OS Mechanisms

Mythili Vutukuru
CSE, IIT Bombay

Key concepts in OS

- 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
- Concurrency: OS runs multiple user programs at the same time, giving each user/program the illusion that it has the entire hardware to itself
- Persistence: OS stores user data persistently on external I/O devices
- We will now understand these concepts and other OS terminology

Concurrent execution & CPU virtualization

- CPU runs multiple programs concurrently
  - Run one process, switch to another, switch 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
  - OS takes care of this switching across processes
- In this manner, OS virtualizes CPU across multiple processes
- OS scheduler decides which process to run on which CPU at what time

Context switching

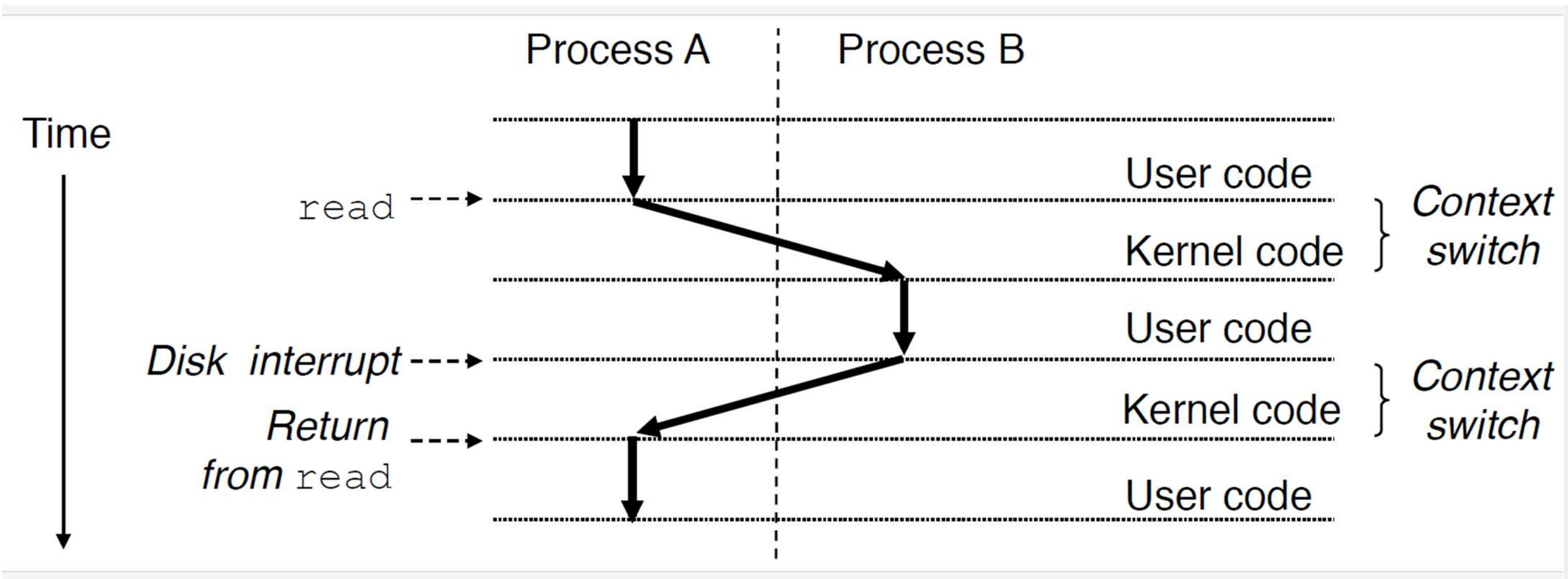


Image credit: CSAPP

Memory 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);

}
```

After the program execution OS will reclaim all the memory it gave to the program to prevent long term memory loss.

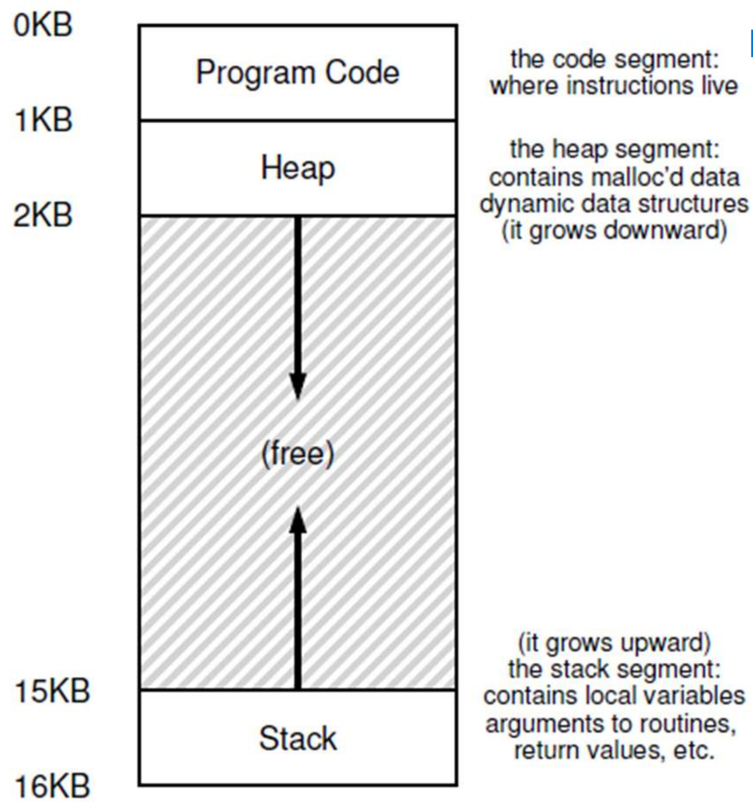
Memory image of a process

- 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

Program counter points here.



Data for each stack are stored in here.

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, 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)

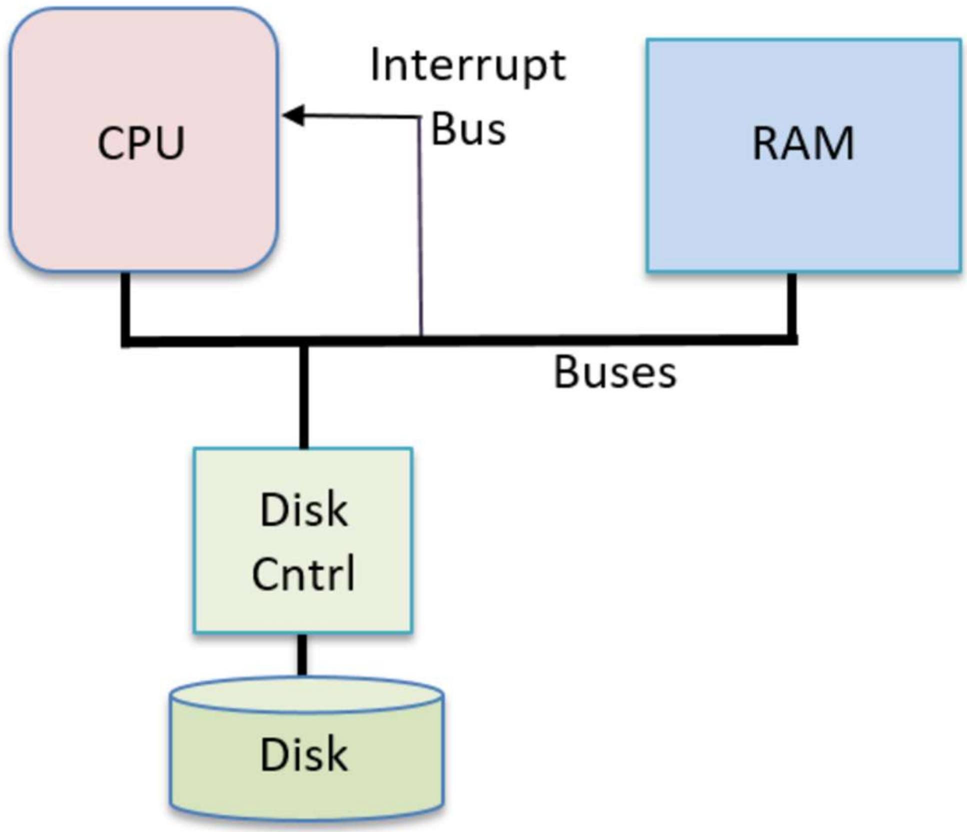


Image credit: Dive into Systems

System calls vs. interrupts

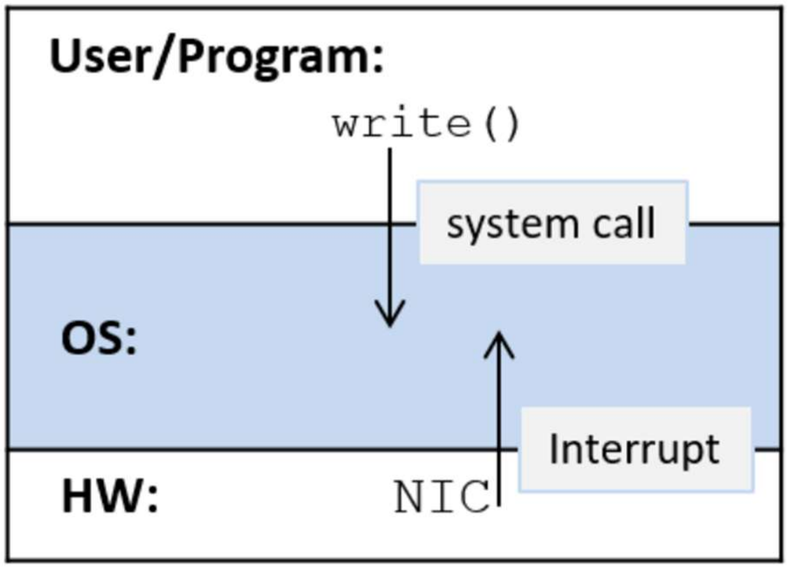


Figure 2. In an interrupt-driven system, user-level programs make system calls, and hardware devices issue interrupts to initiate OS actions.

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

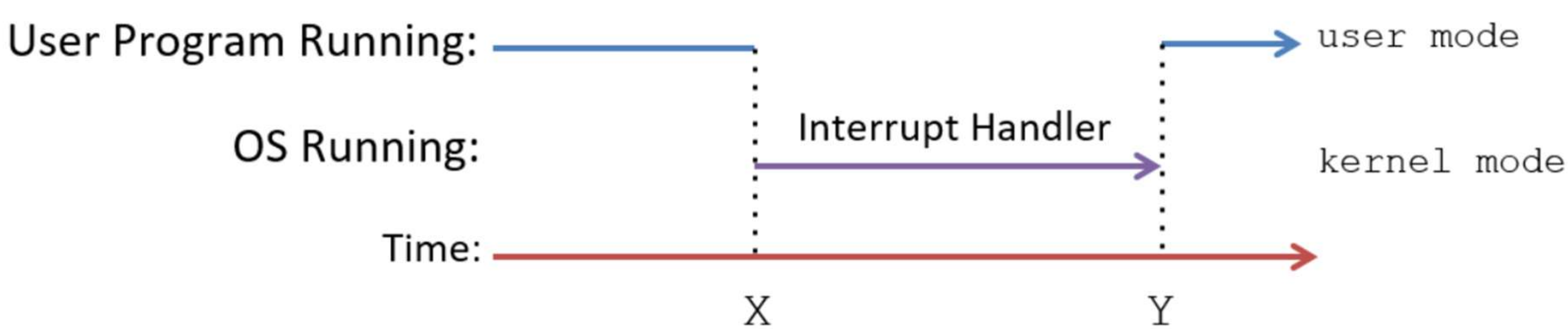


Image credit: Dive into Systems

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

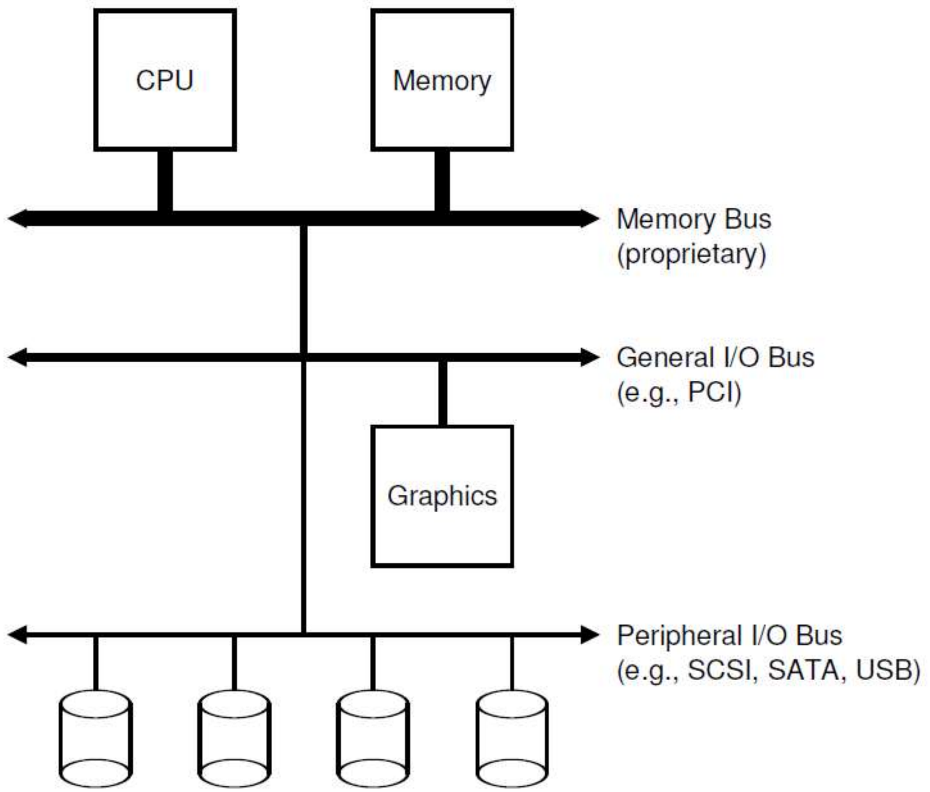


Image credit: OSTEP

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)

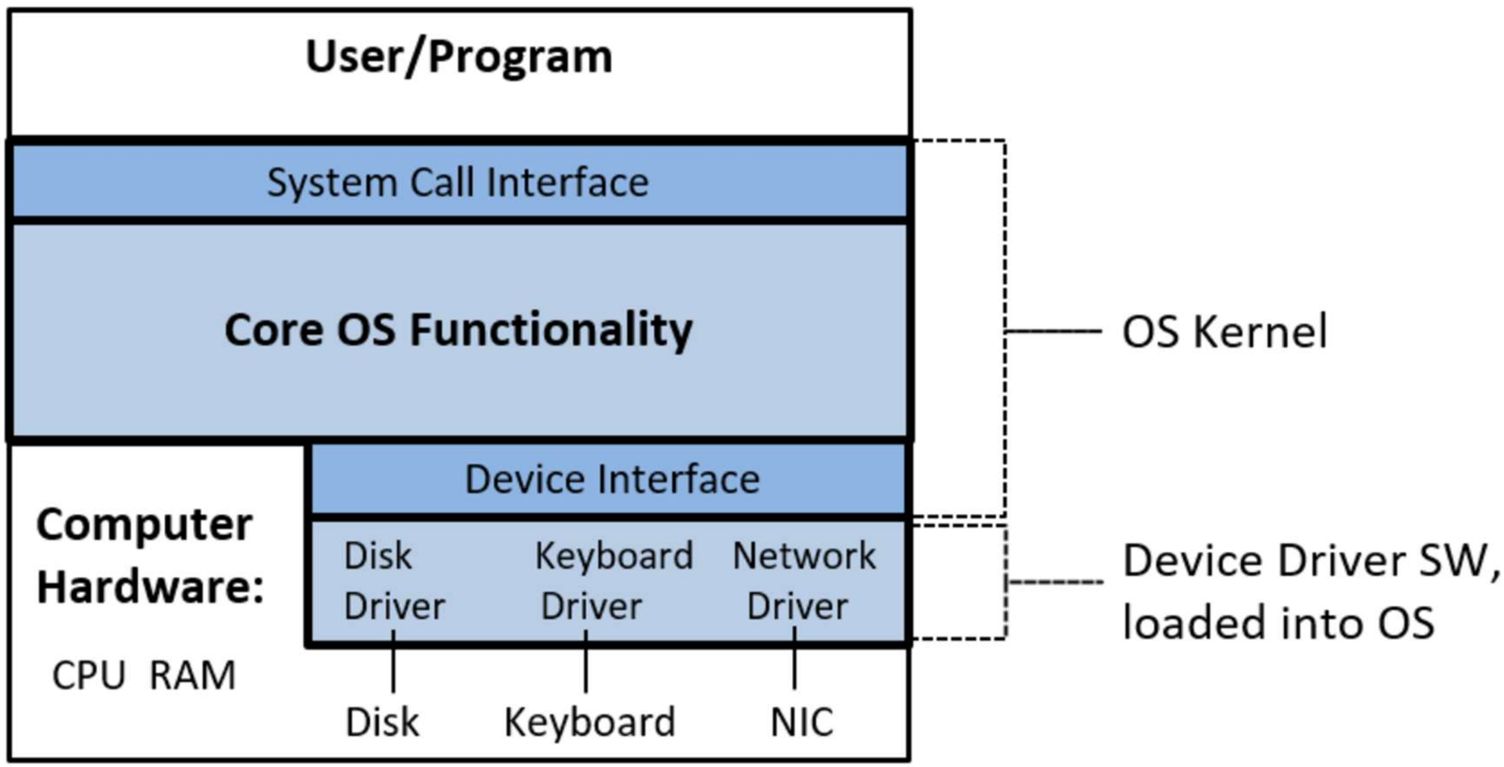


Figure 2. The OS kernel: core OS functionality necessary to use the system and facilitate cooperation between I/O devices and users of the system

Image credit: Dive into Systems