Agenda

Operating System Concepts

- Processes
- Users
- Kernel
- System Calls
- Booting
- Environment

Operating System Concepts: Process

A **process** is an instance of a running program.

Each process gets its on own **address space** and a unique **process Identifier (PID)**

The information about each running process is stored in a **process table**

Communication between processes is called **Interprocess Communication (IPC) -** pipes, sockets, shared memory

On UNIX, new processes are created with **fork**

The **program status word (PSW)** contains information about the program mode, priority, and more

Kernel

Implements core OS functionality

- Mechanisms for hardware to run programs (e.g. software, applications)
- Policies for efficiently managing and sharing resources

Implements the **system call interface**

- APIs for interacting with the hardware
- Examples: gettimeofday, open, fork

Process address space

Associated with each process is an **address space**

Contains:

- Executable program code (**text sement**)
- Function stack (**stack segment**)
- Heap Memory (**data segment**)
- Resources (open files, and more)

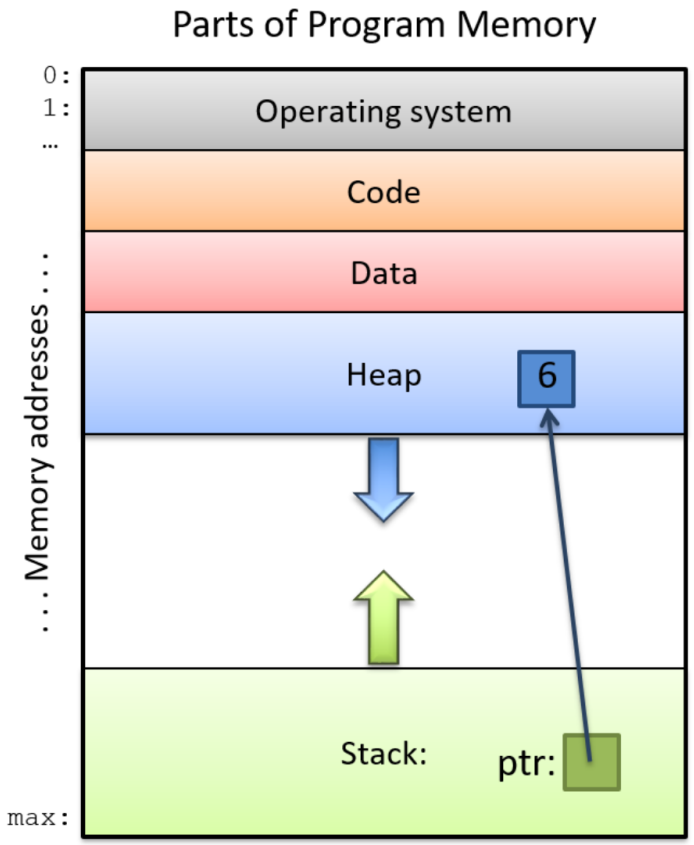


Figure 17. A pointer on the stack points to a block of memory that was allocated from the heap.

Process tree

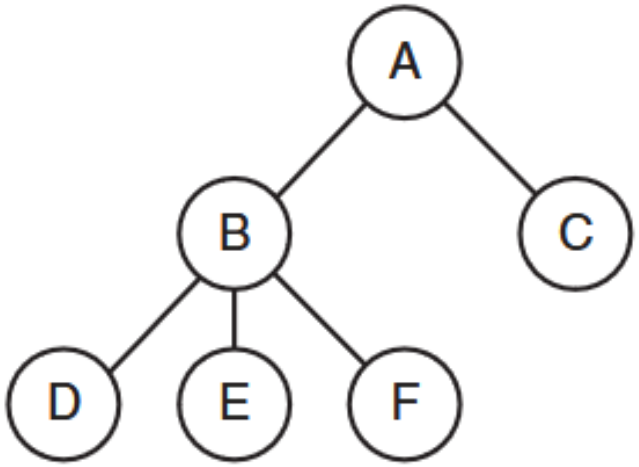


Figure 1-13. A process tree. Process *A* created two child processes, *B* and *C*. Process *B* created three child processes, *D*, *E*, and *F*. Tannenbaum

Processes can spawn other processes (called child processes)

Can you think of an example?

To list of all active processes: **`ps`** (Unix), **`Ctl-Alt-Del`** (Windows)

Users

Each person authorized the use a system is assigned **a user IDentification (UID),** e.g. your username

- alphanumeric, unique

The UID is associated with processes, files

Users can be members of groups, which have a GID

Permissions of files are tied to UID, GIDs

The **superuser** (UNIX) or **administrator** (Window) is a special UID that can override protection rules

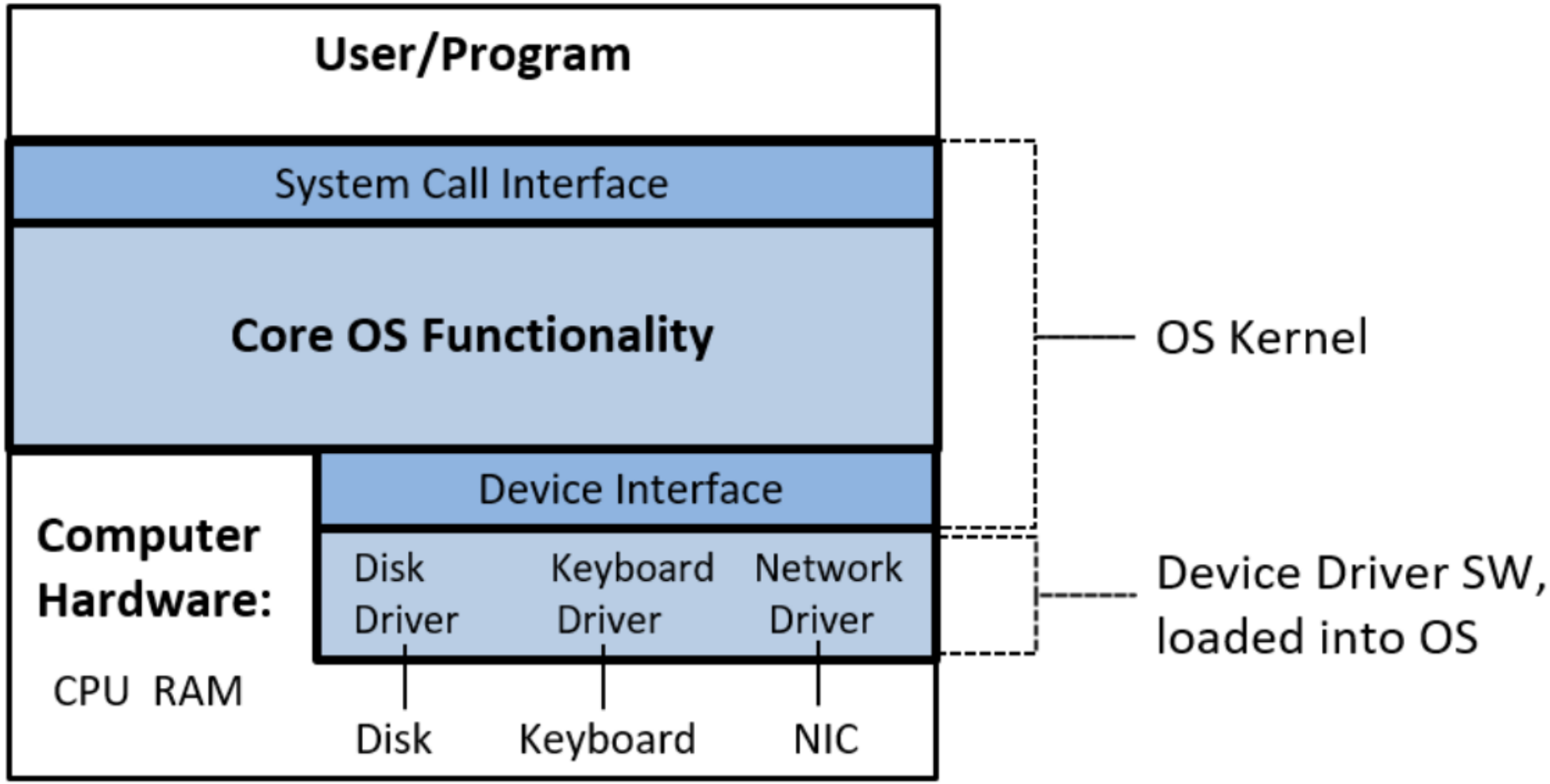
System Calls

To obtain "direct access" to operating system services, user programs must make **system calls**

Programs that make system calls are traditionally called "system programs" and were traditionally implemented in assembly language

System calls invoked from programs by calling a library procedure which calls low-level procedures executed by the **kernel**

Kernel and Operating System



Dive into systems

Kernel Mode and User Mode

- When a CPU is in kernel mode, it can do anything and address any part of memory
  - OS kernel runs in supervisor mode
  - Supervisors can switch to user mode at will

- User mode only has access to its own address space and can’t talk directly to devices
  - User processes run in user mode
  - Must execute a “trap” to switch to kernel mode

Why would we want two modes for accessing hardware?

Example system calls

Working with files: open, close, read, write, link, unlink, lseek, stat

Working with directories: mkdir, rmdir, chdir

Working with processes: fork, wait, exec

Working with time: time

Working with disks: mount, umount

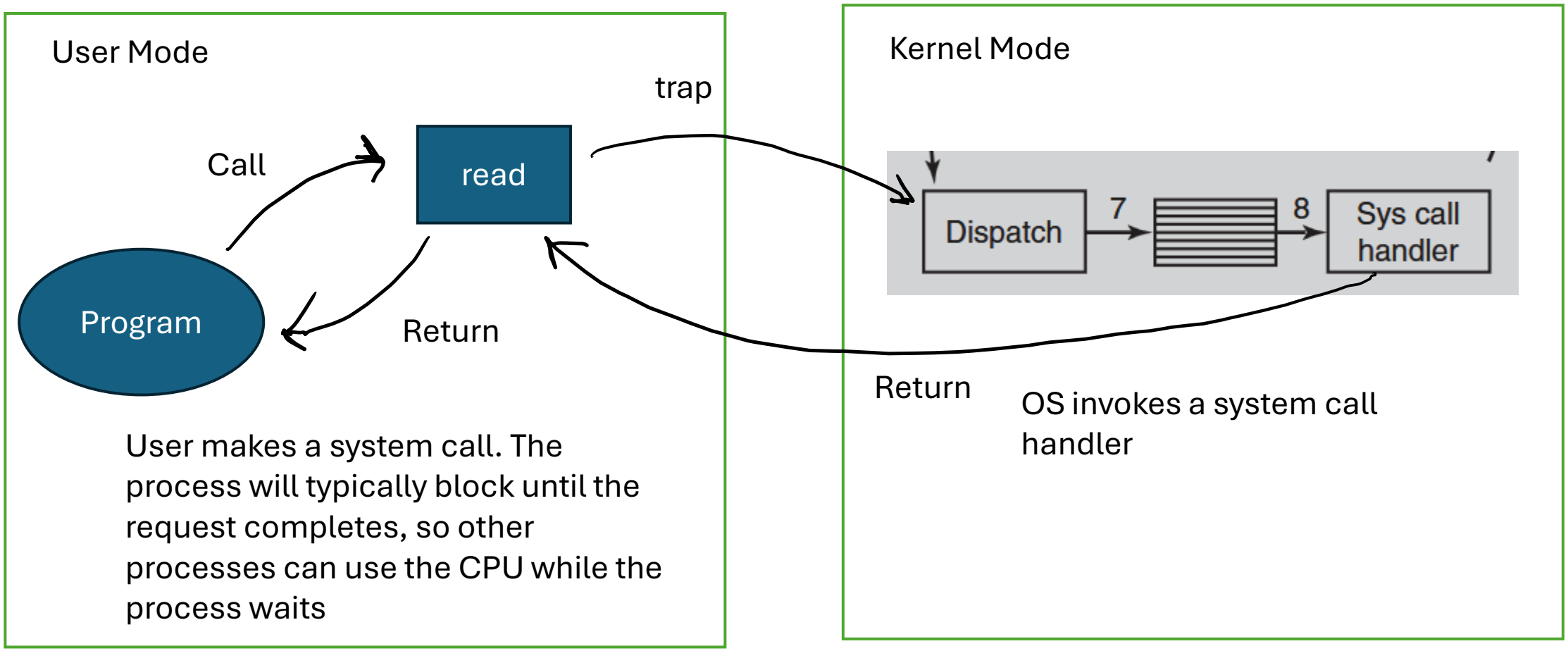
…and more!

System Calls and Error Handling

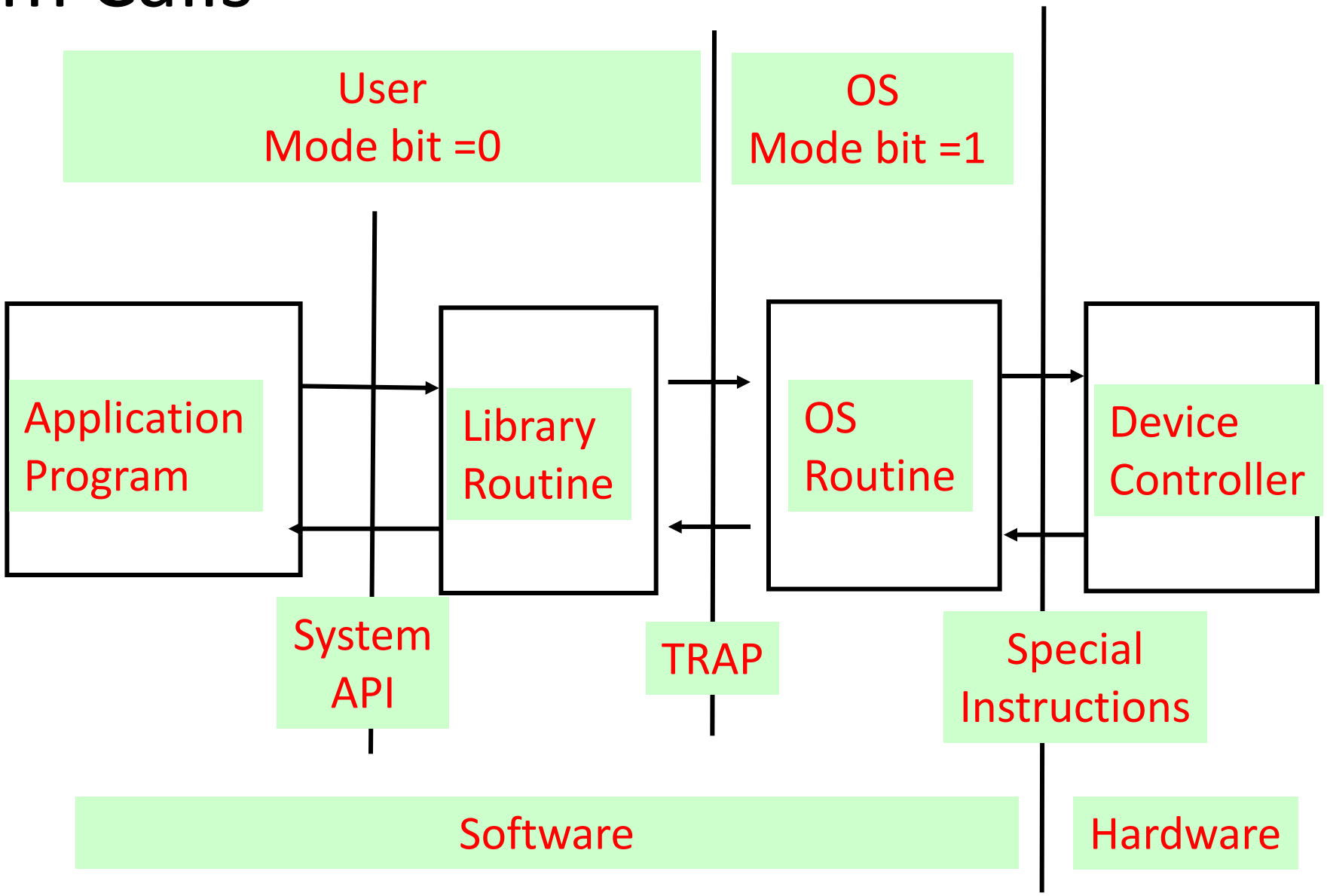
- All system calls return –1 if failed
- **`errno`** – the global variable that holds the integer error code for the last system call
- **`perror(const char *str)`** – a library function which describes the last system call error

- Every process has **`errno`** initialized to 0
- A successful system call never affects **`errno`**
- A failed system call always overwrites **`errno`**
- **`#include <sys/errno.h>`**

System Call Example: read



System Calls



Traps and Interrupts

OS is an interrupt-driven system

A process issues a **trap** to ask the OS to do something

The hardware issues an **interrupt** to the OS

Ex: Network msg arrives

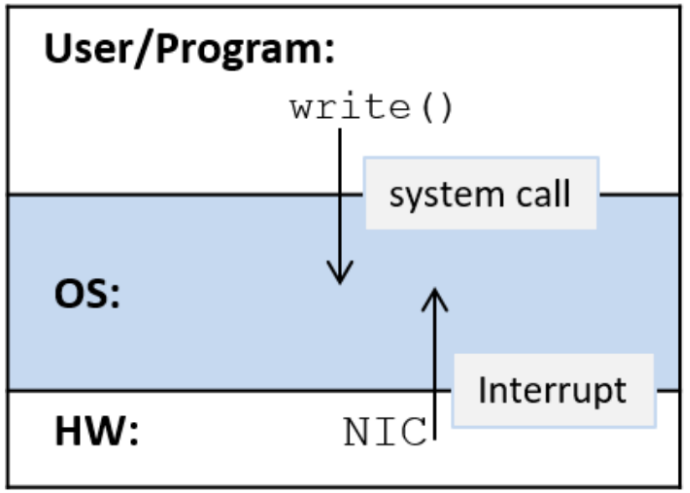


Figure 2. In an interrupt-driven system, user-level programs make system calls, and hardware devices issue interrupts to initiate OS actions.

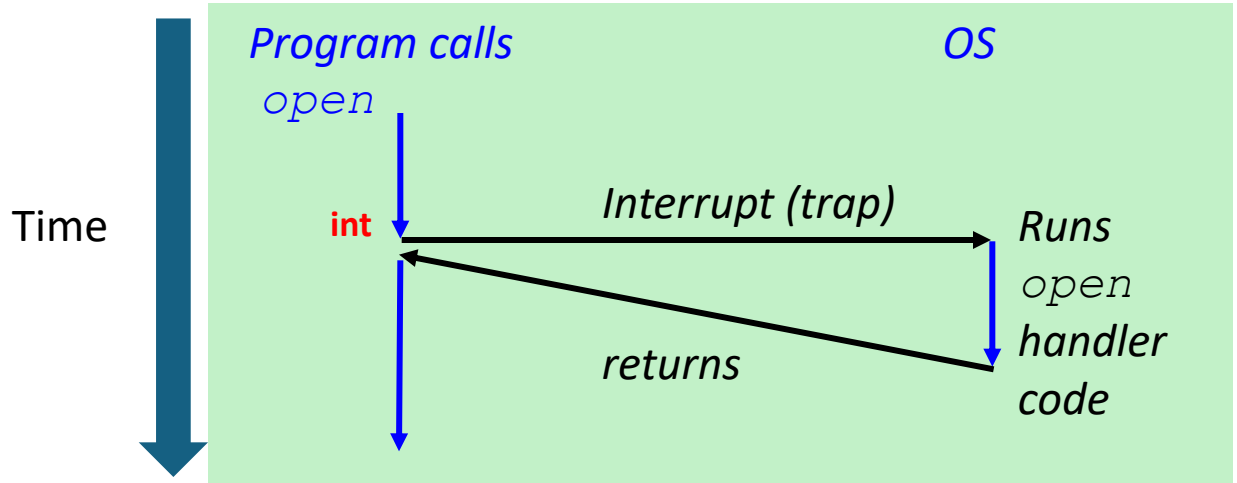
Dive into systems

Software Interrupt (trap)

Traps are implemented as interrupts to the OS that trigger an OS **trap handler** to run

- Example: `int fd = open(filename,options)`

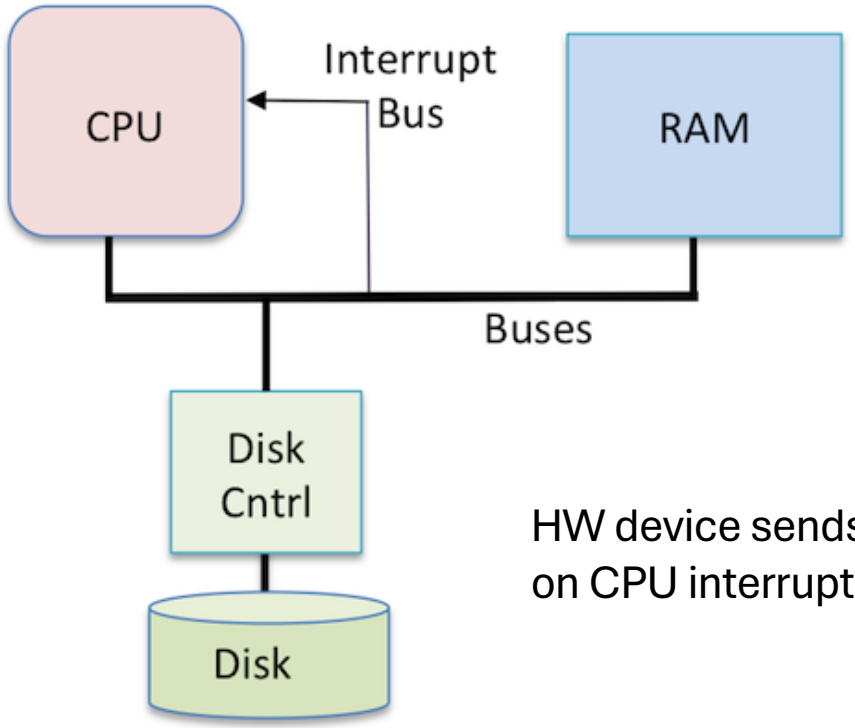
```
0804d070 <__libc_open>:
 . . .
0804d082: int  $0x80    # interrupt 0x80 is trap
```



Hardware Interrupt

Interrupts are implemented as **signals** on the **interrupt bus**

The CPU responds to the interrupt by running an **interrupt handler**, e.g. code configured to execute in response



HW device sends signal on CPU interrupt bus

Booting

The OS "pulls itself up by its bootstraps", or ***boots*** itself on the computer – Dive into systems

BIOS (Basic Input/Output System) and **UEFI** (Unified Extensible Firmware Interface) start the OS

- BIOS/UEFI are examples of **firmware**, or software stored on nonvolatile memory (e.g. persistent memory even without power) in hardware

Example: When a UNIX system boots, it loads the process *init*. All other processes are children of this root process

Environment

The OS maintains a set of **environment variables** that configure the system

Assignments of the form name = value

Example: PATH holds a list of directories to search for executables

```
main(int argc, char* argv[], char* envp[])
```