystem
User Interface
Stack
Heap
Data
Code
Kernel
Security
Drivers
Operating System (OS)
Storage/Disk
Memory
Processor
Networking
Software
Hardware



pid 50293
Stack
Heap
Data
Code
Address Space
Threads
File Descriptors
Network Sockets
pid 02371
Stack
Heap
Data
Code
Address Space
Threads
File Descriptors
Network Sockets
pid 15234
Stack
Heap
Data
Code
Address Space
Threads
File Descriptors
Network Sockets
Filesystem
User Interface
Kernel
Stack
Heap
Data
Code
Security
Drivers
Operating System (OS)
Storage/Disk
Memory
Processor
Networking
Software
Hardware

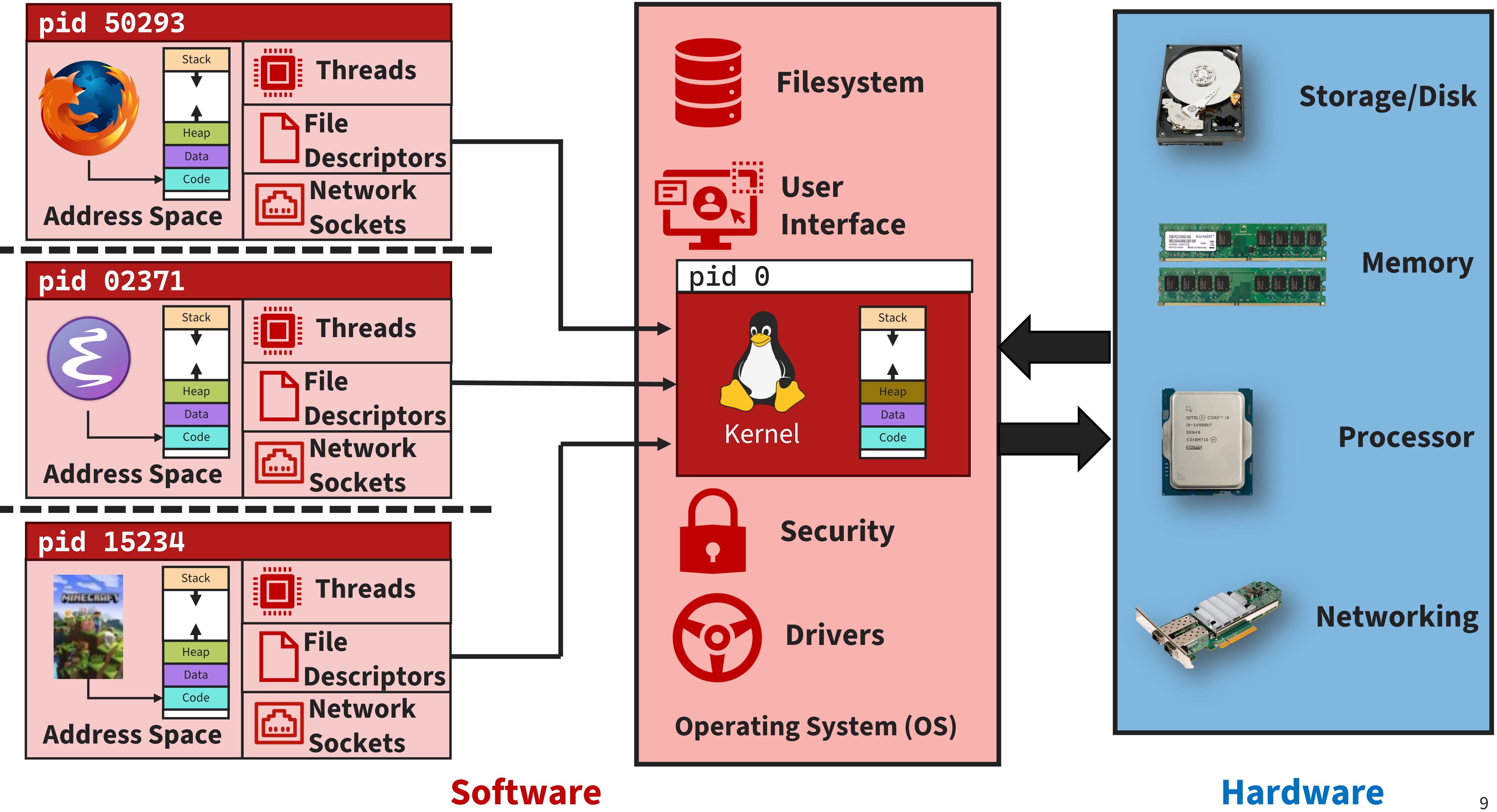

pid 50293
Stack
Heap
Data
Code
Address Space
Threads
File Descriptors
Network Sockets
pid 02371
Stack
Heap
Data
Code
Address Space
Threads
File Descriptors
Network Sockets
pid 15234
MINECRAFT
Stack
Heap
Data
Code
Address Space
Threads
File Descriptors
Network Sockets
Context Switch!
..and again!
Filesystem
User Interface
Kernel
Stack
Heap
Data
Code
Security
Drivers
Operating System (OS)
Storage/Disk
Memory
Processor
Networking
Software
Hardware

PollEverywhere

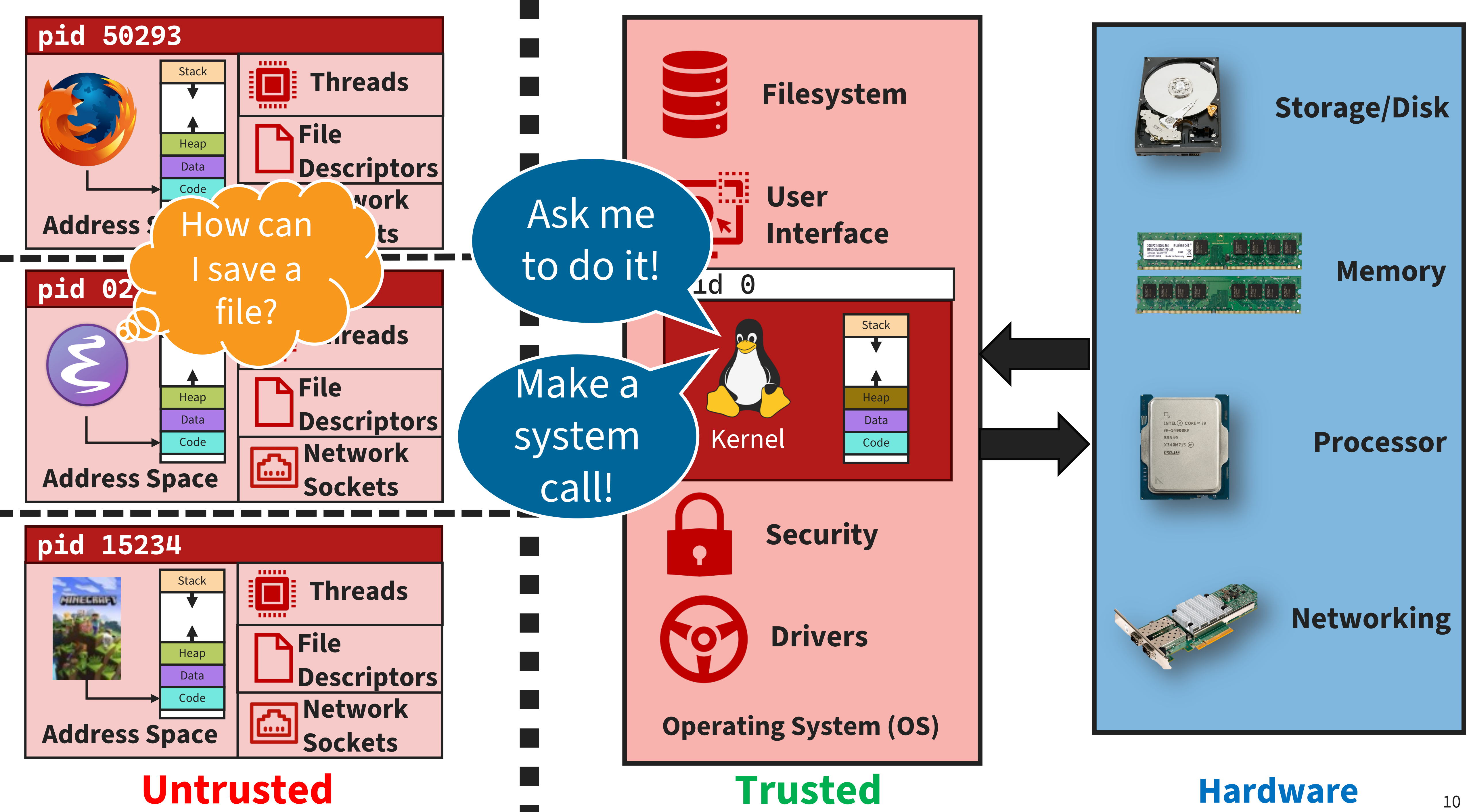
No matter what process is currently running on the CPU, the operating system is always actively running in the background.

a) True
b) False

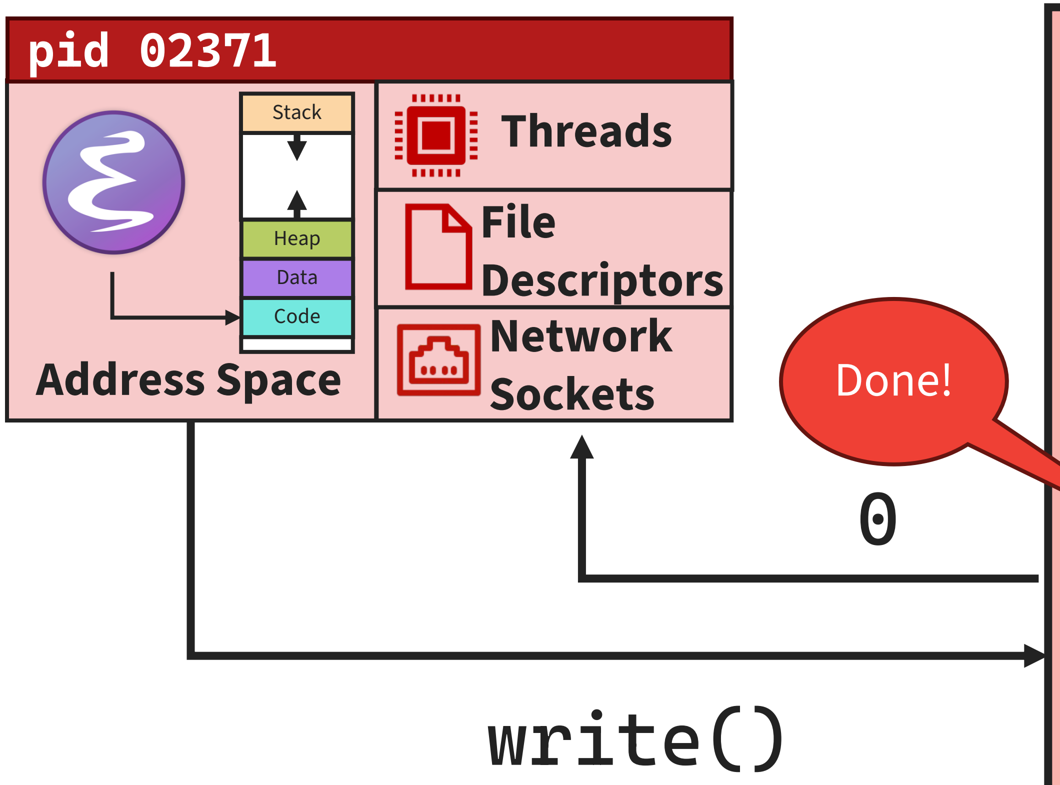
PollEv.com/zjs



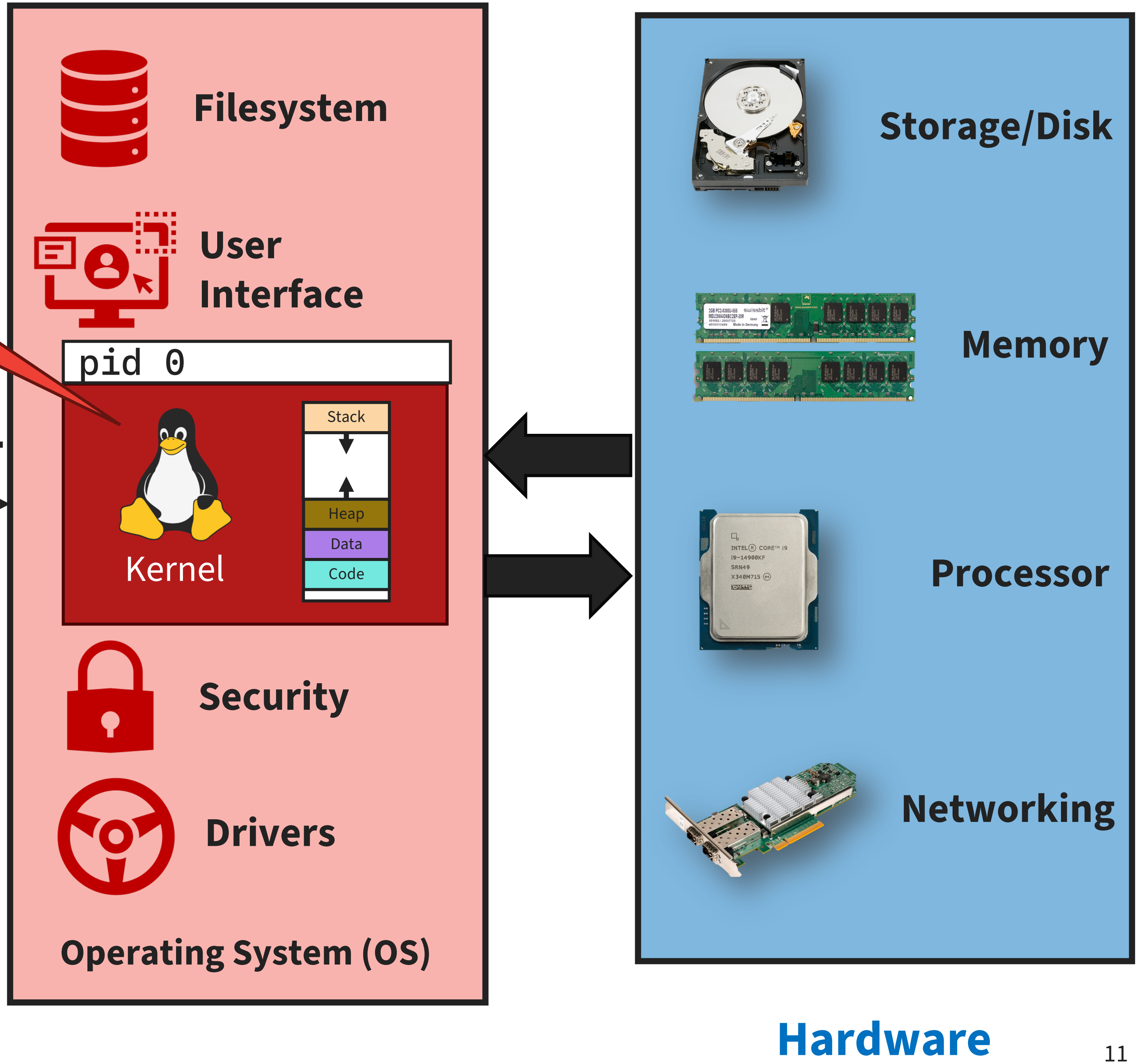
pid 50293
Stack
Heap
Data
Code
Address Space
Threads
File Descriptors
Network Sockets
pid 02371
Stack
Heap
Data
Code
Address Space
Threads
File Descriptors
Network Sockets
pid 15234
Stack
Heap
Data
Code
Address Space
Threads
File Descriptors
Network Sockets
Filesystem
User Interface
pid 0
Kernel
Stack
Heap
Data
Code
Security
Drivers
Operating System (OS)
Storage/Disk
Memory
Processor
Networking
Software
Hardware



pid 50293
Stack
Heap
Data
Code
Address Space
Threads
File Descriptors
Network Sockets
pid 02
Threads
File Descriptors
Network Sockets
Address Space
pid 15234
Stack
Heap
Data
Code
Threads
File Descriptors
Network Sockets
Address Space
How can I save a file?
Ask me to do it!
Make a system call!
Filesystem
User Interface
pid 0
Kernel
Stack
Heap
Data
Code
Security
Drivers
Operating System (OS)
Storage/Disk
Memory
Processor
Networking
Untrusted
Trusted
Hardware



Asks the kernel to write an *n*-character long string to the file referenced by the provided file descriptor



System Calls

System Calls

- A mechanism for processes to request the services from the kernel
    - File manipulation
    - Network communication
    - Process management
- Allows processes to perform **privileged operations** while running in user space
- OS defines **minimal** set of system calls
    - **Abstraction layer** between kernel and user code
- Why minimal?
    1. Portability: easier to implement and maintain
    2. Security: small “attack surface”; easier to protect against vulnerabilities

Portable Operating System Interface (POSIX)

- Originally published in 1988 under the name *IEEE Std 1003.1-1988*
  - Now 19 separate documents
- Colloquially referred to as Unix
- Implemented on multiple OSs
- `unistd.h`

IEEE COMPUTER SOCIETY

Common System Calls

- **`read()`**: Reads data from a file descriptor
- **`write()`**: Writes data to a file descriptor
- **`open()`**: Opens a file and returns a file descriptor
- **`close()`**: Closes an open file descriptor
- **`fork()`**: Creates a new process
- **`exec()`**: Replaces the current process image with a new executable
- **`waitpid()`**: Waits for a specific child process to change state

Error Handling

- System calls often return -1 to indicate an error
- The global variable **errno** is set to indicate the specific error code
- The **perror()** function prints a human-readable error message based on the value of **errno**

```c
#include <stdio.h>
#include <unistd.h>

const char msg[] = "CS 3410!";
int ret = write(STDOUT_FILENO, msg, sizeof(msg)-1);
if (ret == -1) {
  perror("write");
}
```

Demo: A Tale of Three Syscalls

… named `fork()`, `exec()`, and `waitpid()`

How Are Processes Created?

- `fork()`:
  - Allocates process ID **`pid`**
  - Creates and initializes PCB
  - Creates and initializes new address space
  - Informs scheduler a new process is ready
- `exec(program, arguments)`:
  - Loads program into the address space
  - Copies arguments into memory address space
  - Initializes hardware context to start execution at “start”

How Do Processes Terminate?

- **`exit()`**: used by a process to terminate itself
- **`abort()`**: used by a parent process to terminate a child process
- **`wait()`** and **`waitpid()`**: used by a parent process to wait for a child process to terminate and retrieve its exit status

PollEverywhere

The operating system can interrupt user code using system calls.

a) True
b) False

PollEv.com/zjs

Signals

Signals

Signals are the mechanism for the kernel (or another process) to communicate with processes.

Common Signals:

- **SIGINT**: the **interrupt** signal
- **SIGTERM:** the (graceful) termination signal requests a process to terminate
- **SIGKILL**: the kill signal *forces* a process to terminate immediately
    - **Cannot be caught or ignored**
- **SIGSEGV**: the segmentation fault
- **SIGCHLD**: sent to parent process when a child process terminates or stops

Sending Signals

`kill(pid_t pid, int sig)`

`kill()` sends the signal **`sig`** to the process with the process ID **`pid`**

`raise(int sig)`

`raise()` sends the signal **`sig`** to the calling process

Handling Signals

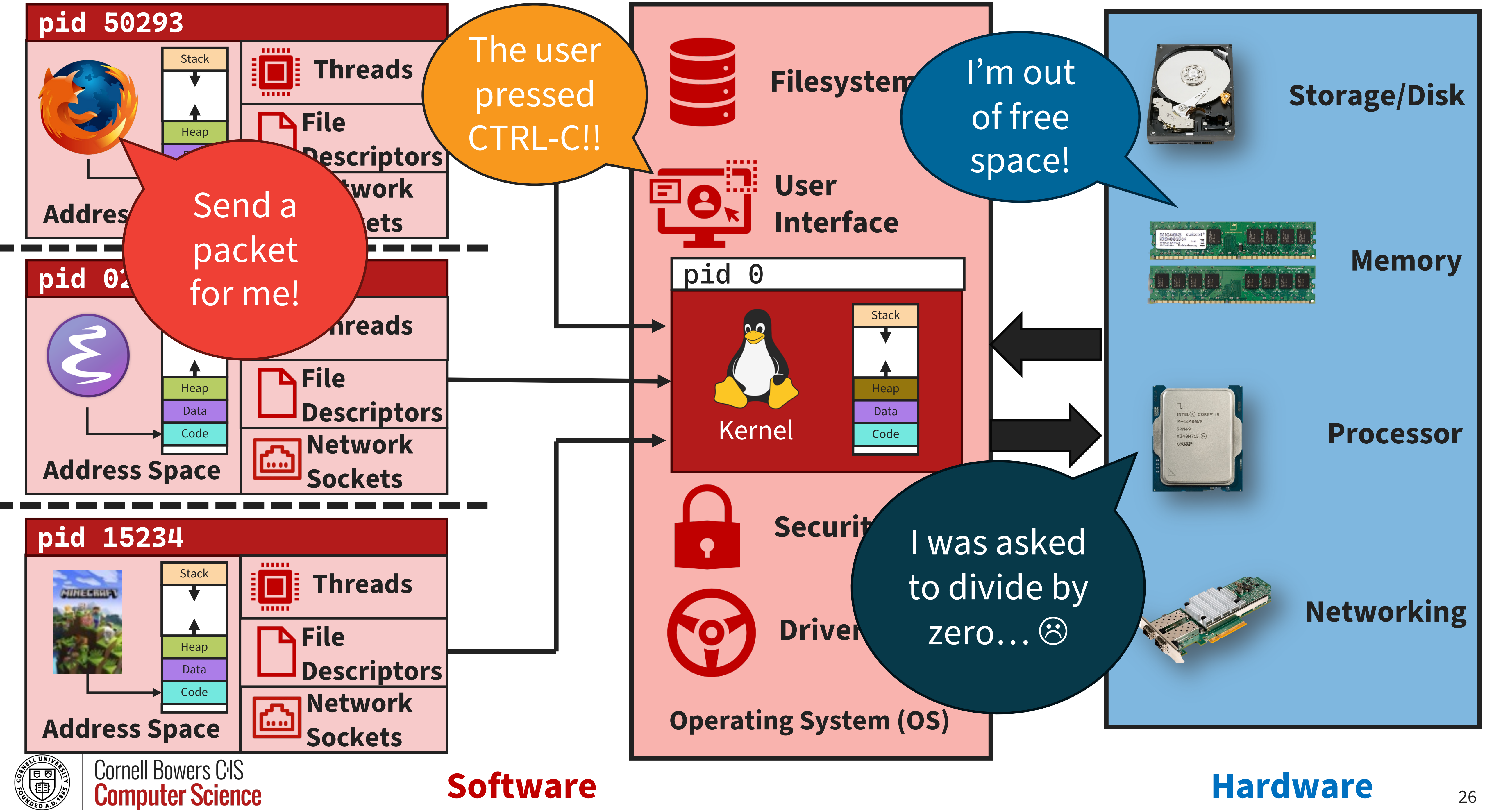
```c
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <unistd.h>

void handle_signal(int sig) {
  printf("Caught signal %d\n", sig);
  exit(1);
}

int main() {
  signal(SIGINT, handle_signal); // Set up the signal handler for SIGINT.
  while (1) {
    printf("Running. Press Ctrl+C to stop.\n");
    sleep(1);
  }
  return 0;
}
```

Exceptional Control Flow

Traps, Faults, and Interrupts


pid 50293
Stack
Heap
Address
Threads
File
Descriptors
twork
ets
Send a packet for me!
The user pressed CTRL-C!!
pid 02
hreads
Heap
Data
Code
Address Space
File
Descriptors
Network
Sockets
pid 15234
Stack
Heap
Data
Code
Address Space
Threads
File
Descriptors
Network
Sockets
Filesystem
User
Interface
pid 0
Kernel
Stack
Heap
Data
Code
Securit
Driver
Operating System (OS)
I'm out of free space!
I was asked to divide by zero... ☹
Storage/Disk
Memory
Processor
Networking
Software
Hardware

Types of Interrupts

An **interrupt** is an unscheduled event that needs immediate attention which disrupts the normal execution of a program.

Hardware Interrupts:

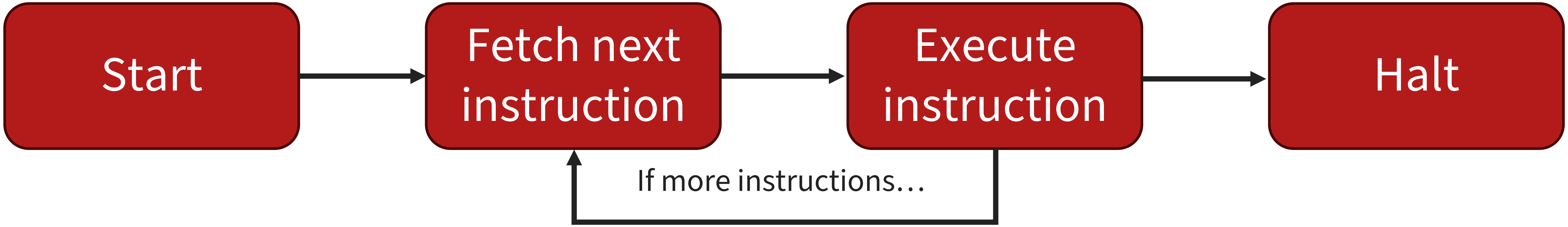
- Generated by hardware devices
    - Ex: keyboard input, hardware failure
- Some can be ignored (maskable); some can't (unmaskable)

Software Exceptions:

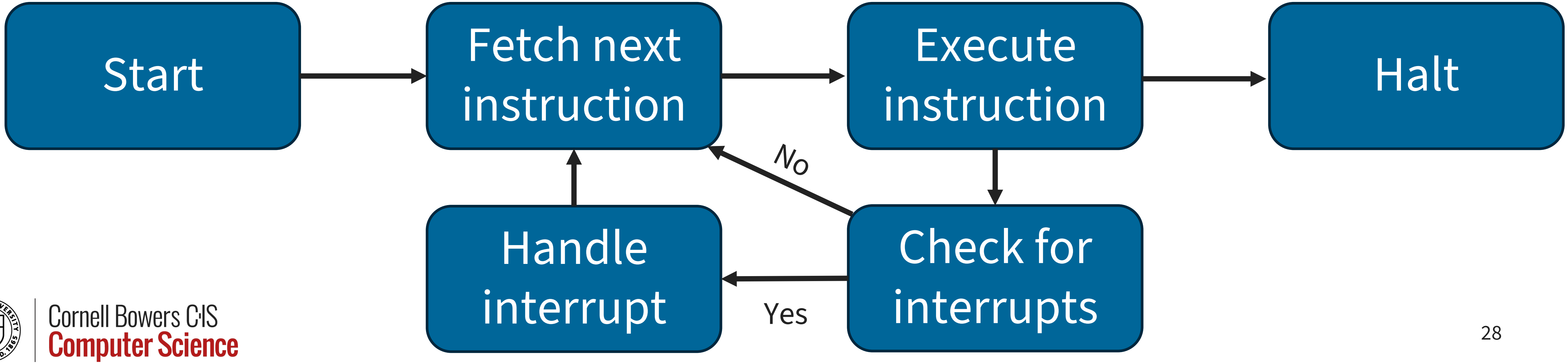
- Generated by programs (i.e., instructions!)
- Intentional interrupts are **traps** (e.g., system calls)
- Unintentional interrupts are **faults** (e.g., divide by zero, segmentation fault)

Instruction Cycle & Interrupts

Without Interrupts



With Interrupts



How Interrupts Work

- **Interrupt Signal**
  - An interrupt signal is sent to the CPU by a hardware device or software
- **Saving State**
  - The CPU saves the current state of the running process (e.g., program counter, registers) so it can resume execution later
- **Interrupt Handling**
  - The CPU transfers control to the interrupt handler associated with the interrupt. The interrupt handler processes the event (e.g., reading data from a device)
- **Restoring State**
  - The CPU restores the saved state and resumes execution of the interrupted process

What are Interrupts For?

EFFICIENCY

RESPONSIVENESS

MULTITASKING

How System Calls Actually Work

Let’s write “Hello, World!” without the standard library!

Executing a System Call in RISC-V

Process (User Mode)

1. Call systems call function in standard library
2. Place function args. in `a0–a5` and/or **user stack**
3. Place syscall number in `a7`
4. Call `ecall`

…

Hardware

5. Save registers to **kernel stack** and current caller PC in special register `sepc`
6. Switch to **kernel mode**
7. Lookup syscall handler address in trap table
8. Jump to syscall handler

12. Restore registers from **kernel stack**
13. Switch to **user mode**
14. Jump to instr. after `ecall`

Kernel (Kernel Mode)

9. Do work of syscall
10. Place result in `a0`
11. Execute “supervisor exception return” `sret`

Unprivileged vs. Privileged Specifications

RISC-V Technical Specifications

Owned by Jeff Scheel •••
Last updated: Feb 21, 2025 • 5 min read

Below is a comprehensive list of all ratified technical publications.

[ISA Specifications] [Profiles] [Non-ISA Specifications] [Compatibility Test Framework]

ISA Specifications

These are the current, published versions of the ISA specifications. Prior published versions and the original ratification specifications for included extensions can be found on the RISC-V Technical Specifications Archive page.

Specification name (PDF link)	Version	Published	RISC-V Community	Source Repository
The RISC-V Instruction Set Manual Volume I: Unprivileged ISA	20240411	May 2024	Unprivileged Horizontal Committee	riscv/riscv-isa-manual
The RISC-V Instruction Set Manual Volume II: Privileged Architecture	20240411	May 2024	Privileged Horizontal Committee	riscv/riscv-isa-manual

Demo

```
.global printone
printone:
  mv t0, a0         # save the function
argument: a character pointer

  # Make a system call: write(0, t0, 1)
  addi a7, x0, 64  # syscall number: write
  addi a0, x0, 0   # first argument: fd
  mv   a1, t0      # second argument: buf
  addi a2, x0, 1   # third argument: count
  ecall

  ret
```

```
int printone(char* c);

int main() {
    printone("h");
    printone("i");
    printone("\n");
    return 0;
}
```

Today's Goals

- Review: Processes
- System Calls: A C Perspective
- Exceptional Control Flow
  - Signals
  - Interrupts
- System Calls: How They Actually Work