Introduction to Operating Systems

Mythili Vutukuru

CSE, IIT Bombay

What is a computer system?

- Software + hardware to run user applications and programs, to accomplish some tasks
- Components:
    - Hardware: CPU, memory, I/O devices, …
    - System software: Operating System (OS), …
    - User software: user applications (browser, email client, games, ...)
- Real-life computer systems (e.g., an e-commerce website) have multiple interconnected computers, each running one or more applications that communicate with each other

Real life computer systems are complex

- Real life systems are complex
  - Multiple interacting components and sub-systems
  - Each component independently developed, but have to work together for a common purpose
  - Prone to failures, bugs, crashes
- But still, we expect:
  - The system always does what it is supposed to do … (functional correctness)
  - Quickly, efficiently, for a large number of users, lots of data … (performance)
  - Even when it is overloaded or when failures occur … (reliability)

Why study operating systems?

- Knowledge of hardware (architecture) + system software (OS), and how user programs interact with these lower layers, is essential to writing “good” (high performance, reliable) user programs
  - What exactly happens when you run a user program?
  - How to make your program run faster and more efficiently?
  - Why do programs crash and how to avoid it?
  - How to make your programs more secure?
- OS expertise is one of the most important building blocks when building high performance, robust, complex real life systems

Beyond OS to real systems

- Architecture + OS: Basic foundation to understand how a user program runs on a single machine
- Networking: How programs talk to each other across machines
- Databases and data storage: How applications store data efficiently and reliably in one or more machines
- Performance engineering: how to make programs run faster
- Distributed systems: How multiple applications across multiple machines work together to perform a useful task reliably
- Virtualization, cloud computing, security, …

What is an operating system?

- Middleware between user programs and system hardware
  - Not user application software but system software
  - Example: Linux, Windows, MacOS
- Manages computer hardware: CPU, main memory, I/O devices (hard disk, network card, mouse, keyboard etc.)
  - User applications do not have to worry about low-level hardware details
- Operating system has kernel + other extra useful software
  - Kernel = the core part of the operating system

What is an operating system?

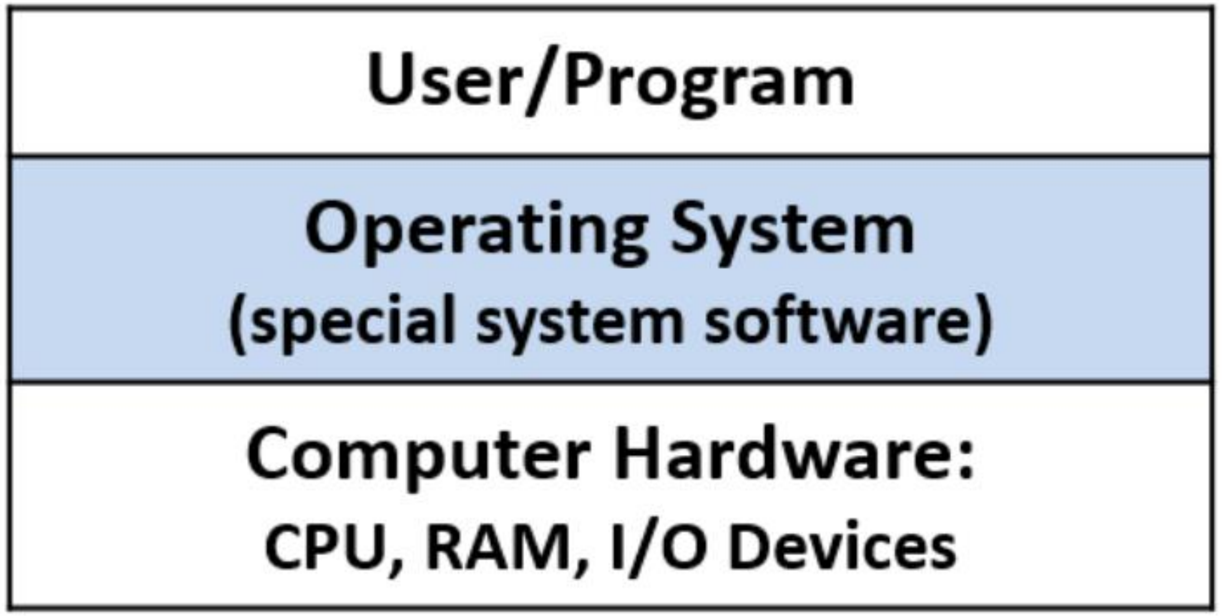

Figure 1. The OS is special system software between the user and the hardware. It manages the computer's hardware and implements abstractions to make the hardware easier to use.

Image credit: Dive into Systems

History of operating systems

- Started out as a library to provide common functionality to access hardware, invoked via function calls from user program
  - Convenient to use OS instead of each user writing code to manage hardware
  - Centralized management of hardware resources is more efficient
- Later, computers evolved from running a single program to multiple processes concurrently
  - Multiple untrusted users must share same hardware
- So OS evolved to become trusted system software providing isolation between users, and protecting hardware
  - Multiple users are isolated and protected from each other
  - System hardware and software is protected from unauthorized access by users

Hardware organization

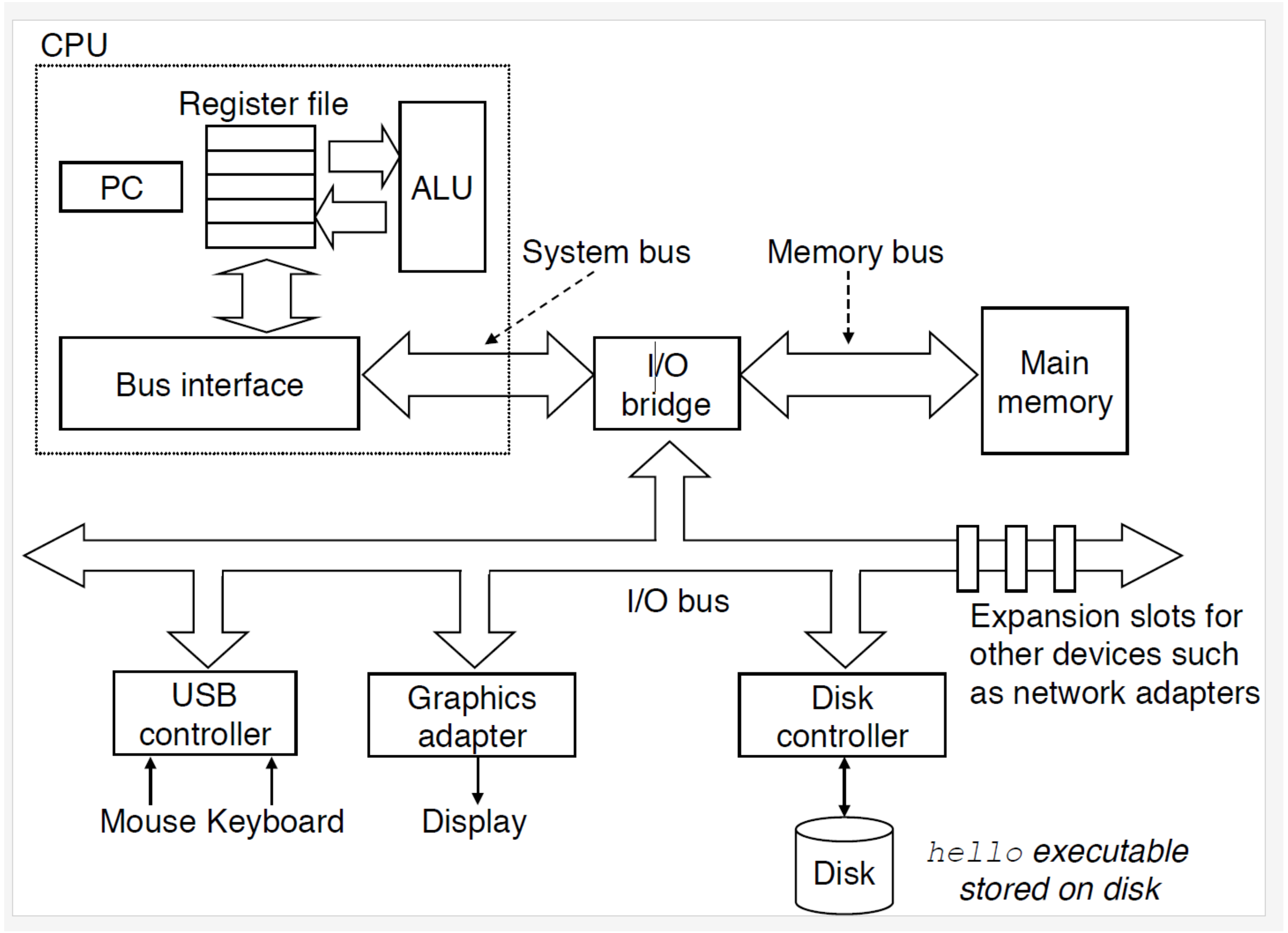

Image credit: CSAPP

What is a program?

- User program = code (instructions for CPU) + data
- Stored program concept
  - User programs stored in main memory or Random Access Memory (RAM)
  - Instructions/data occupy multiple contiguous bytes in memory
  - Memory is byte-addressable: data accessed via memory address / location / byte#
  - CPU fetches code/data from RAM using memory address, and executes instructions
- CPU runs processes = running programs
- Modern CPUs have multiple cores for parallel execution
  - Each core runs one process at a time each
  - Modern CPUs have hyper-threading, where each core can run more than one process also (OS treats hyper-threading cores also as multiple CPU cores)

CPU ISA

- Every CPU has
  - A set of instructions that the hardware can execute
  - A set of registers for temporary storage of data within the CPU
- High level language (C code) translated into CPU instructions by compiler
  - Can directly write machine code, but cumbersome
- Instructions and registers defined by ISA = Instruction Set Architecture
  - Specific to CPU manufacturer (e.g., Intel CPUs follow x86 ISA)
- Registers: special registers (specific purpose) or general purpose
  - Program counter (PC) is special register, has memory address of the next instruction to execute on the CPU
  - General purpose registers can be used for anything, e.g., operands in instructions
- Size of registers defined by architecture (32 bit / 64 bit)

CPU instructions

- Some common examples of CPU instructions
  - Load: copy content from memory location → register
  - Store: copy content from register → memory location
  - Arithmetic and logical operations like add: reg1 + reg2 → reg3, compare, ..
  - Jump: change value of PC
- Simple model of CPU
  - Each clock cycle, fetch instruction at PC, decode, access required data, execute, update PC, repeat
  - PC increments to next instruction, or jumps to some other value
- Many optimizations to this simple model
  - Pipelining: run multiple instructions concurrently in a pipeline
  - Many more in modern CPUs to optimize #instructions executed per clock cycle

Memory hierarchy

- Hierarchy of storage elements which store instructions and data
  - CPU registers (small number, accessed in <1 nanosec)
  - Multiple levels of CPU caches (few MB, 1-10 nanosec)
  - Main memory or RAM (few GB, ~100 nanosec)
  - Hard disk (few TB, ~1 millisec)
- Hard disk is non-volatile storage, rest are volatile
  - Hard disk stores files and other data persistently
- As you go down the hierarchy, memory access technology becomes cheaper, slower, less expensive
- CPU caches transparent to OS, managed by hardware
  - Software only accesses memory, doesn’t know if served from cache or DRAM

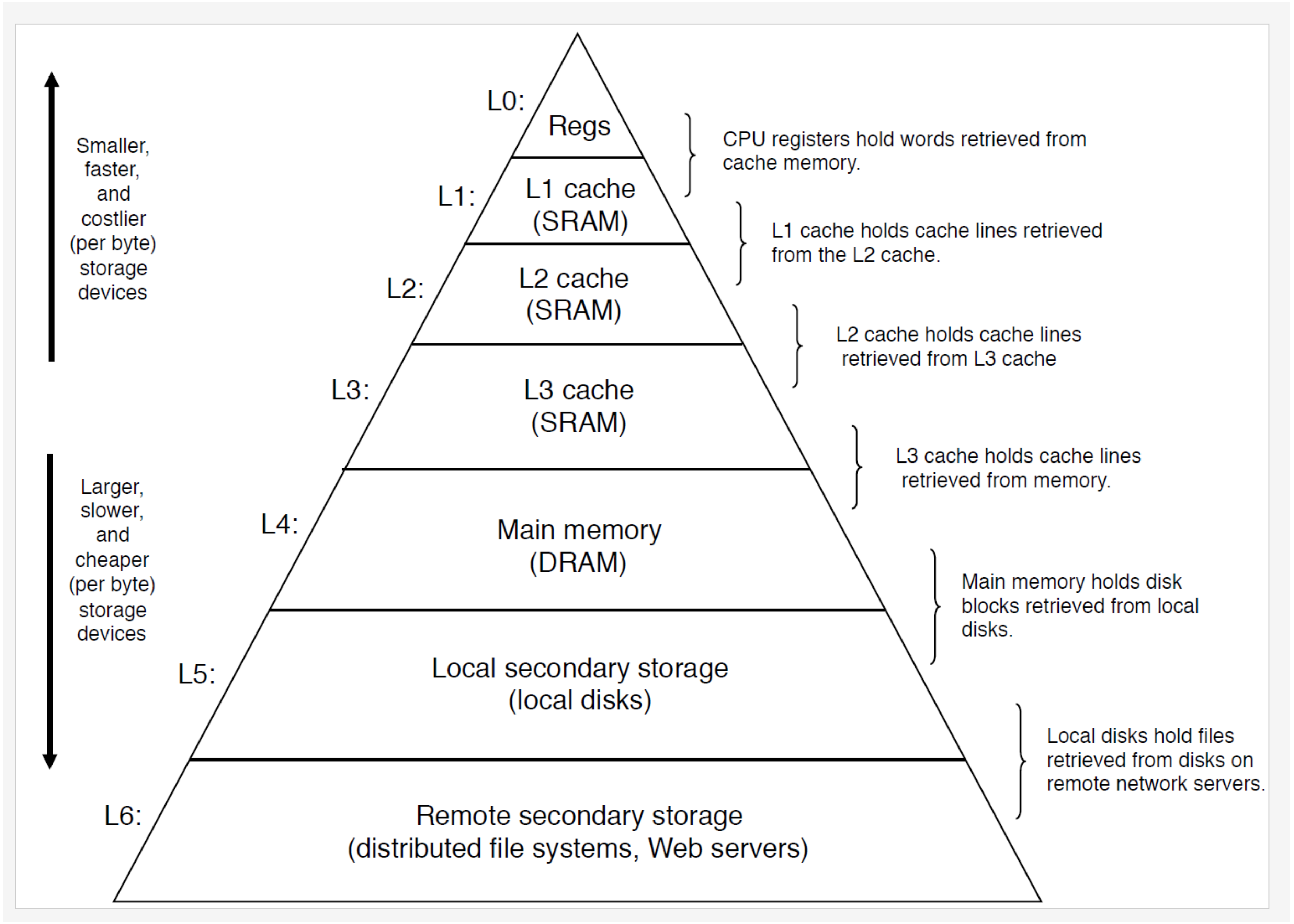
Smaller, faster, and costlier (per byte) storage devices
Larger, slower, and cheaper (per byte) storage devices
L0:
Regs
L1:
L1 cache (SRAM)
L2:
L2 cache (SRAM)
L3:
L3 cache (SRAM)
L4:
Main memory (DRAM)
L5:
Local secondary storage (local disks)
L6:
Remote secondary storage (distributed file systems, Web servers)
CPU registers hold words retrieved from cache memory.
L1 cache holds cache lines retrieved from the L2 cache.
L2 cache holds cache lines retrieved from L3 cache
L3 cache holds cache lines retrieved from memory.
Main memory holds disk blocks retrieved from local disks.
Local disks hold files retrieved from disks on remote network servers.

Image credit: CSAPP

Running a program

- What happens when you run a C program?
  - C code translated into executable by compiler

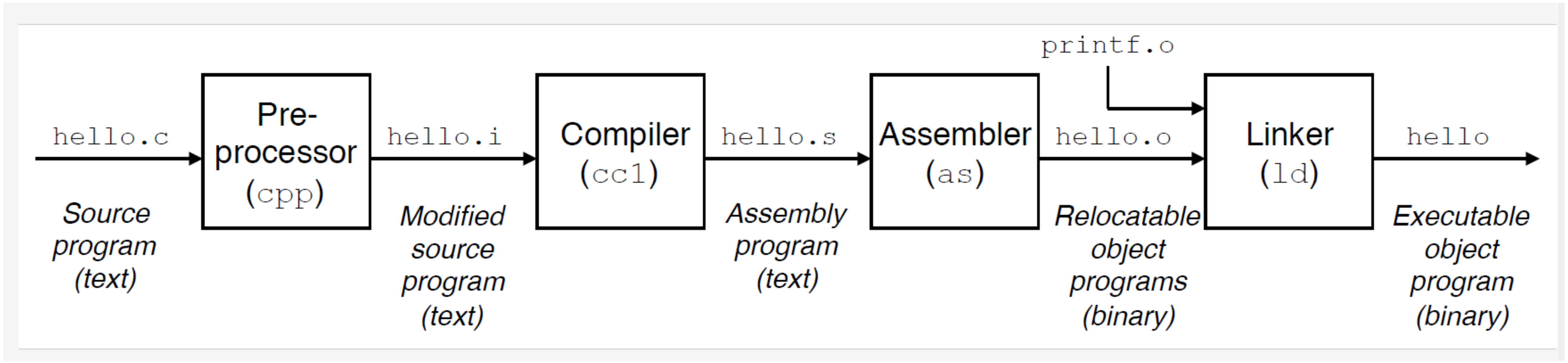

Image credit: CSAPP

Running a program

- What happens when you run a C program?
    - C code translated into executable by compiler
    - Executable file stored on hard disk (say, “a.out”)
    - When executable is run, a new process is created
    - Process allocated space in RAM to store code and data
    - CPU starts executing the instructions of the program
- When CPU is running a process, CPU registers contain the execution context of the process
    - PC points to instruction in the program, general purpose registers store data in the program, and so on

Role of OS in running a process

- Allocates memory for new process in RAM
  - Loads code, data from disk
- Initializes CPU context
  - PC points to first instruction
- Process starts to run
  - OS steps in as needed

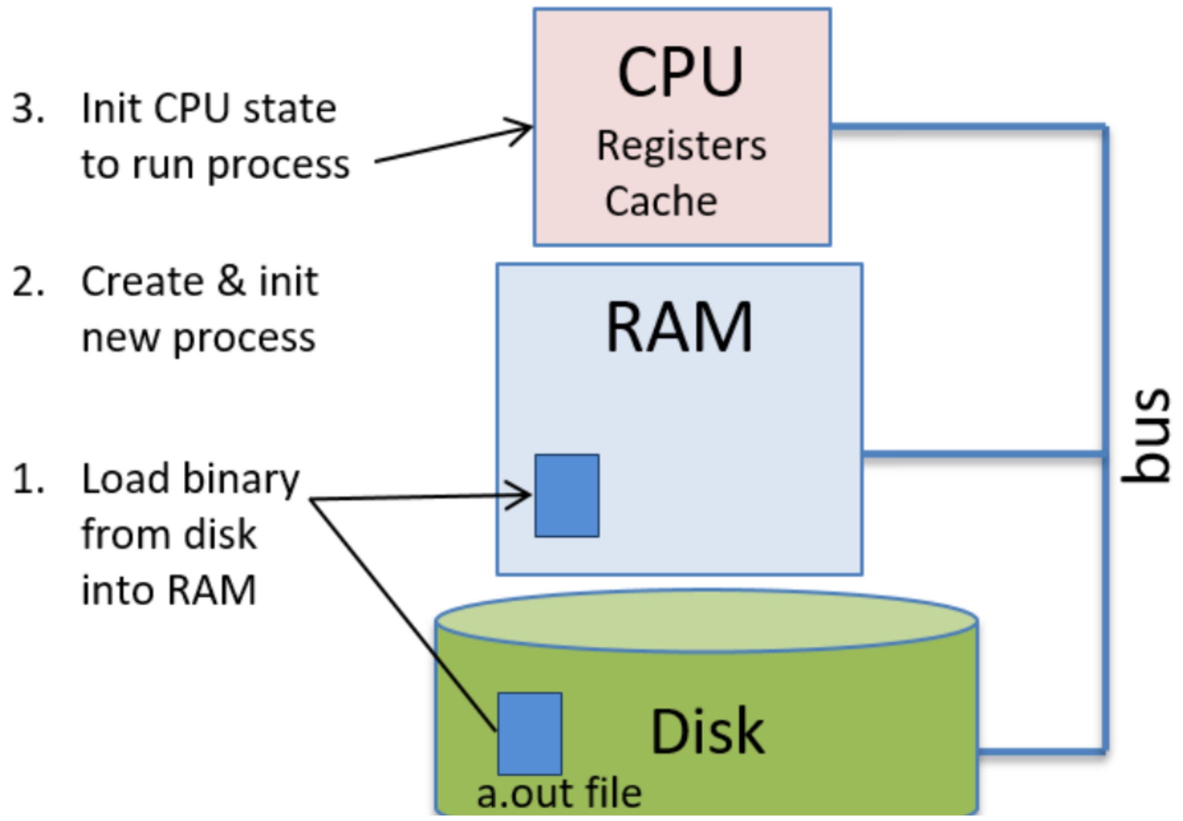
3. Init CPU state to run process
2. Create & init new process
1. Load binary from disk into RAM
CPU
Registers
Cache
RAM
Disk
a.out file
bus

Image credit: Dive into Systems

Concurrent execution

- CPU runs multiple programs concurrently
  - Run one process, switch to another, switch again, ...
- How to ensure correct concurrent execution?
  - Run user code of process A for some time
  - Pause A, save context of A, load context of B: context switching
  - Run user code of process B for some time
  - Pause B, save context of B, restore context of A, run A
- Every process thinks it is running alone on CPU
  - Saving and restoring context ensures process sees no disruption
- OS takes care of this switching across processes
  - OS virtualizes CPU across multiple processes
  - OS scheduler decides which process to run on which CPU at what time

Context switching

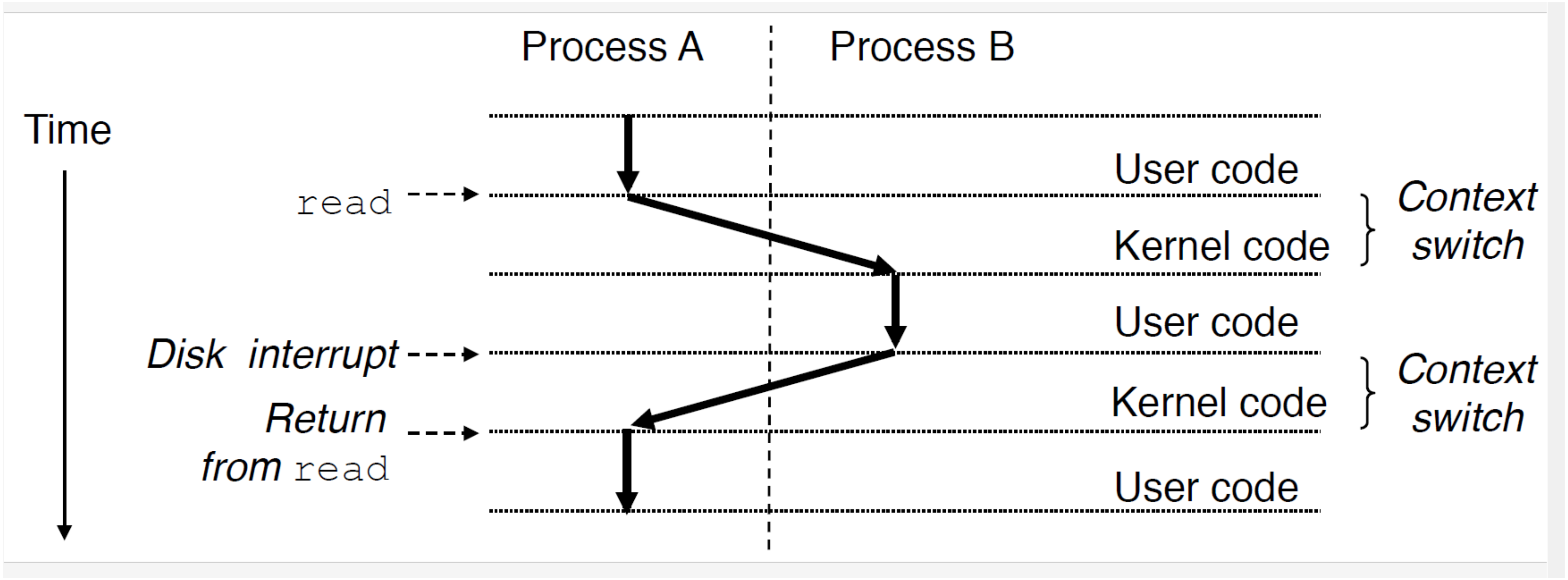

Image credit: CSAPP

Memory allocation for a process

- When is memory allocated for code/data in RAM?
- When OS creates process, memory to store compiled executable allocated in RAM
  - Executable contains code (instructions) in the program, global/static variables in program
- Should we allocate memory for local variables, arguments of functions in executable?
  - No, since we do not now if/how many times the function will be called at runtime
- Similarly, malloc is for dynamic memory allocation at runtime, not compile time

```
int g;

int increment(int a) {
    int b;
    b = a+1;
    return b;
}

main() {
    int x, y;
    x = 1;
    y = increment(x);

    int *z = malloc(40);

}
```

Example memory allocation

- Variable “g” allocated at compile time
- Function local variables, arguments stored on “stack”
  - Example: variables “x”, “y” of main, variables “a”, “b” of function “increment”
  - During function call, arguments and local variables are “pushed” (allocated memory) on the stack, “popped” when function returns
- Dynamically memory allocated on “heap”
  - Malloc returns address of allocated chunk on heap
  - Example: memory address of 40 bytes on heap is stored in pointer variable “z” which is on the stack

```
int g;

int increment(int a) {
    int b;
    b = a+1;
    return b;
}

main() {
    int x, y;
    x = 1;
    y = increment(x);

    int *z = malloc(40);

}
```

Memory image of a process

- Memory image of a process: code+data of process in memory
  - Code: CPU instructions in the program
  - Compile-time data: global/static variables in program executable
  - Runtime data: stack+heap for dynamic memory allocation at runtime
- Heap and stack can grow/shrink as process runs, with help of OS
  - Stack pointer CPU register keeps track of top of stack
- Memory image also contains other code (not directly part of the program) that the process may want to execute, e.g., programming language libraries, kernel code and data, and so on

Address space of a process

- Which memory addresses contain what part of memory image?
- OS gives every process the illusion that its memory image is laid out contiguously from memory address 0 onwards
    - This view of process memory is called the virtual address space
- In reality, processes are allocated free memory in small chunks all over RAM at some physical addresses, which the programmer is not aware of
    - Pointer addresses printed in a program are virtual addresses, not physical
- When a process accesses a virtual address, OS arranges to retrieve data from the actual physical address
- OS virtualizes memory for all processes

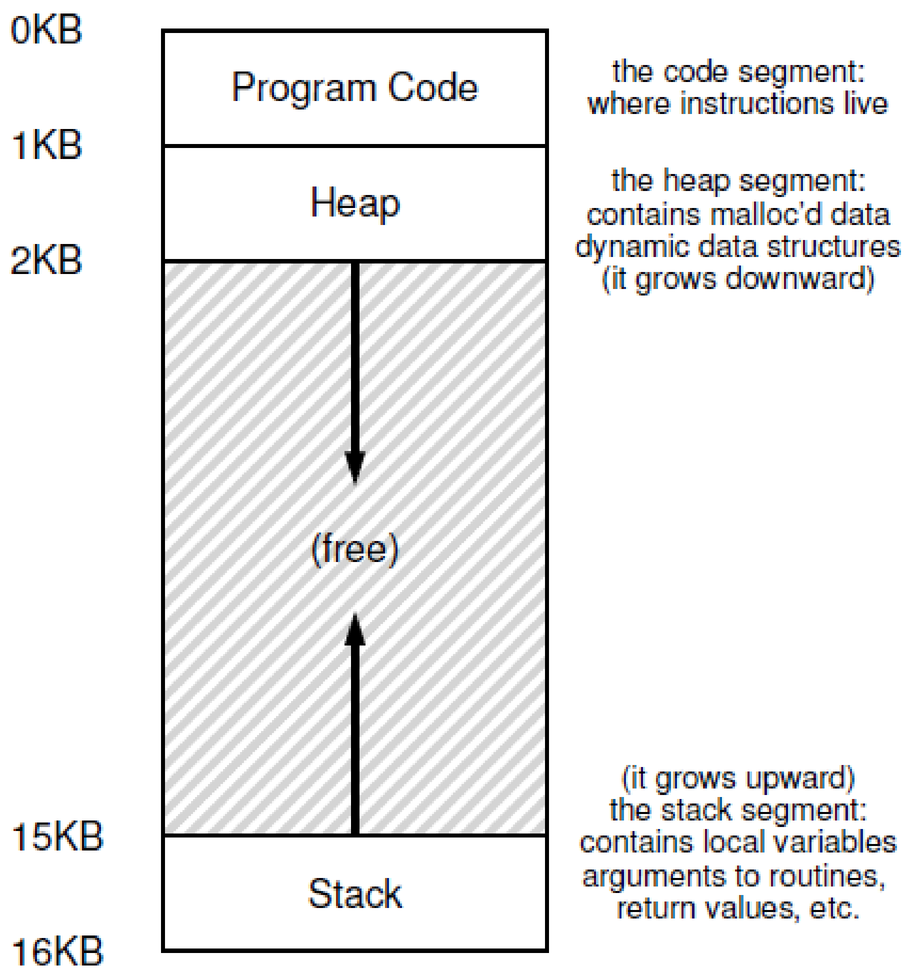
0KB
Program Code
the code segment: where instructions live
1KB
Heap
the heap segment: contains malloc'd data dynamic data structures (it grows downward)
2KB
(free)
(it grows upward) the stack segment: contains local variables arguments to routines, return values, etc.
15KB
Stack
16KB

Figure 13.3: **An Example Address Space**

Image credit: OSTEP

Isolation and privilege levels

- How to protect processes from one another?
  - Can one process mess up the code or data of another process?
- Modern CPUs have mechanisms for isolation
- Privileged and unprivileged instructions
  - Privileged instruction access (perform) sensitive information (actions)
  - Regular instructions (e.g., add) are unprivileged
- CPU has multiple modes of operation (Intel x86 CPUs run in 4 rings)
  - Low privilege level (e.g., ring 3) only allows unprivileged instructions
  - High privilege level (e.g., ring 0) allows privileged instructions also

User mode and kernel mode

- User programs runs in user (unprivileged) mode
  - CPU is in unprivileged mode, executes only unprivileged instructions
- OS runs in kernel (privileged) mode
  - CPU is in privileged mode, can execute both privileged and unprivileged instructions
- CPU shifts from user mode to kernel mode and executes OS code when following events occur:
  - System calls: user request for OS services
  - Interrupts: external events that require attention of OS
  - Program faults: errors that need OS attention
- After performing required actions, OS returns back to user program, CPU shifts back to user mode

System calls

- When user program requires a service from OS, it makes a system call
  - Example: Process makes system call to read data from hard disk
  - Why? User process cannot run privileged instructions that access hardware
  - CPU jumps to OS code that implements system call, and returns back to user code
- Normally, user program does not call system call directly, but uses language library functions
  - Example: printf is a function in the C library, which in turn invokes the system call to write to screen

Interrupts

- In addition to running user programs, CPU also has to handle external events (e.g., mouse click, keyboard input)
- Interrupt = external signal from I/O device asking for CPU's attention
- Example: program issues request to read data from disk, and disk raises interrupt when data is available (instead of program waiting for data)

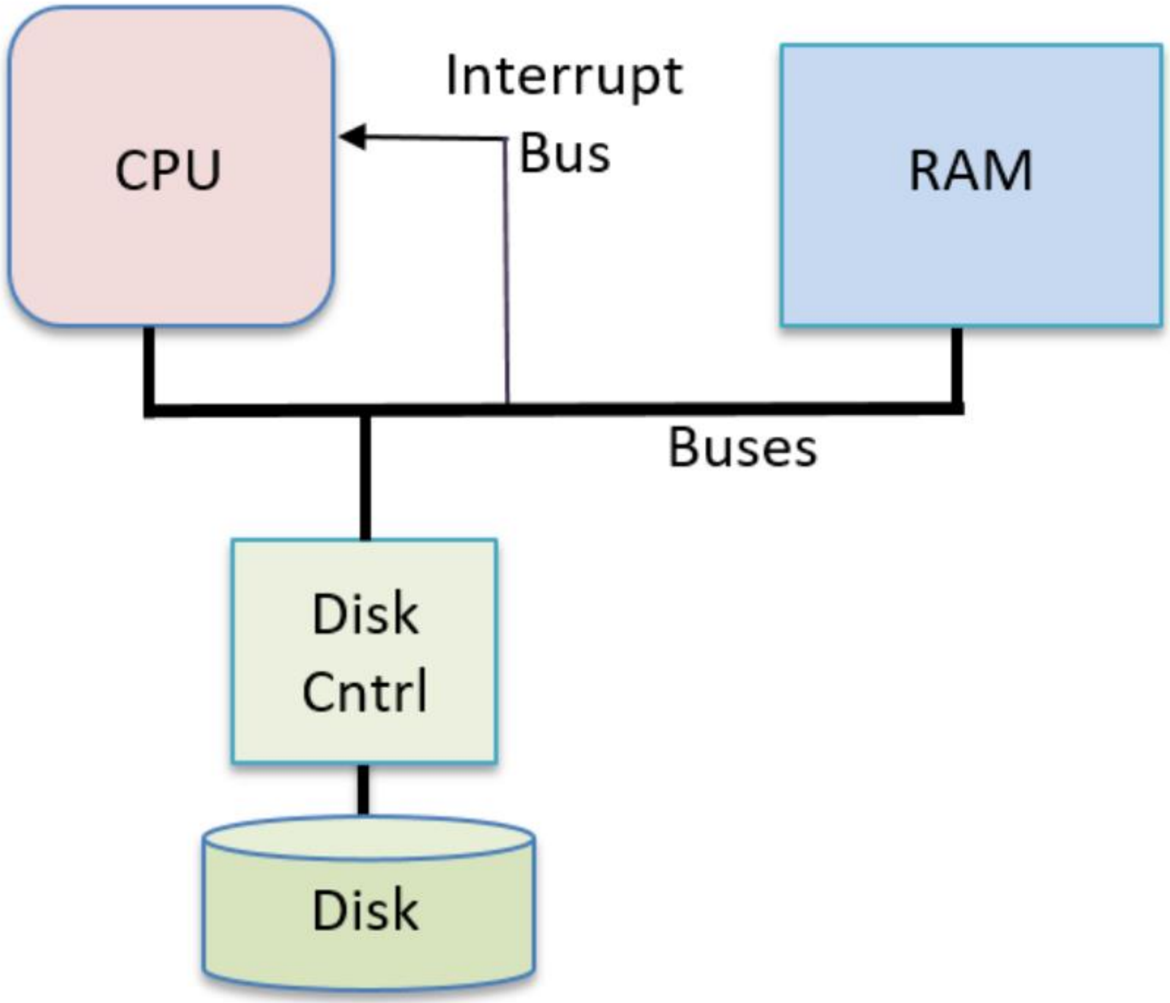

Image credit: Dive into Systems

System calls vs. interrupts

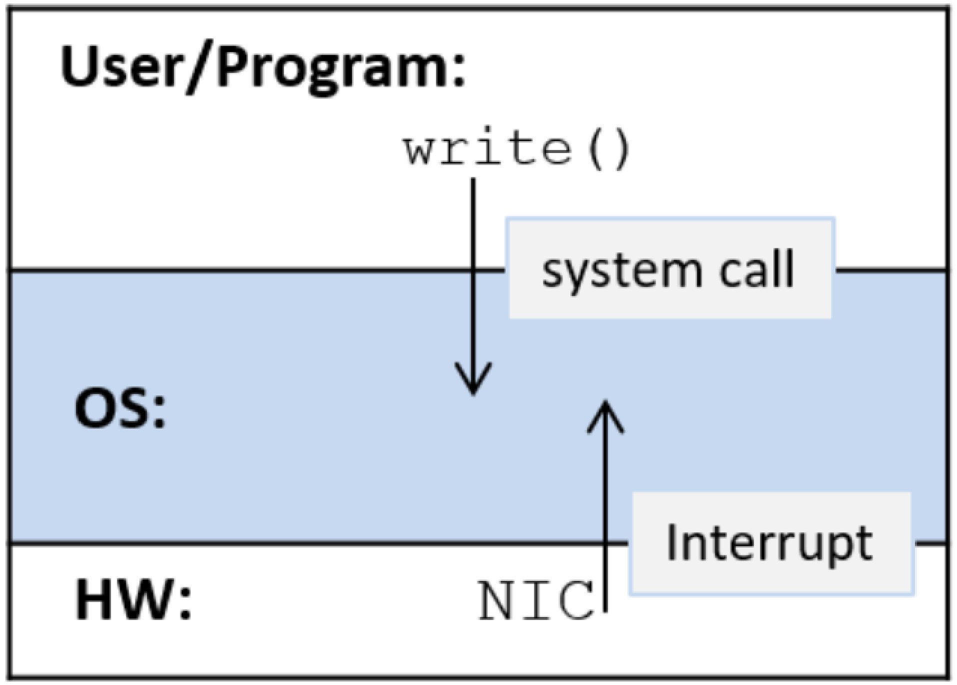

Figure 2. In an interrupt-driven system, user-level programs make system calls, and hardware devices issue interrupts to initiate OS actions.

Image credit: Dive into Systems

Interrupt handling

- How are interrupts handled?
  - CPU is running process P and interrupt arrives
  - CPU saves context of P, runs OS code to handle interrupt (e.g., read keyboard character) in kernel mode
  - Restore context of P, resume P in user mode
- Interrupt handling code is part of OS
  - CPU runs interrupt handler of OS and returns back to user code

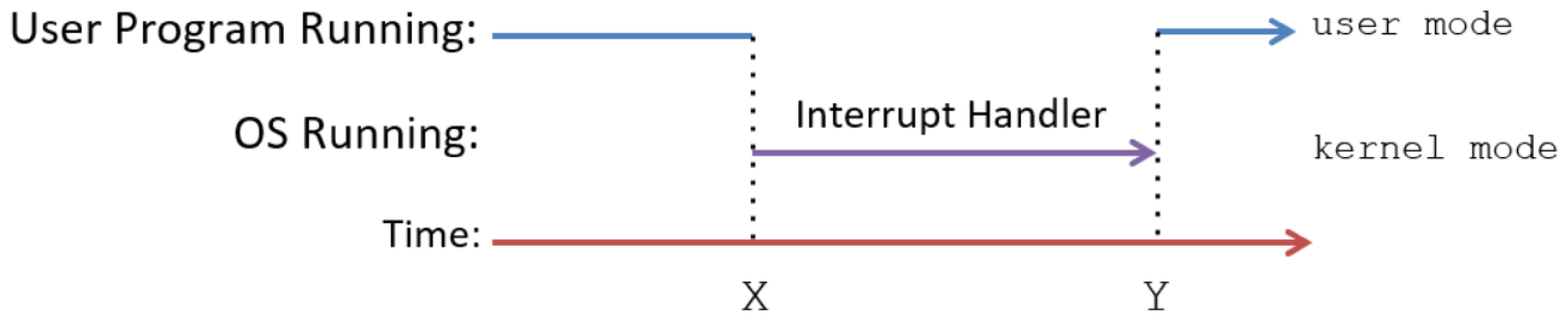

Image credit: Dive into Systems

Device controller and device driver

- I/O device is managed by a device controller
  - Microcontroller which communicates with CPU/memory over bus
- Device specific knowledge required to correctly communicate with device controller to handle I/O operations
  - Done by special software called device driver
  - Part of operating system code
- Functions performed by device driver
  - Initialize I/O devices
  - Start I/O operations, give commands to device (e.g., read data from hard disk)
  - Handle interrupts from device (e.g., disk raises interrupt when data is ready)

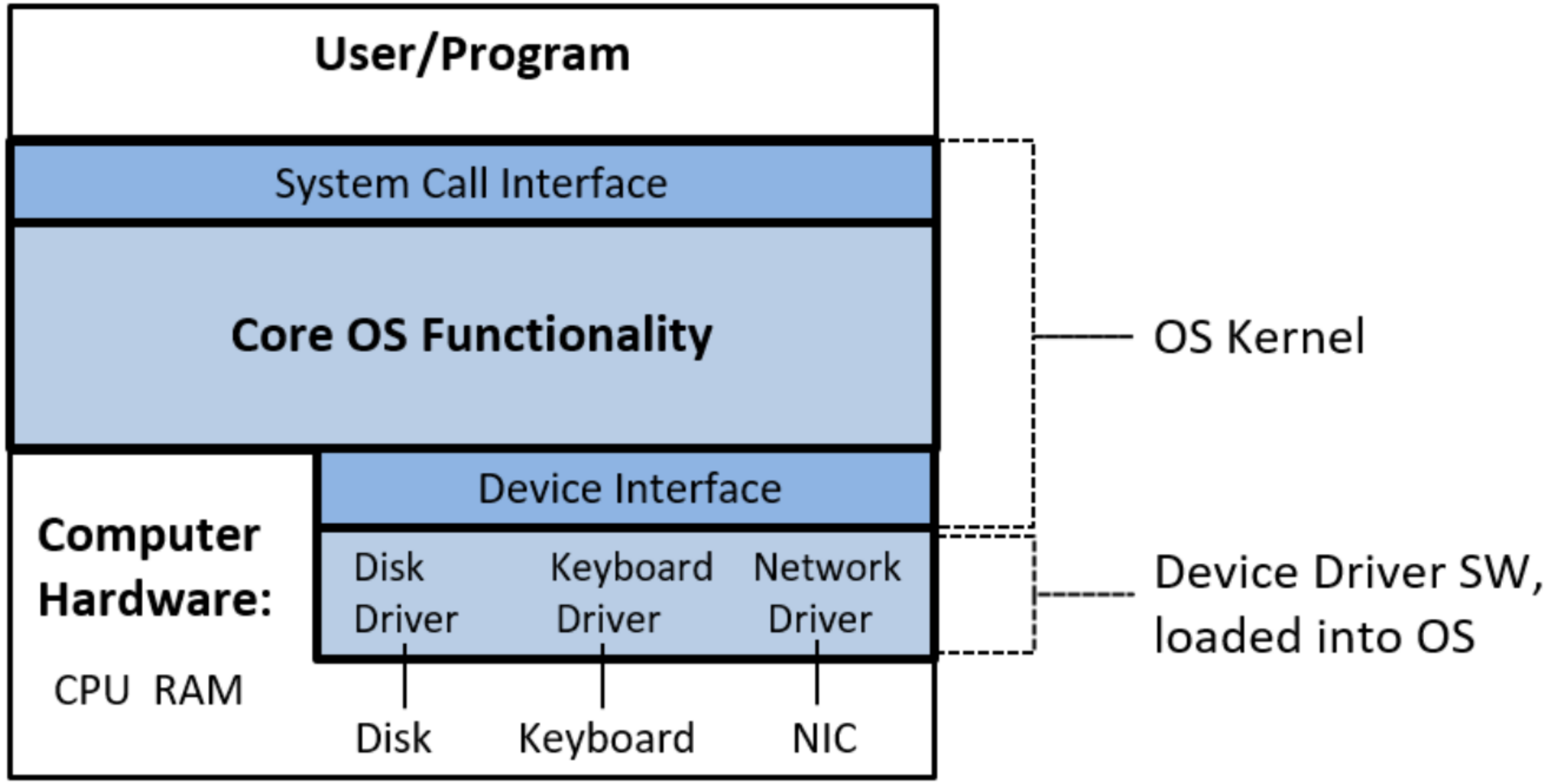

Figure 2. The OS kernel: core OS functionality necessary to use the system and facilitate cooperation between I/O devices and users of the system

Image credit: Dive into Systems

Summary

You want to run a program?
Then you should ask the OS
It manages the hardware for you
So that you do not make a mess
It creates a process in RAM
Initializes it from your a.out
Sets the CPU running on your code
And out of the scene it bows out

…until you call it again for an interrupt or a system call…