Introduction to Operating Systems

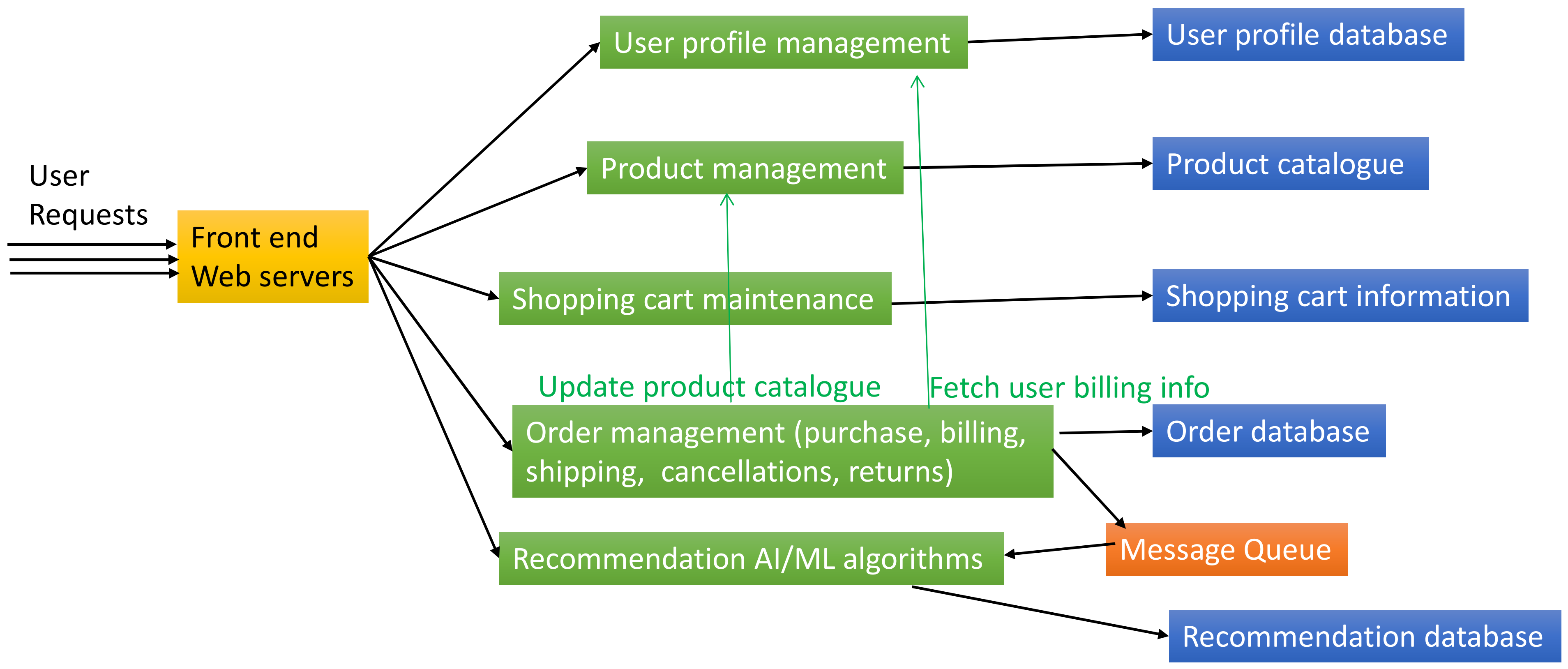
Mythili Vutukuru

CSE, IIT Bombay

Computer systems

- Real-world computer systems are complex
  - Multiple components/tiers distributed across several machines
  - Handle high number of user requests efficiently, reliably
- Example: consider an e-commerce application
  - Clients access multi-tier applications hosted in data centers or public clouds
  - Front-end components (e.g., web servers) receive user requests, reply to user with responses, consult various application servers to build responses
  - App servers contain business logic to process different types of user requests
  - Application data is stored in several database servers in the backend
  - Each of these components is built over one or more computers

Example: e-commerce system



The building blocks

- A single computer system is the building block for all large, distributed computer systems that run real world applications
- What does a computer system contain?
  - Hardware: CPU, memory, I/O devices, …
  - System software: Operating System (OS), …
  - User software: user applications (browser, email client, games, application servers, databases, AI/ML algorithms, …)
- We must understand the basic building blocks of a single system before we can build large-scale systems for real applications

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?
  - How to make your programs more secure, reliable, tolerant to failures?
  - Why is your program running slowly and how to fix it?
  - How much CPU/memory is your program consuming, and why?
- OS expertise is one of the most important skills when building high performance, robust, complex real life systems

Beyond OS to real systems and future courses

- 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 across 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 functionality of the OS
  - Other useful programs = shell, commands on shell, other utilities that help users interact with the OS

What is an operating system?

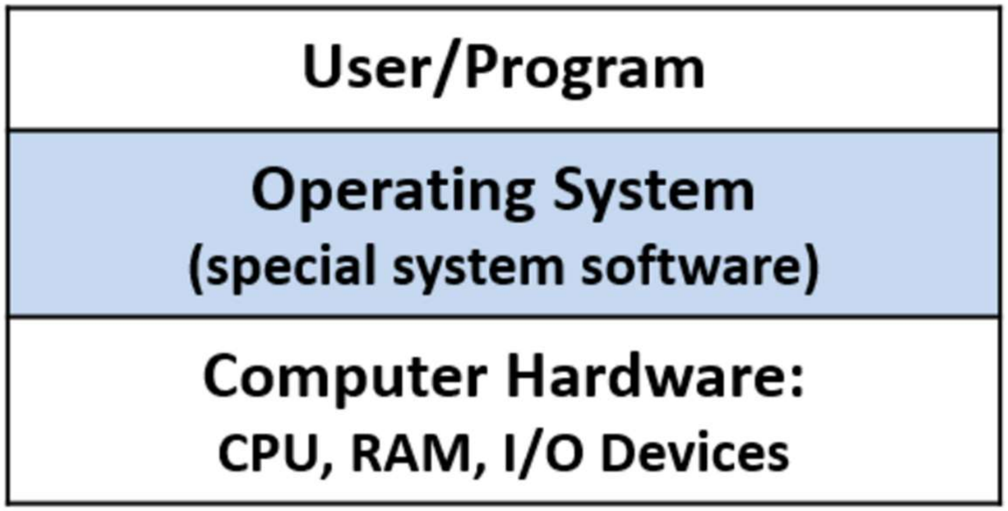


Figure 1. The OS is special system software between the user and the hardware. It manages the computer's hardware and implements abstractions to make the hardware easier to use.

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

Key concepts in operating systems

- The OSTEP textbook identifies 3 concepts that are fundamental to OS:
- Virtualization: OS gives a "virtual" or logical view of the hardware to the users, hiding the messy real "physical" view, so that each user/program has the illusion of having entire hardware to itself
- Concurrency: OS runs multiple user programs at the same time, while sharing the system resources across users efficiently and securely
- Persistence: OS stores user data persistently on external I/O devices
- We will now understand these concepts and other OS terminology

What is a program?

- User program = code (instructions for CPU) + data to do a specific task
- Stored program concept
  - User programs stored in main memory (instructions + data)
  - Memory is byte-addressable: data accessed via memory address / location / byte#
  - CPU fetches code/data from memory using 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 (one physical core can appear as multiple logical cores by sharing hardware, and hence run multiple processes at once)

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 (compile time data allocated at start, runtime data allocated as program runs)
  - 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 executable
  - Allocates memory for stack, heap
- Initializes CPU context
  - PC points to first instruction
- Process starts to run
  - CPU runs user instructions now
  - OS is out of picture, but steps in later as needed

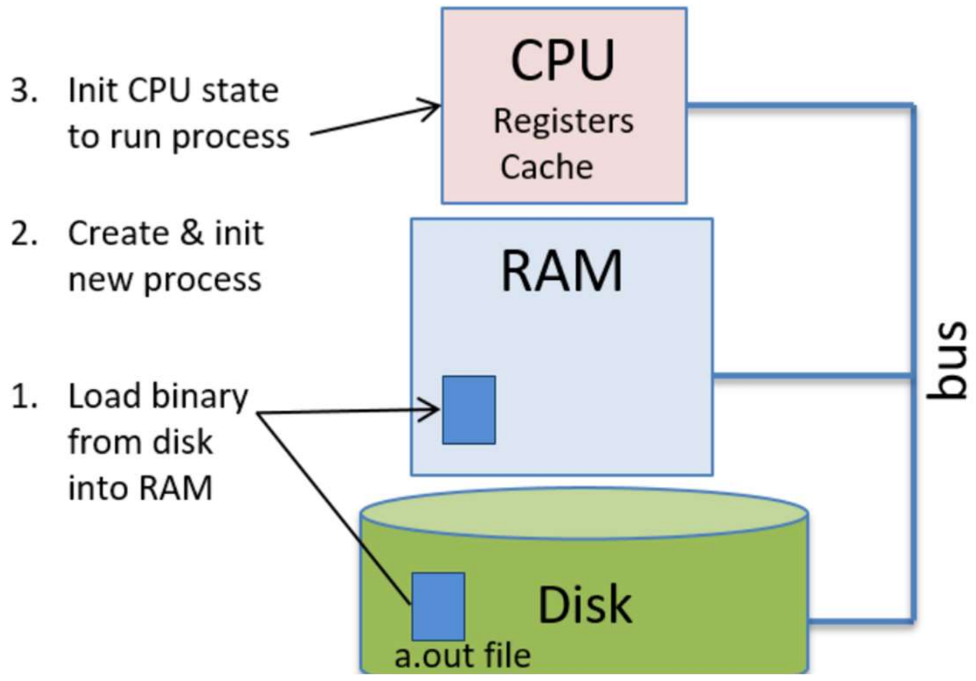

3. Init CPU state to run process
2. Create & init new process
1. Load binary from disk into RAM
CPU
Registers
Cache
RAM
Disk
a.out file
bus

Image credit: Dive into Systems

Concurrent execution & CPU virtualization

- CPU runs multiple programs concurrently
  - OS runs one process for a bit, then switches to another, switches again, ...
- How does OS 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
- In this manner, OS virtualizes CPU across multiple processes
- OS scheduler decides which process to run on which CPU at what time

Context switching

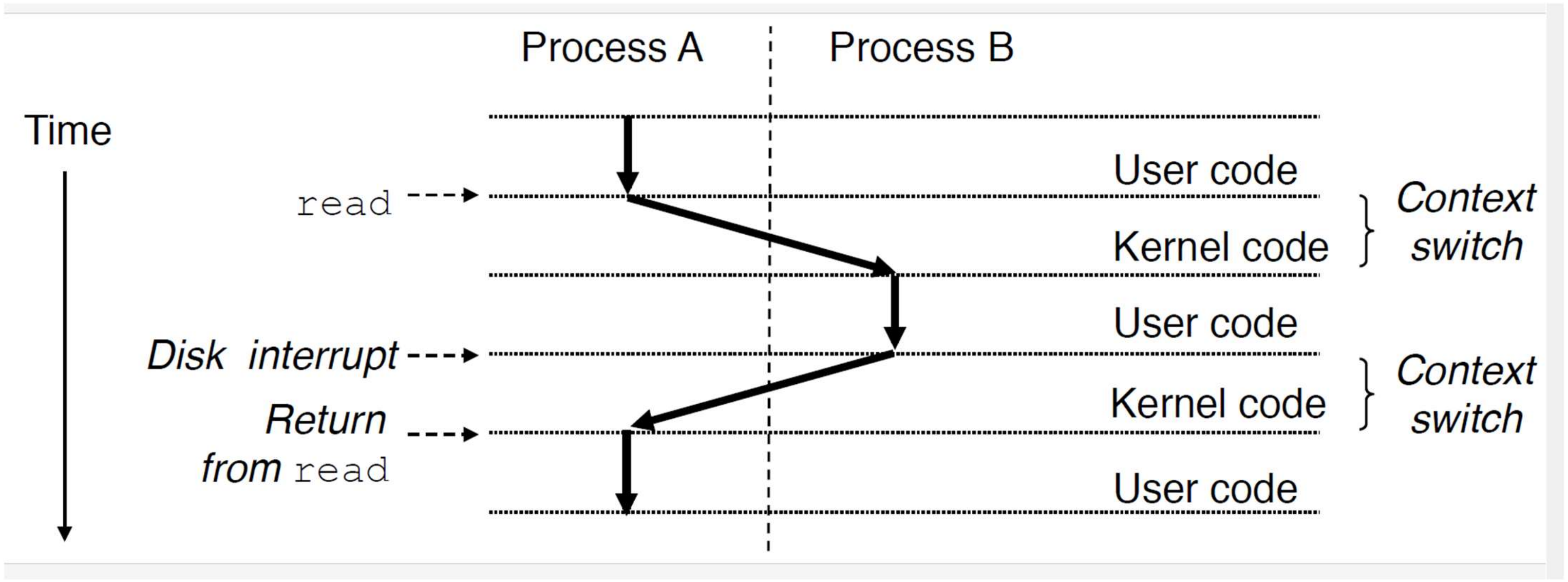


Image credit: CSAPP

Memory 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 (more later)

Address space of a process

- 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, gives illusion of a virtual address space to processes

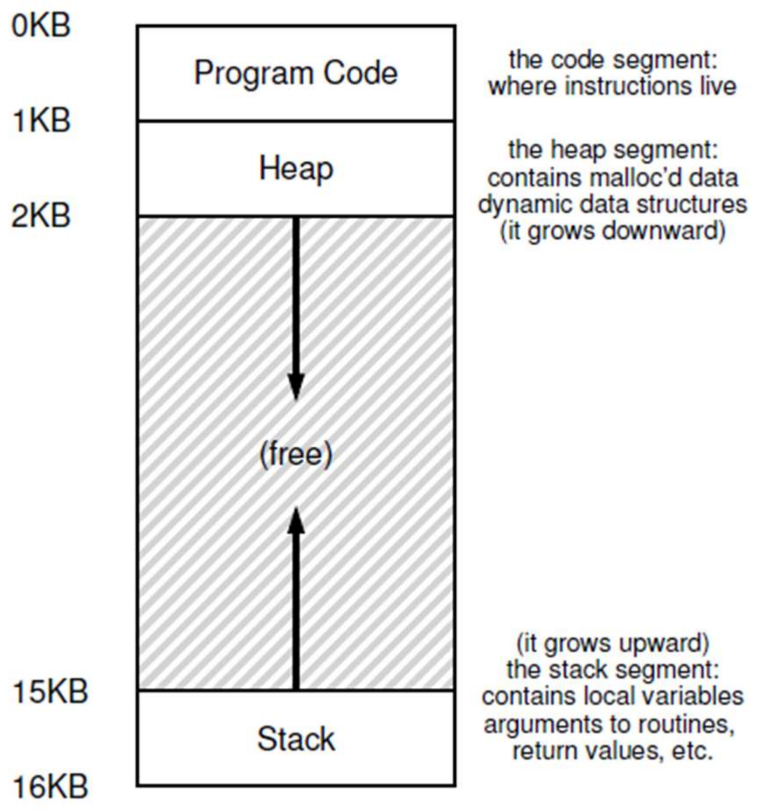

0KB
Program Code
the code segment: where instructions live
1KB
Heap
the heap segment: contains malloc'd data dynamic data structures (it grows downward)
2KB
(free)
(it grows upward) the stack segment: contains local variables arguments to routines, return values, etc.
15KB
Stack
16KB

Figure 13.3: **An Example Address Space**

Image credit: OSTEP

Isolation and privilege levels

- How to protect concurrent processes from one another?
  - Can one process mess up the code or data of another process?
  - When we virtualize, how do we share safely?
- 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 in kernel mode, 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, to prevent one user from harming another
  - CPU jumps to OS code that implements system call, and returns back to user code after system call completes
- 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)

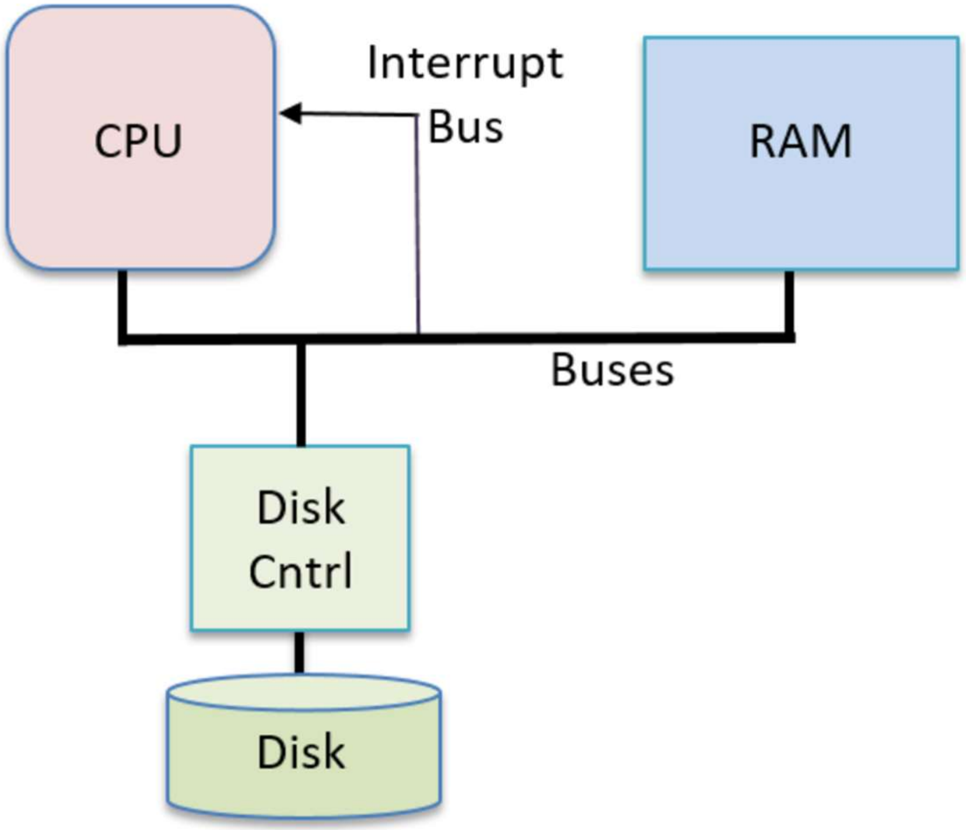


Image credit: Dive into Systems

System calls vs. interrupts

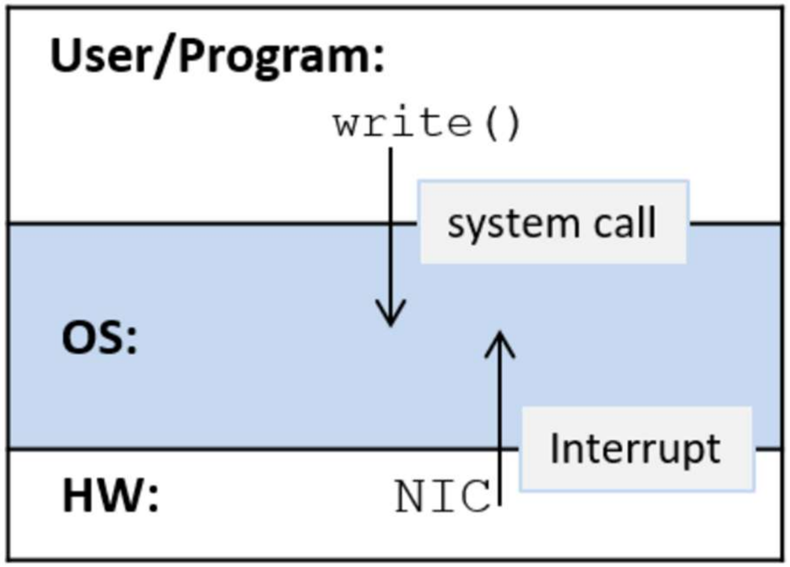


Figure 2. In an interrupt-driven system, user-level programs make system calls, and hardware devices issue interrupts to initiate OS actions.

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

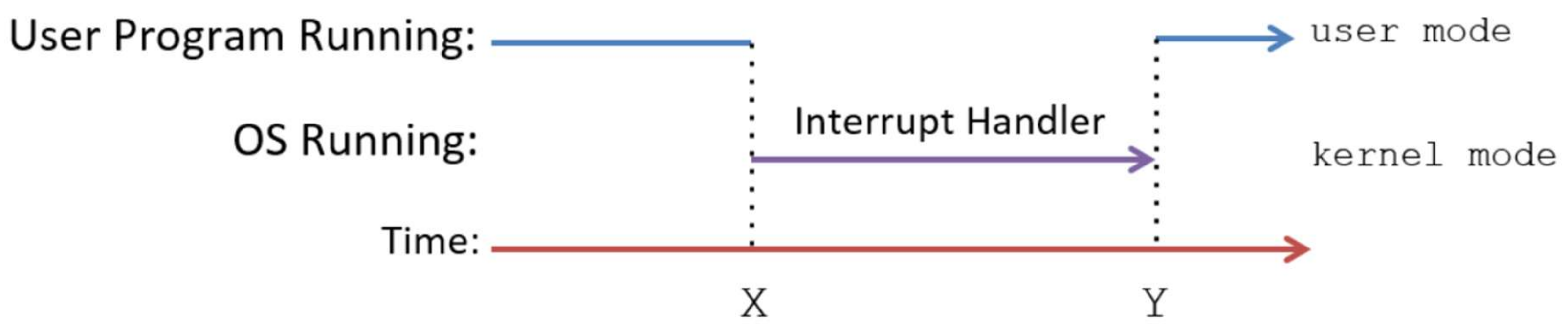


Image credit: Dive into Systems

I/O devices

- CPU and memory connected via high speed system (memory) bus
- I/O devices connect to the CPU and memory via other separate buses
  - Interface with external world
  - Store user data persistently
- OS manages I/O devices on behalf of the users

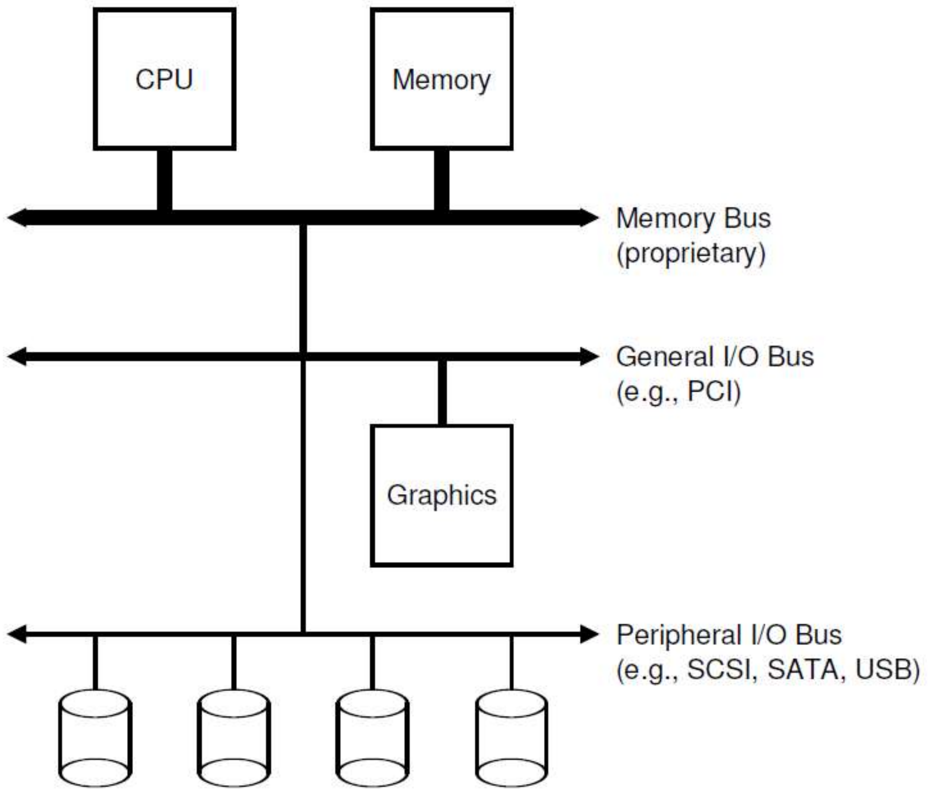

CPU
Memory
Memory Bus (proprietary)
General I/O Bus (e.g., PCI)
Graphics
Peripheral I/O Bus (e.g., SCSI, SATA, USB)

Image credit: OSTEP

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 kernel 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)

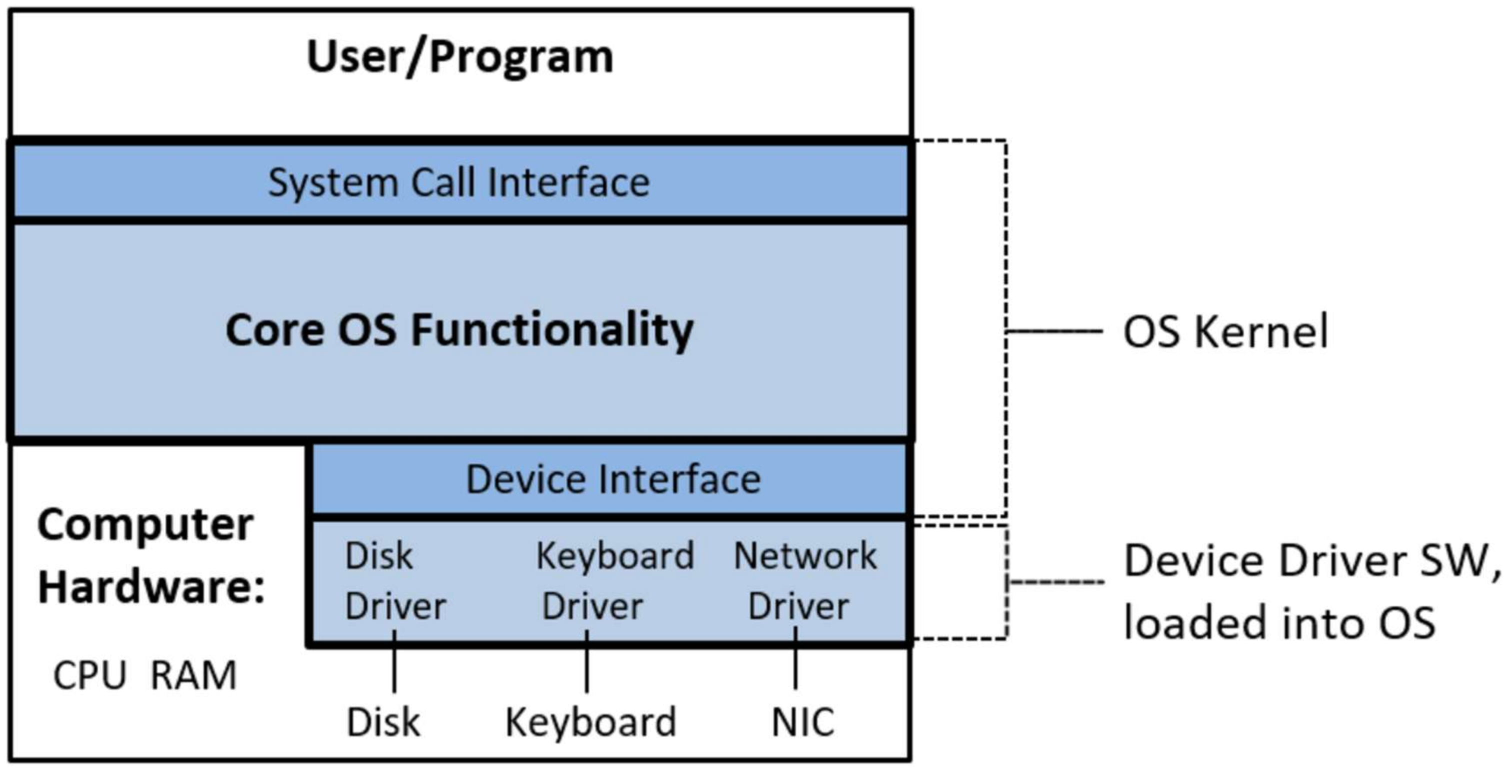


Figure 2. The OS kernel: core OS functionality necessary to use the system and facilitate cooperation between I/O devices and users of the system

Image credit: Dive into Systems