# OPERATING SYSTEM

# Study Notes

Zhijie Xia
Study Notes
github.com/zhijie-os

# Contents

# Chapter 1

# Overview

## 1.1 Different views of operating system

An abstract view of components of a computer system.

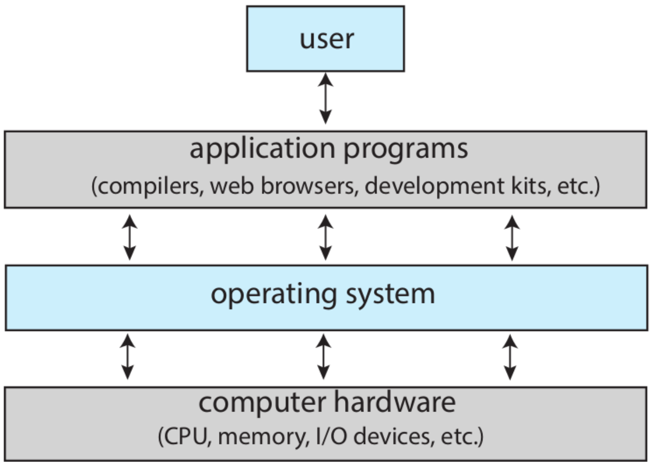

user
application programs
(compilers, web browsers, development kits, etc.)
operating system
computer hardware
(CPU, memory, I/O devices, etc.)

Figure 1.1: Abstract view of the components of a computer system.

### 1.1.1 User view

Operating system is designed for ***ease of use***, with some attention paid to performance and security and none attention paid to ***resource utilization*** -how various hardware and software resources are shared.

### 1.1.2 System view

The operating system is the program most intimately involved with the hardware. We can view an operating system as a ***resource allocator***.

A slightly different view of an operating system emphasizes the need to control the various I/O devices and user programs. Therefore, an operating system is a ***control program***.

### 1.1.3 Defining Operation Systems

A common definition: The operating system is the one program running at all times on the computer — usually called ***kernel***. Along with the kernel, there are two other types of programs: ***system programs***, which are associated with the operating system but are not necessarily part of the kernel, and application programs, which include all programs not associated with the operation of the system.

## 1.2 Computer-System Organization

A modern general-purpose computer system consists of one or more CPUs and a number of device controllers connected through a common ***bus*** that provides access between components and shared memory.

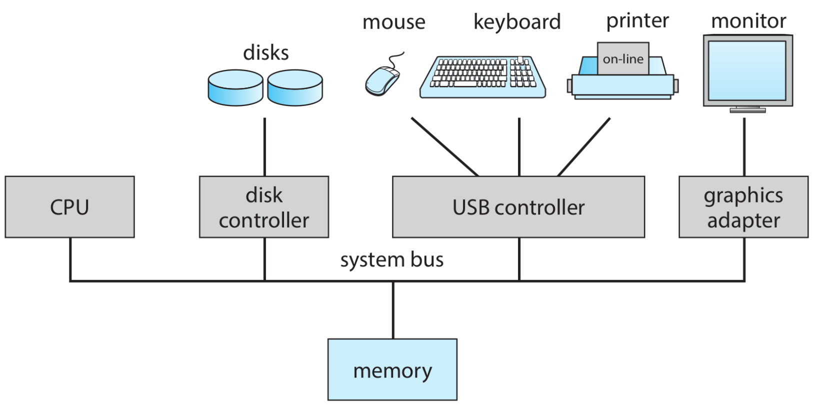


Figure 1.2: A typical PC computer system

Typically, operating systems have a ***device driver*** for each device controller. This device driver understands the device controller and provides the rest of the operating system with a uniform interface to the device. The CPU and the device controllers can tbiexecute in parallel, competing for memory cycles. To ensure orderly access to the shared memory, a memory controller synchronizes access to the memory.

### 1.2.1 Interrupts

To start an I/O operation, the device driver loads the appropriate registers in the device controller. The device controller, in turn, examines the contents of these registers to determine what action to take. The controller starts the transfer of data from the device to its local buffer. Once the transfer of data is complete, the device controller informs the device driver that it has finished its operation by issuing an ***interrupt***.

#### Overview of Interrupt

Hardware may trigger an interrupt at any time by sending a signal to the CPU, usually by way of the ***system bus*** (The system bus is the main communications path between the major components).

Interrupts are a key part of how operating system and hardware interact.

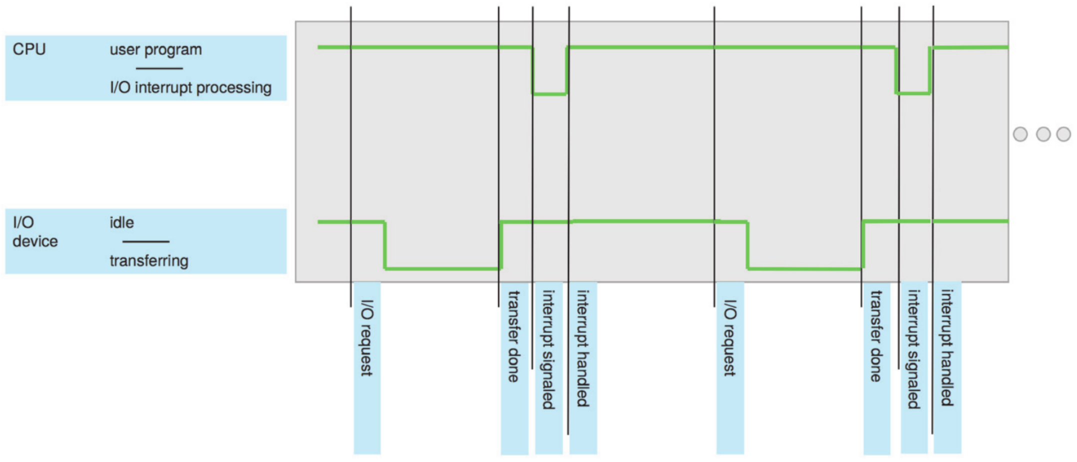


Figure 1.3: Interrupt timeline for a single program doing output.

When the CPU is interrupted, commonly, it transfers control to the appropriate interrupt service routine. A generic routine is used to examine the interrupt information. The routine, in turn, would call the interrupt-specific handler.

Interrupts need to be handled very qucickly, as they occur very frequently. ***interrupt vetor***, which is a table of pointers to interrupt routine, is used to call interrupt routines indirectly. The interrupt architecture must also save the state information of whatever was interrupted, so that it can store this information after servicing the interrupt.

## Implementation

The CPU hardware has a wire called ***interrupt-request line*** that the CPU senses after executing every instruction. When the CPU detects that a controller has asserted a signal on the interrupt-quest line, it reads the interrupt number and jumps to the ***interrupt-handler routine*** by using that interrupt number as an index into the interrupt vector.

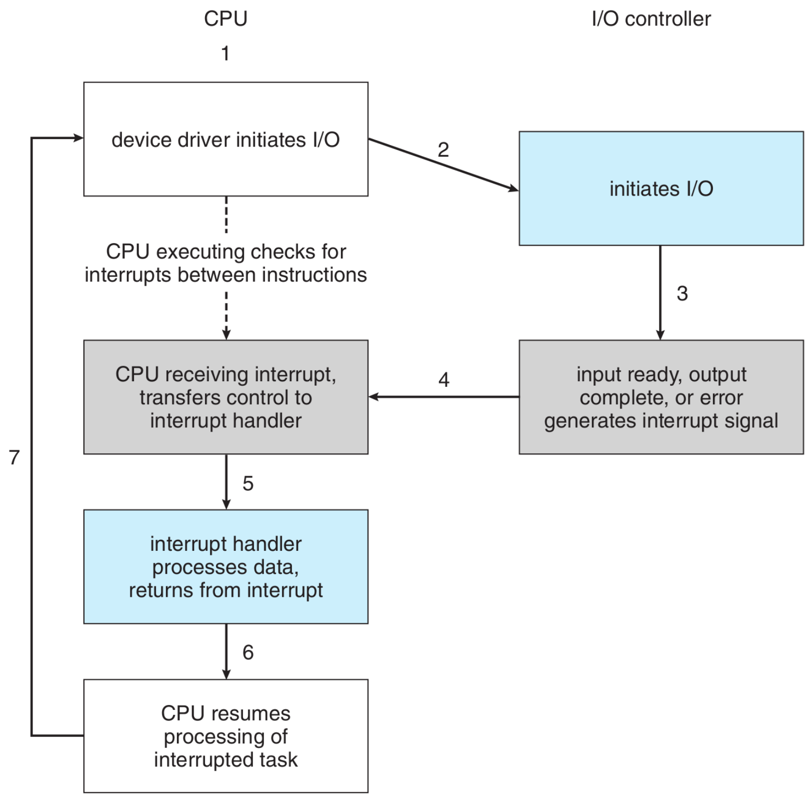

CPU
I/O controller
1
device driver initiates I/O
2
initiates I/O
CPU executing checks for interrupts between instructions
3
CPU receiving interrupt, transfers control to interrupt handler
4
input ready, output complete, or error generates interrupt signal
7
5
interrupt handler processes data, returns from interrupt
6
CPU resumes processing of interrupted task

Figure 1.4: Interrupt-driven I/O cycle.

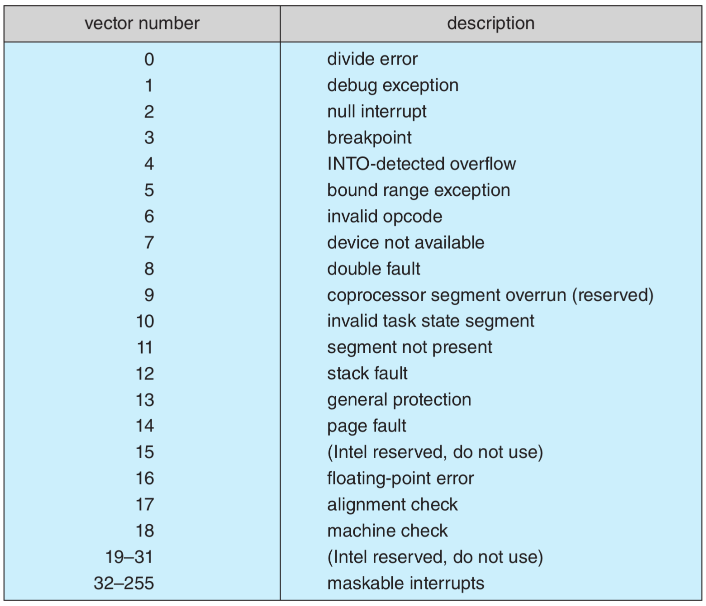
***More sophisticated interrupt-handling features needed***:

1. Defer interrupt handling during critical processing.
2. Dispatch to the proper interrupt handler for a device.
3. Multilevel interrupts, so that operating system can distinguish between high- and low-priority interrupts and can respond with the appropriate degree of urgency.

Defer Interrupt handling during critical processing=¿

***(1).***Most CPUs have two interrupt request lines:

1. Nonmaskable interrupt: Reversed for events such as unrecoverable memory errors.
2. Maskable interrupt: Can be turned off before entering critical instruction sequences. It is used by device controller to request service.



| vector number | description |
|---|---|
| 0 | divide error |
| 1 | debug exception |
| 2 | null interrupt |
| 3 | breakpoint |
| 4 | INTO-detected overflow |
| 5 | bound range exception |
| 6 | invalid opcode |
| 7 | device not available |
| 8 | double fault |
| 9 | coprocessor segment overrun (reserved) |
| 10 | invalid task state segment |
| 11 | segment not present |
| 12 | stack fault |
| 13 | general protection |
| 14 | page fault |
| 15 | (Intel reserved, do not use) |
| 16 | floating-point error |
| 17 | alignment check |
| 18 | machine check |
| 19–31 | (Intel reserved, do not use) |
| 32–255 | maskable interrupts |

Figure 1.5: Intel processor event-vector table.

***(2).***In practice, computers have more devices than they have address elements in the interrupt vector. A common way to solve this problem is to use ***interrupt chaining***, in which each element in the interrupt vector points to the head of a list of interrupt handlers.

***(3).***The interrupt mechanism also implements a system of ***interrupt priority levels***. These levels enable the CPU to defer the handling of low-priority interrupts without masking all interrupts and makes it possible for a high-priority interrupt to preempt the execution of a low-priority interrupt.

### 1.2.2 Storage Structure

The CPU can load instructions only from memory $\Rightarrow$ Any programs must first be loaded into memory to run.

1. General-purpose computers run most of their programs from rewritable memory, called ***main memory***(aka Random-acess memory or RAM). RAM is volatile (loss its content when pwoer off).

2. The first program to run on computer power-on is ***bootstrap program***. Electrically erasable programmable read-only memory (EEPROM) or other forms of firmware is used for ***bootstrap program*** since EEPROM is nonvolatile.

However, main memory is usually too small to store all needed data permanently and it is volatile. Thus, most computer systems provide ***secondary storage*** as an extension of main memory in order to store large quantities of data permanently.

The most common secondary-storage devices are ***hard-disk drives (HDDs)*** and ***nonvolatile memory (NVM) devices***. Most programs are stored in secondary storage until they are loaded into the main memory.

***Tertiary storage***: Large enough and slow enough memory devices. For example, cache memory, CD-ROM, magnetic tapes and so on. They are used for special purposes such as store backup copies of material.

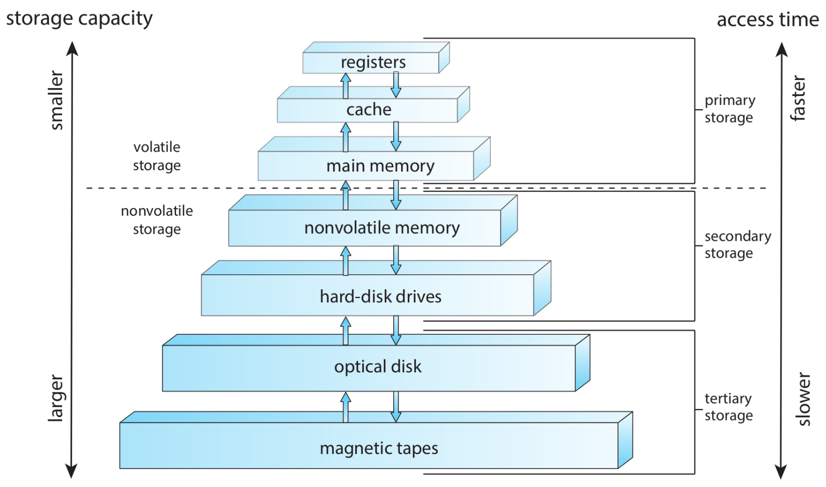


Figure 1.6: Storage Device Hierarchy