Systems programming and OS interface

CS503: Operating systems, Spring 2019

Pedro Fonseca
Department of Computer Science
Purdue University

Last lecture

- Architecture
    - Typical computer
    - Processor model (registers, execution model)
    - Memory: slow; single instruction can cause several accesses
        - Memory hierarchy
    - I/O: Devices, controllers, device drivers, communication
    - Hardware semantics

Last lecture

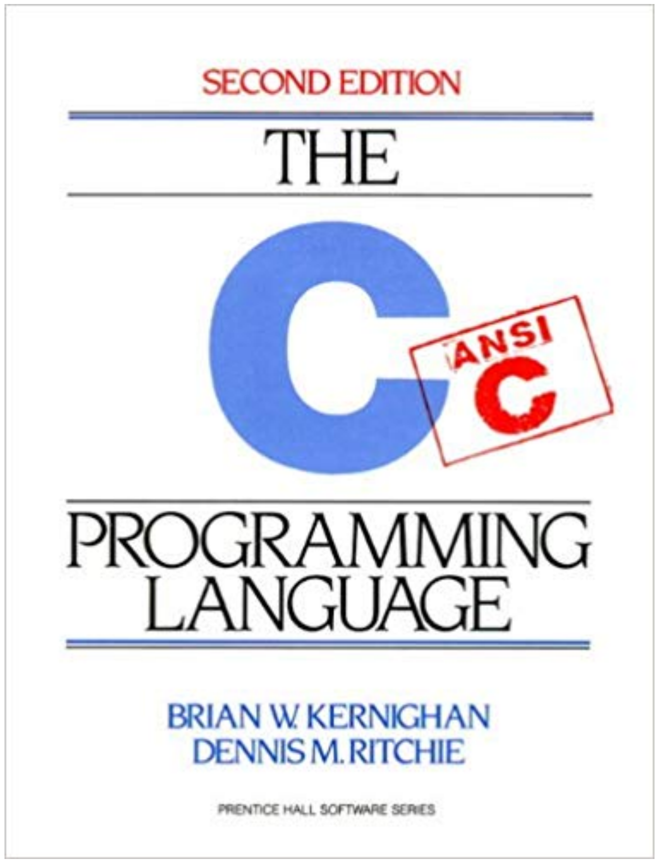
- Program structure
  - Address space of a program is an array
  - Memory sections
  - Building a program:
    - Preprocessor, compiler, assembler, linker
  - Linker: links different object files
    - static: before run-time (static libraries)
    - dynamic: during run-time (dynamic libraries)

Goals for today

- Systems programing:
  - Some pitfalls, tips and tricks
  - Where to learn more about C?
- The OS interface

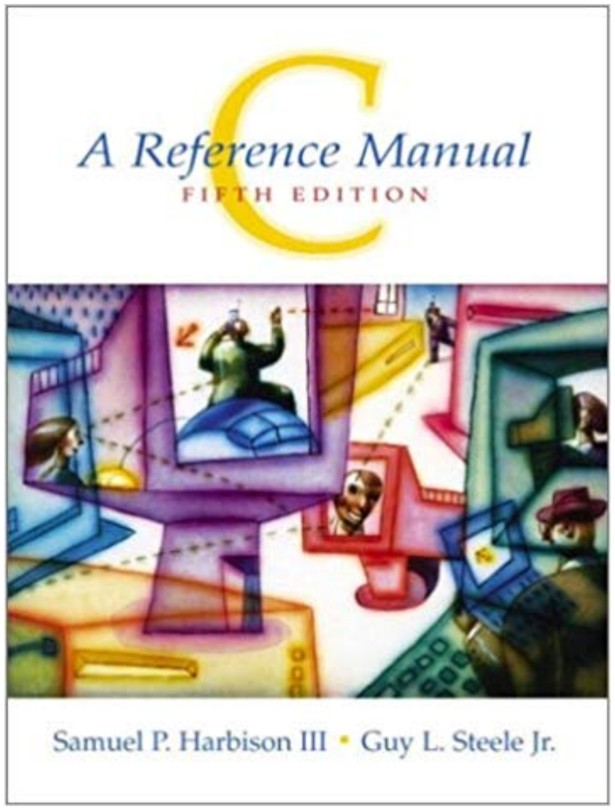
Systems programming

C resources: “the bible for C”



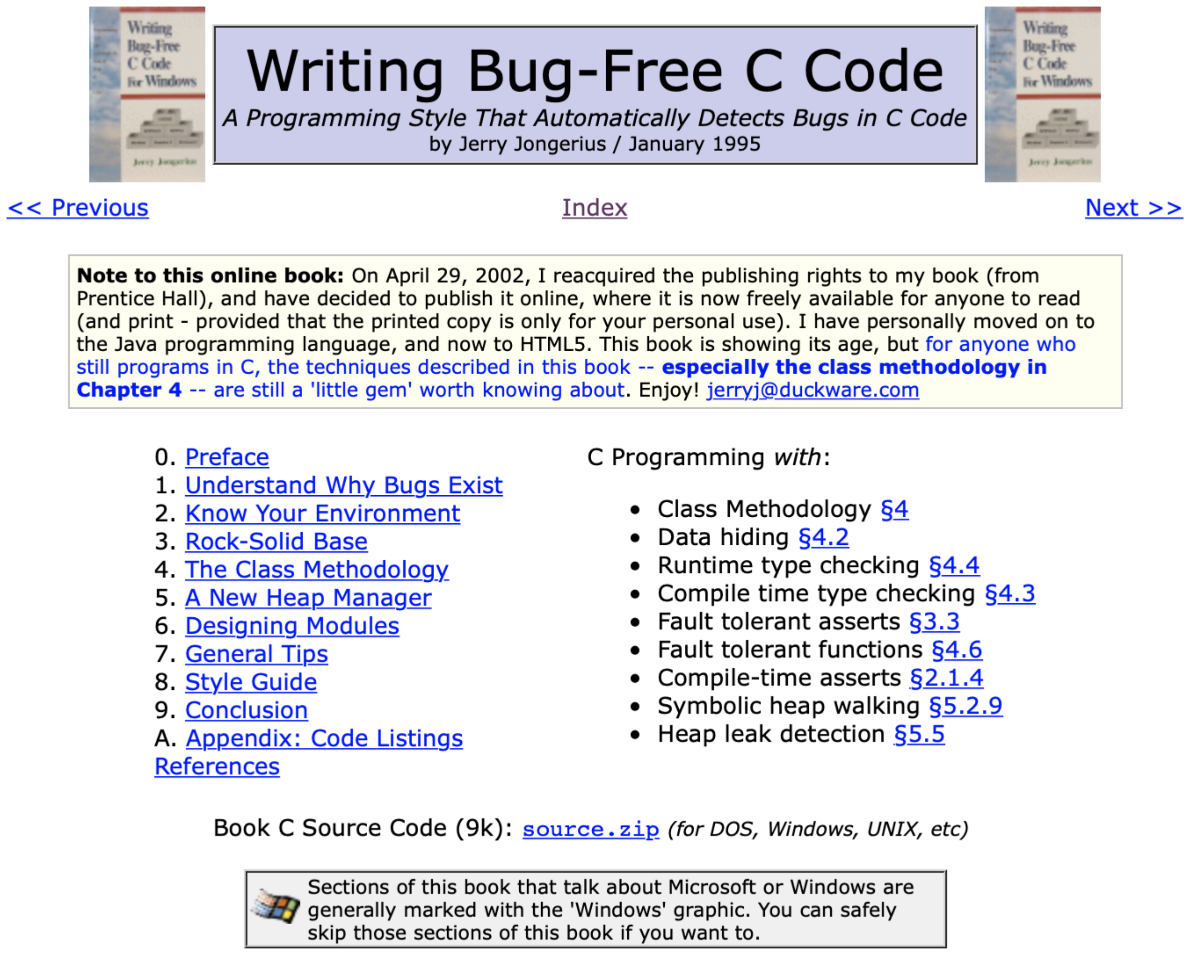
C Programming Language, 2nd Edition 2nd Edition
by Brian W. Kernighan, Dennis M. Ritchie

C resources



C: A Reference Manual, 5th Edition 5th Edition
by Samuel P. Harbison, Guy L. Steele Jr.

C resources



Writing Bug-Free C Code

A Programming Style That Automatically Detects Bugs in C Code

by Jerry Jongerius / January 1995

<< Previous Index Next >>

Note to this online book: On April 29, 2002, I reacquired the publishing rights to my book (from Prentice Hall), and have decided to publish it online, where it is now freely available for anyone to read (and print - provided that the printed copy is only for your personal use). I have personally moved on to the Java programming language, and now to HTML5. This book is showing its age, but for anyone who still programs in C, the techniques described in this book -- **especially the class methodology in Chapter 4** -- are still a 'little gem' worth knowing about. Enjoy! jerryj@duckware.com

C Programming *with*:

- Class Methodology §4
- Data hiding §4.2
- Runtime type checking §4.4
- Compile time type checking §4.3
- Fault tolerant asserts §3.3
- Fault tolerant functions §4.6
- Compile-time asserts §2.1.4
- Symbolic heap walking §5.2.9
- Heap leak detection §5.5

Book C Source Code (9k): `source.zip` *(for DOS, Windows, UNIX, etc)*

Sections of this book that talk about Microsoft or Windows are generally marked with the 'Windows' graphic. You can safely skip those sections of this book if you want to.

https://www.duckware.com/bugfreec/index.html

C basics (pre-requisites)

- Preprocessor
- Types
- Evaluation of expressions, operator precedence
- Pointers, arrays, addresses
- Wrong parameters to printf
- Structures
- Scope and lifetime of variables
- Other…

Do you know enough C for CS503?

- Example 1: Write a C program to input elements in an array and sort array using pointers: sort the array in ascending or descending order using function pointers
- Example 2: Implement a double linked list: implement append(), delete(), and find()
- Example 3: Implement a memory allocator: implement the malloc() and free() functions using a statically allocated buffer

If you can not solve this type of exercises, you do not satisfy the pre-requisites

Option 1: Work very, very hard during this semester to learn it

Option 2: Learn C now, take CS503 next semester

Some tips on C and systems programming

Some typical C mistakes

- Wrong assumptions about precedence
- Assumptions about evaluation order
- Confusing pointers
- Of-by-one errors when iterating arrays
- Incorrect casting
- = vs. ==
- Preprocessor
- Other?

Preprocessor

- Typical code in a header file:

```
#ifndef _LIMITS_H_
#define _LIMITS_H_
<header content>
#endif
```

- Prevents circular inclusions
  - Still need to ensure that the compiler can resolve circular declarations

Preprocessor

- Preprocessor macros are usually capitalized
- A simple multiplication macro?

```
#define MULTIPLY(x, y) x * y

int main(){
   int result = MULTIPLY(2, 4);
   int result2 = MULTIPLY(1+1, 4);
   printf("%d %d\n", result, result2);
}
```

- What's wrong with it?

Preprocessor: after expansion

- Preprocessor macros are usually capitalized
- A simple multiplication macro?

```
#define MULTIPLY(x, y) x * y

int main(){
   int result = 2 * 4;
   int result2 = 1+1 * 4;
   printf(“%d %d\n”, result, result2);
}
```

- What’s wrong with it?

Expansion produces incorrect results…

Preprocessor

- A simple squaring macro?

```
#define SQUARE(x) x * x

int main(){
   int result = SQUARE(2);
    printf(“%d\n”, result);
}
```

- What’s the problem with it?

Preprocessor

- A simple squaring macro?

```
#define SQUARE(x) x * x

int main(){
   int result = SQUARE(2);
    printf(“%d\n”, result);
}
```

- What’s the problem with it?

No problem yet…

Preprocessor

- A simple squaring macro?

```
#define SQUARE(x) x * x

int value = 0;

int inc(){
   value++;
   return value;
}

int main(){
   int result = SQUARE(inc());
   printf(“%d\n”, result);
}
```

Preprocessor

- A simple squaring macro?

```
#define SQUARE(x) x * x

int value = 0;

int inc(){
   value++;
   return value;
}

int main(){
   int result = inc() * inc();
   printf("%d\n", result);
}
```

SQUARE invokes inc() twice
AND
inc() has side-effects

Preprocessor

- Using macros can be tricky!
  - They don’t behave always as functions
- Good practice:
  - Name them according to the convention because they behave differently from normal functions
  - Shield macro definitions
  - Don’t use functions with side-effects when passing arguments to macros
- Bottom line: be careful with macros

```c
#include <stdio.h>

int main()
{
    int num1, num2, sum;
    int *ptr1, *ptr2;
    ptr1 = &num1;
    ptr2 = &num2;
    printf("Enter any two numbers: ");
    scanf("%d%d", ptr1, ptr2);
    v = *ptr1 + *ptr2;
    printf("%d", v);
}
```

Good practice

- Always, always, always indent consistently (and properly)
- Alway use braces, even for single statement blocks

  -
    ```
    if (XXXX){
        printf("hello\n");
    }
    ```
- **If….else** is better than **if (x) …. if (!x)….**

Good practice

- Compile with all compiler warnings enabled (e.g., “-Wall”)
  - And… don’t dismiss any warning
- The point where you seen something strange is not necessarily where the bug is
  - Specially if there is memory corruption
  - E.g., go inside a function, write data outside of a buffer, do a lot of arithmetic, return -> crash

Good practice

- Test frequently
- Don’t keep adding code if you know something is wrong
- Don’t add code you don’t understand
    - Or do random “fixes”
- Use a good IDE with source code navigation

More C and systems programming tips

- “C programming tips”
  http://www.pgbovine.net/c-programming-tips.htm

- “50 tips for improving your software development game”
  https://techbeacon.com/50-tips-improving-your-software-development-game

- “C Programming Tips”
  https://hownot2code.com/2016/11/29/c-programming-tips/

The OS interface

Kernel and User mode

- Kernel mode
  - When the CPU runs in this mode:
    - It can execute any machine instructions
      - MOV DBn / MOV CRn (move to debug/control register)
      - HLT (halt the CPU)
      - LMSW, SMSW (load/store Machine Status Word register)
      - IN/OUT
    - It can access/modify any memory location
    - It can access/modify any register in the CPU and any device
  - The OS kernel instructions run in kernel mode

Kernel and User mode

- User mode:
  - When the CPU runs in this mode:
    - The CPU can use a limited set of instructions
    - The CPU can only modify sections of memory assigned to the process running the program
    - The CPU can only access a subset of the registers in the CPU and it generally cannot access registers in devices
  - User processes run in user mode

Kernel and User mode

- When the OS boots, it start in kernel mode
- In kernel mode the OS sets up all the interrupt vectors and initializes all the devices
- Then it start the first process and switches to user mode
- In user mode it run all the background system processes (daemons or services)
- The it runs the user shell or windows manager

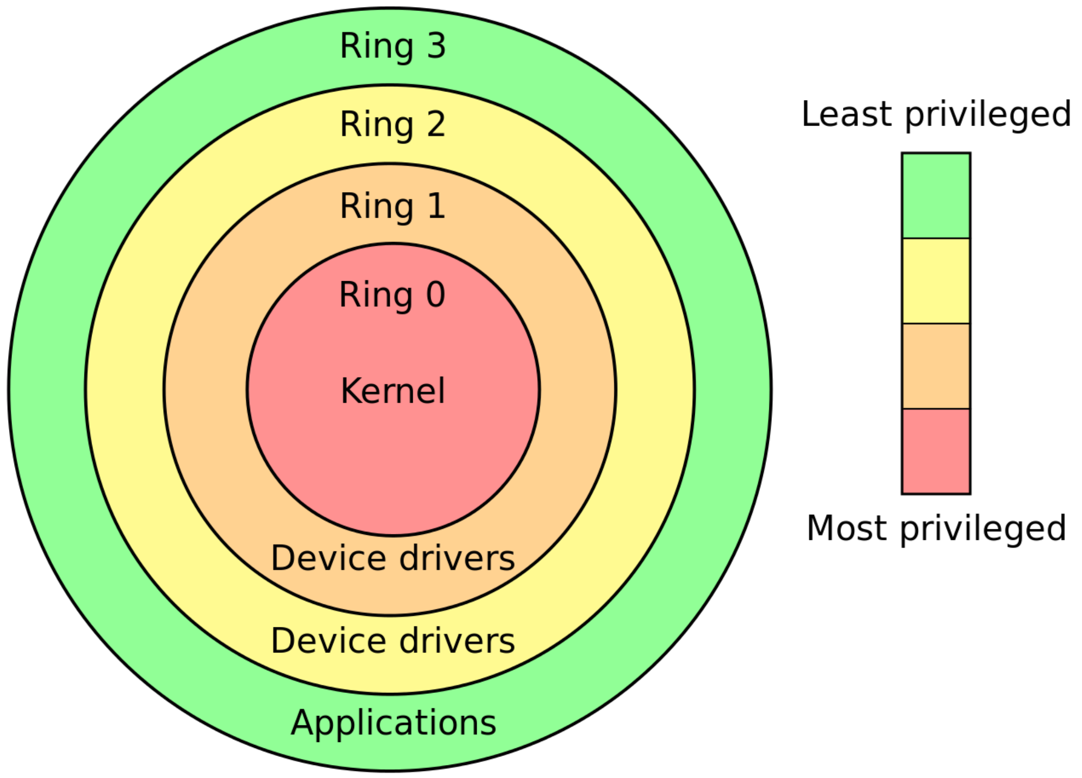
Kernel and User mode

- User programs run in user mode
- Programs switch to kernel mode to request OS services (through the **system call** interface)
- Also user programs switch to kernel mode when an interrupt is raised
- They switch back to user mode when interrupt returns
- The interrupt handling code (part of OS) is executed in kernel mode
- The interrupt vector can be modified only in kernel mode
- Most of the CPU time is spent in user mode
  - There is overhead in switching between user mode to kernel mode

Kernel and User mode

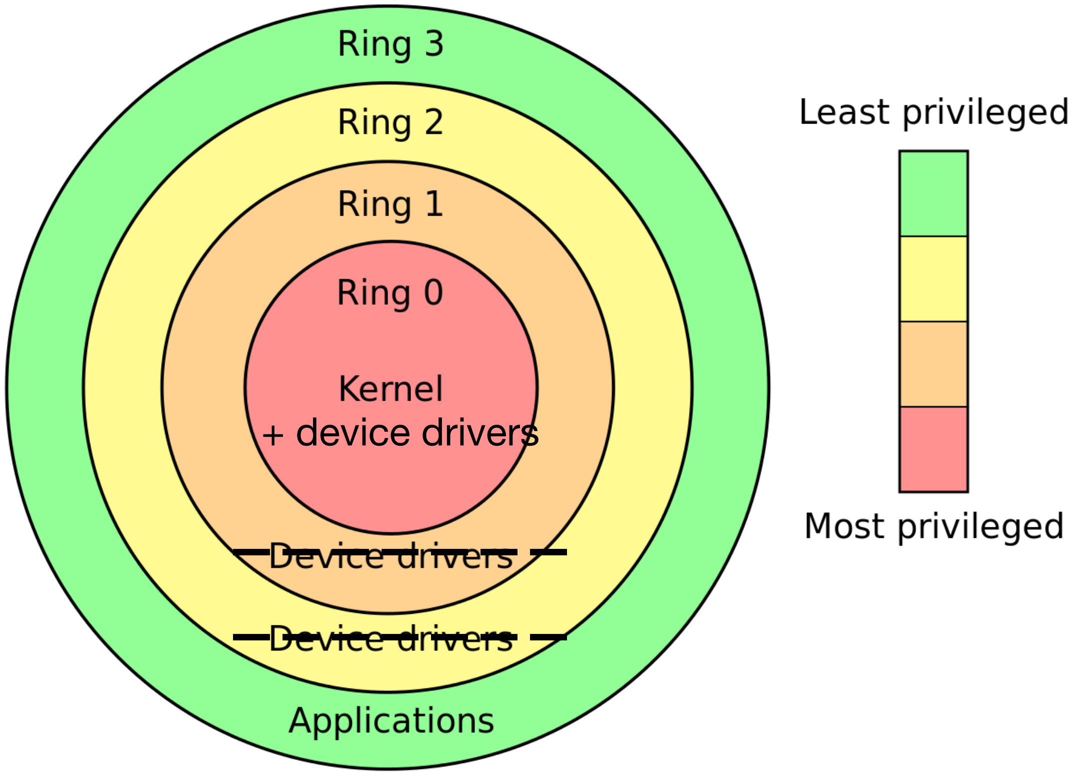
- How does the HW know if code is to run in kernel mode or user mode?

X86 protection mechanisms



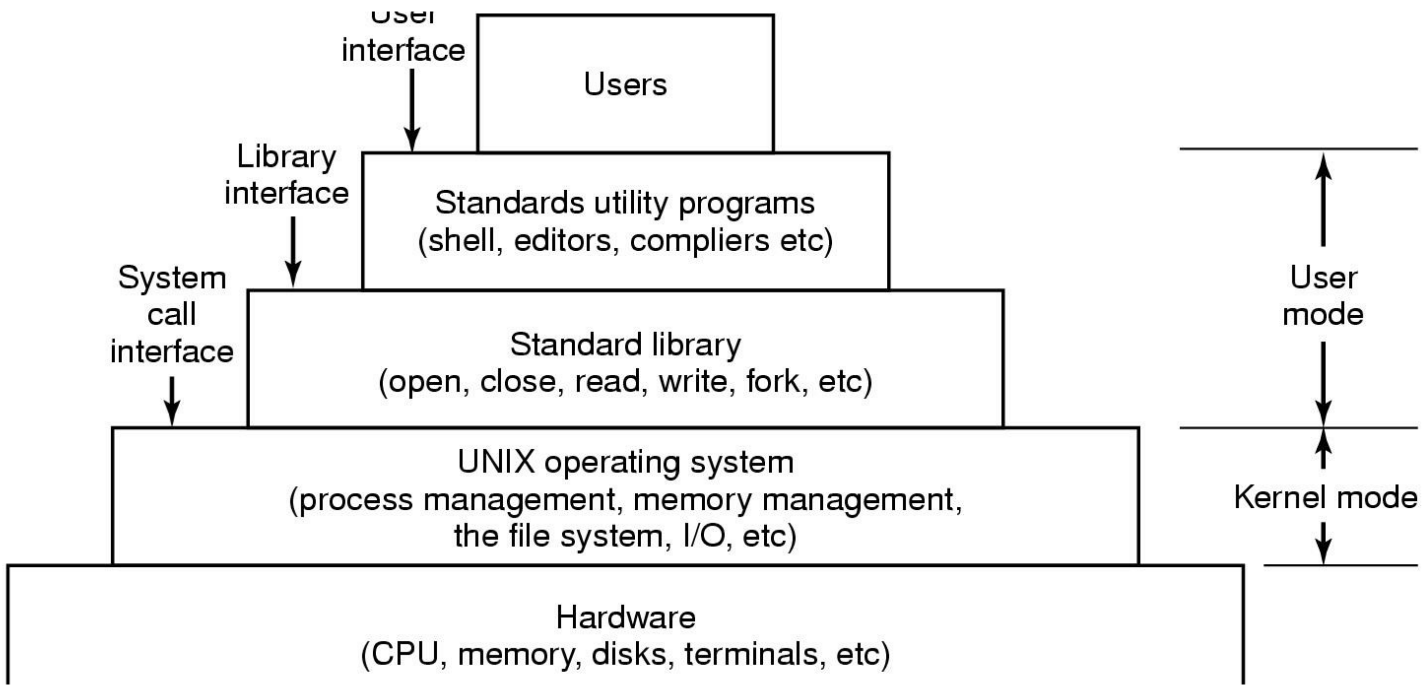
The idea

X86 protection mechanisms



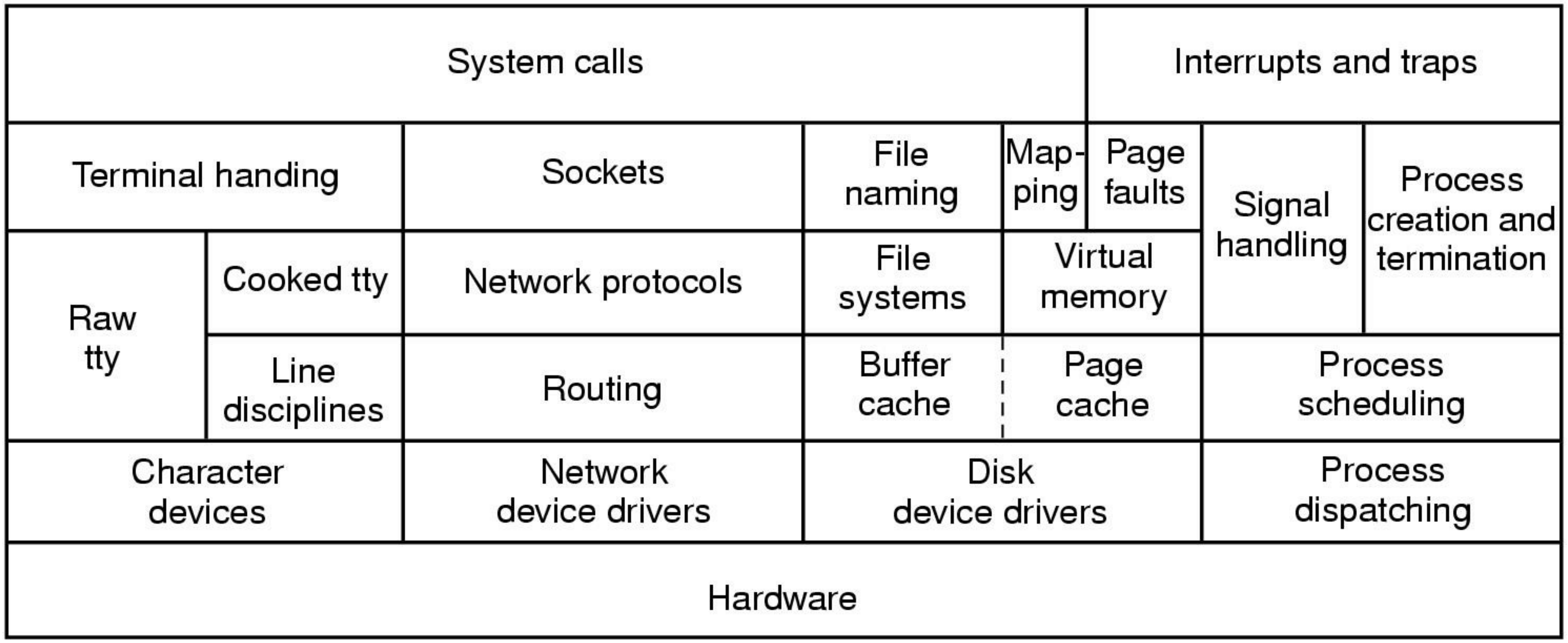
In practice

UNIX interfaces


User interface
Users
Library interface
Standards utility programs
(shell, editors, compliers etc)
System call interface
Standard library
(open, close, read, write, fork, etc)
UNIX operating system
(process management, memory management,
the file system, I/O, etc)
Hardware
(CPU, memory, disks, terminals, etc)
User mode
Kernel mode

The layers of a UNIX system

UNIX Kernel



System calls
Interrupts and traps
Terminal handing
Sockets
File naming
Map-ping
Page faults
Signal handling
Process creation and termination
Raw tty
Cooked tty
Line disciplines
Network protocols
Routing
File systems
Virtual memory
Buffer cache
Page cache
Process scheduling
Character devices
Network device drivers
Disk device drivers
Process dispatching
Hardware

Interrupts

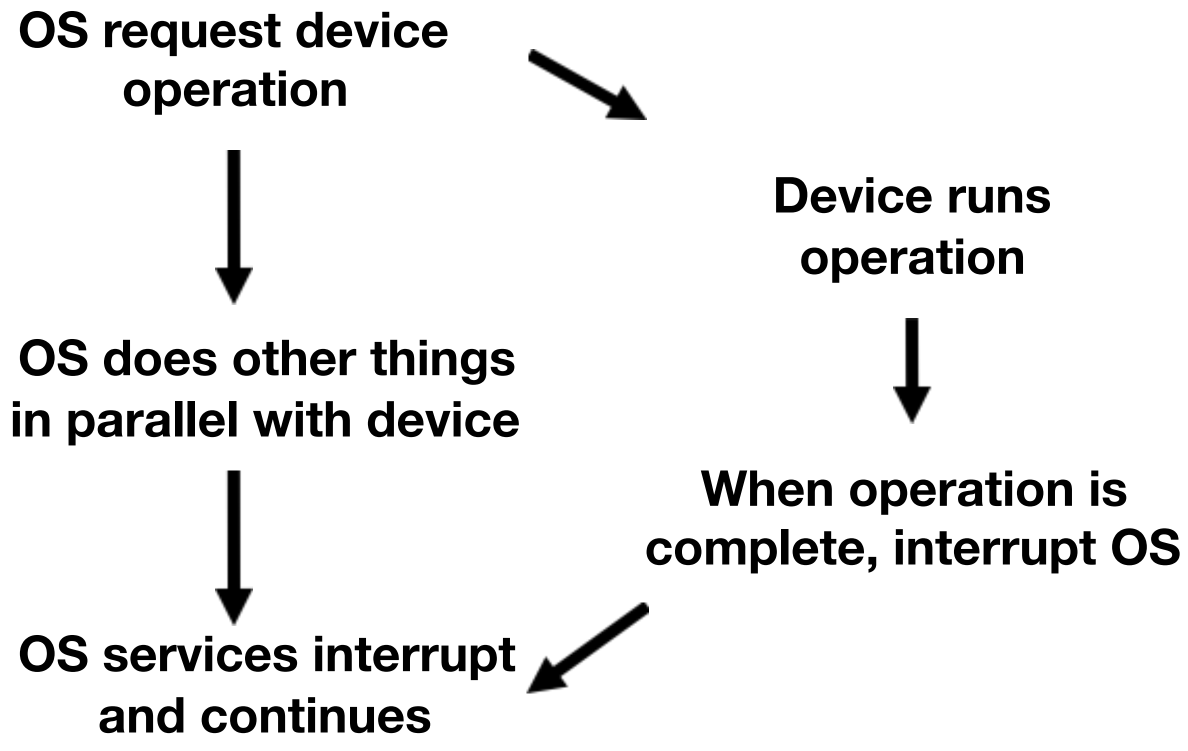
- An interrupt is an event that requires immediate attention. In hardware, a device set the interrupt line to high
- When an interrupt is received, the CPU stops whatever it is doing and it jumps to the **interrupt handler** for the specific interrupt
- After executing the handler, it return to the same place where the interrupt happened and the program continues
- Examples: mouse, key press, network packet

Steps to handle an interrupt

- THe CPU saves the program counter (PC) and registers in the execution stack
- CPU looks up the corresponding interrupt handler in the interrupt vector
- CPU jumps to the interrupt handler and runs it
- CPU restores the registers and returns back to the place in a program that was previously interrupted. The program continues execution as if nothing happened.
- In some cases it reties the instruction that was interrupted (e.g., virtual memory page fault handlers)

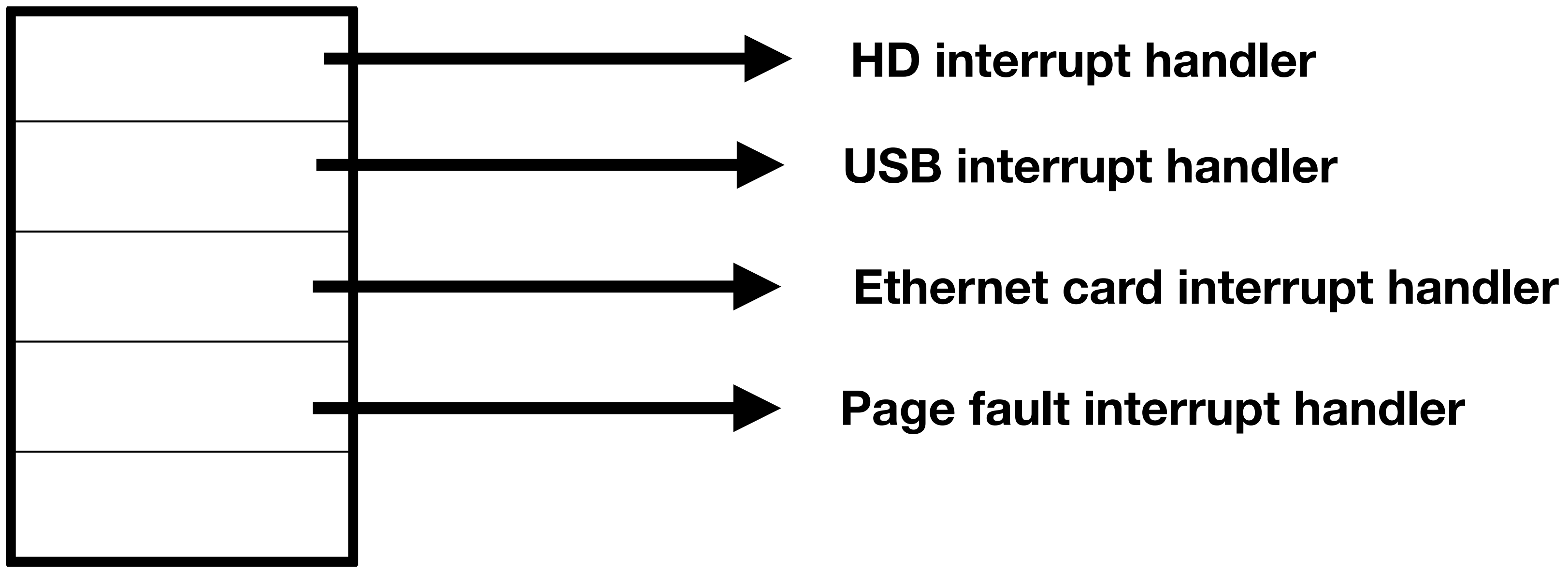
Running with interrupts

- Interrupts allow CPU and device to run in parallel without waiting for each other


OS request device operation
Device runs operation
OS does other things in parallel with device
When operation is complete, interrupt OS
OS services interrupt and continues

Interrupt vector

- Array of pointers that point to different interrupt handlers for each type of interrupt



Interrups and kernel mode

- Interrupt must be handled in kernel mode
  - Interrupt handler must read device/CPU registers and execute instruction that are only available in kernel mode
- Interrupt vector can be modified only in kernel mode (security)
- Interrupt vector initialized during boot time; modified when drivers added to the system

Types of interrupts

- **Device interrupts** generated by the devices when a request is complete or an event that requires CPU attention happens
    - The mouse is moved
    - A key is pressed
    - ….

Types of interrupts

- **Math exceptions** generated by the CPU when there is a math error
    - Divide by zero
- **Page faults** generated by the Memory Management Unit (MMU) that converts virtual memory addresses to physical memory addresses
    - Invalid address: interrupt sends a SEGV signal to the process
    - Access to a valid address but there is not a page in memory; this is used by the kernel to load a page from disk
    - Invalid permission (i.e., writing on a read only page) causes a SEGV signal to the process

Types of interrupts

- Software interrupts generated by software with a special instruction (INT). This is one way for a program running in user mode to request operating system services
  - int 0x80 (system call number in EAX)
  - int 0x3 (debug interrupt)

System calls

- System calls is the way user programs request services from the OS
- **System calls** use **software interrupts**
- Examples of system calls are:
  - open(filename, mode)
  - read(file, buffer, size)
  - write(file, buffer, size)
  - fork()
  - execve(cmd, args)
- System calls in the API of the OS from the user program's point of view, see /usr/include/sys/syscall.h (Linux)

Why software interrupts instead of function calls?

- Software interrupts switch into kernel mode
- OS services need to run in kernel mode because:
  - They need privilege instructions
  - Access to devices and kernel data structures
  - They need to enforce the security in kernel mode

System calls

- Only operations that need to be executed by the OS in kernel mode are part of the system calls

- Functions like sin(x) and cos(x) are not system calls

- Some C functions like printf(s) run mainly in user mode…but call **write()** to flush buffers

- **malloc**(size) runs mostly in user mode but calls **sbrk**() or **mmap**() to extend the heap

System calls

- Libc (the C library) provides wrappers for the system calls that eventually generate the system calls

System calls and interrupts example

- 1. The user program calls the **write(fd, buff, n)** system call to write to disk
- 2. The write wrapper in libc generates a software interrupt for the system call
- 3. The OS checks the arguments: it verifies that:
  - fd is a file descriptor, for a file opened in write mode
  - it checks that the range [buff, buff + n - 1] is a valid memory range
  - if any check fails returns -1 and sets errno to the error value

System calls and interrupts example

- 4. The OS tells the hard drive to write the buffer in [buff, buff+n] to disk to the file specified by fd
  - More on this later in the course
- 5. The OS puts the current process in wait state, until the disk operation is complete. Meanwhile, the OS switches to another process
- 6. Eventually, the disk completes the write operation and generates an interrupt
- 7. The interrupt handler puts the process calling write into ready state so this process will be schedule by the OS in the next chance

Summary

- C resources and systems programming
    - Brush up your C skills
    - Required for labs, exams and in-class quizzes
- OS interface:
    - Privilege modes and their use cases
        - Unix interfaces and kernel/user mode
    - Interrupts and interrupt handling
    - System call implementation