UNIVERSITY OF BIRMINGHAM

Computer Systems
Elements of an Operating System

Lecture Objective / Overview

In this lecture, we shall see:

- What happens when your computer starts
- The role of memory in a computer system
- How an OS manages multitasking

Starting your Computer …

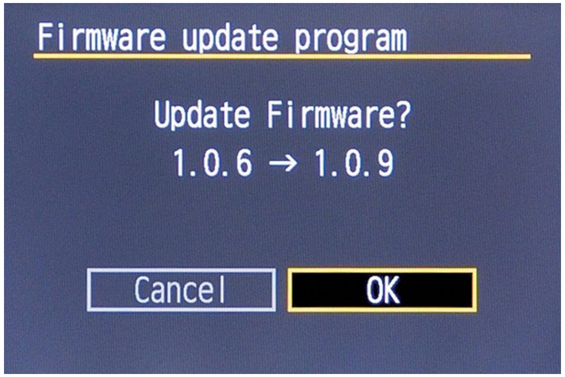
引导程序

- A small bootstrap program is loaded when your computer is started or rebooted
  - Based on the saying "to pull oneself up by one's bootstraps"
  - A seemingly impossible task
  - A self-sustaining process that proceeds without external help
- First coined in 1953 by Computer Scientist Werner Buchholz when talking about the "self-loading procedure" of the IBM Type 701 Computer

https://ieeexplore.ieee.org/document/4051191

Starting your Computer …

- Stored in Read-Only Memory (ROM) or Electrically Erasable Programmable Read-Only Memory (EEPROM).
  - Why? Can not be easily infected by a virus
- Known as firmware
- Initializes all aspects of the system:
  - CPU registers
  - Device controllers
  - Memory contents
- Loads the operating system kernel into memory



Firmware update program

Update Firmware?

1.0.6 → 1.0.9

Once the Firmware is loaded …

- Some services are provided outside of the kernel
  - These are loaded at boot time to become system processes or system daemons 系统守护进程
  - These run the entire time the kernel is running
- On UNIX, the first system process is “init”
- Once this is all loaded, the operating systems sits and waits for events to occur…

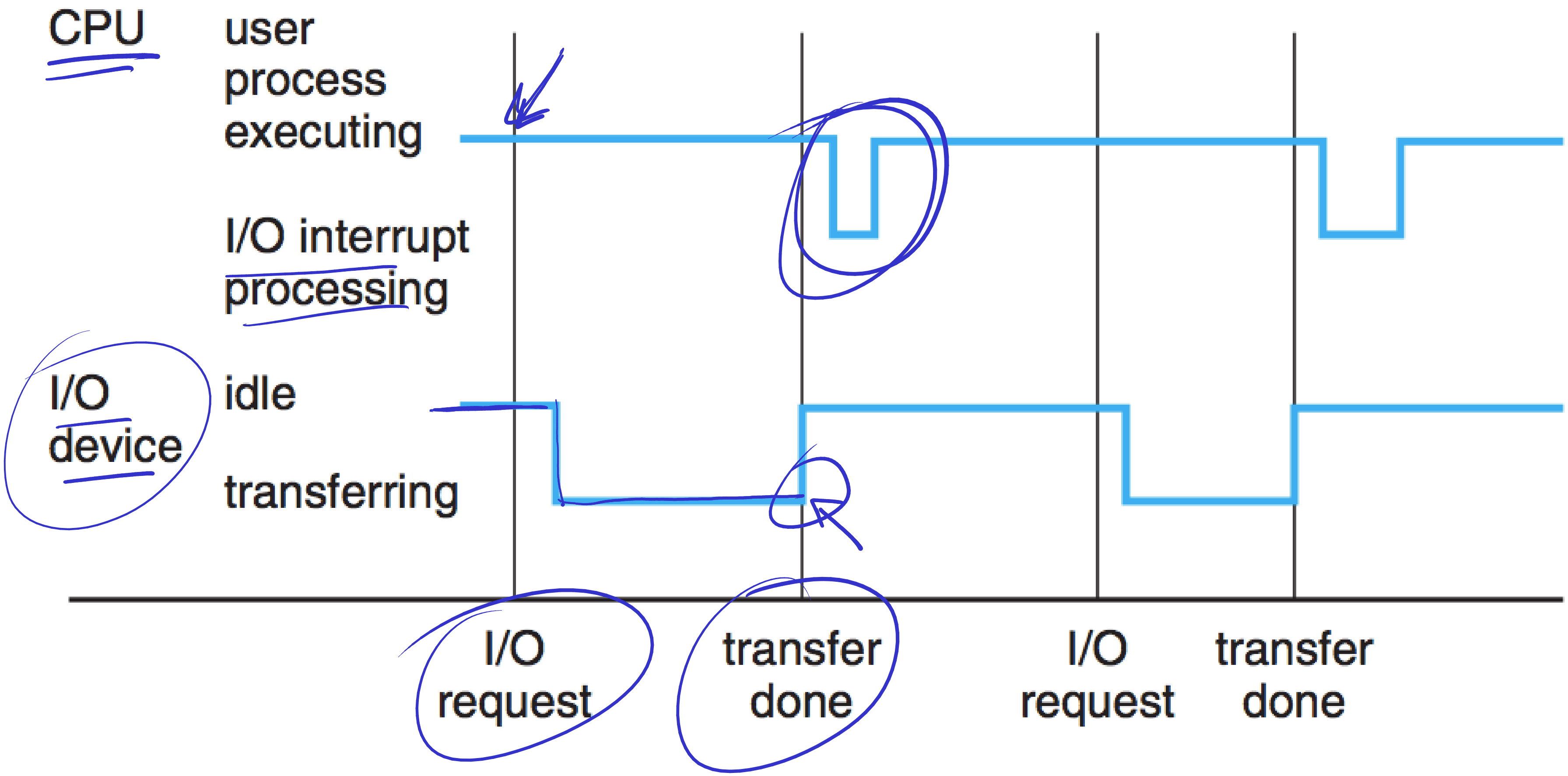
Something happens …

- **Examples:** A user clicks a mouse or a program tries to access a file
- These events are called **interrupts**
  - Can be triggered by hardware or software.
- Hardware may trigger an interrupt at any time by sending a signal to the CPU
- Software triggers an interrupt by executing a special operation
  - A **system call**

How does the CPU react?

- Stops what it is doing
- Immediately transfers execution to a fixed location
- This usually contains the starting address where the service routine for the interrupt is located
- The interrupt service routine executes
- On completion, the CPU resumes interrupted computation

Interrupt Timeline



CPU

user process executing

I/O interrupt processing

I/O device

idle

transferring

I/O request

transfer done

I/O request

transfer done

Memory

- An array of bytes, where each byte is addressable
- Read-only Memory (ROM)
- Electrically Erasable Programmable ROM (EEPROM)
- ROM cannot be modified
  - Suitable for bootstrap programs and game cartridges(!)
- EEPROM can only be changed infrequently
  - Most smartphones store factory-bundled programs on EEPROM
- CPU needs memory that it can read and write to
  - Random-Access Memory (RAM)
  - Implemented in a semiconductor technology called Dynamic RAM (DRAM)

Register / Cache Memory

- Registers are another type of memory
- Main memory goes away when machine is switched off
- Secondary storage: magnetic disks, optical disks, tapes.
- **Cache Memory**:
  - Data that has been used a lot is cached into a faster storage system
  - If CPU is looking for information, cache is first checked.

Multi-tasking in the CPU

- Eventually the first job finishes waiting and gets the CPU back.
- Time-sharing:
  - CPU executes multiple jobs by switching between them
  - Switches occur so frequently that the users can interact with each program while it is running
  - The KDF9 could handle up to 4 completely independent programs at once!

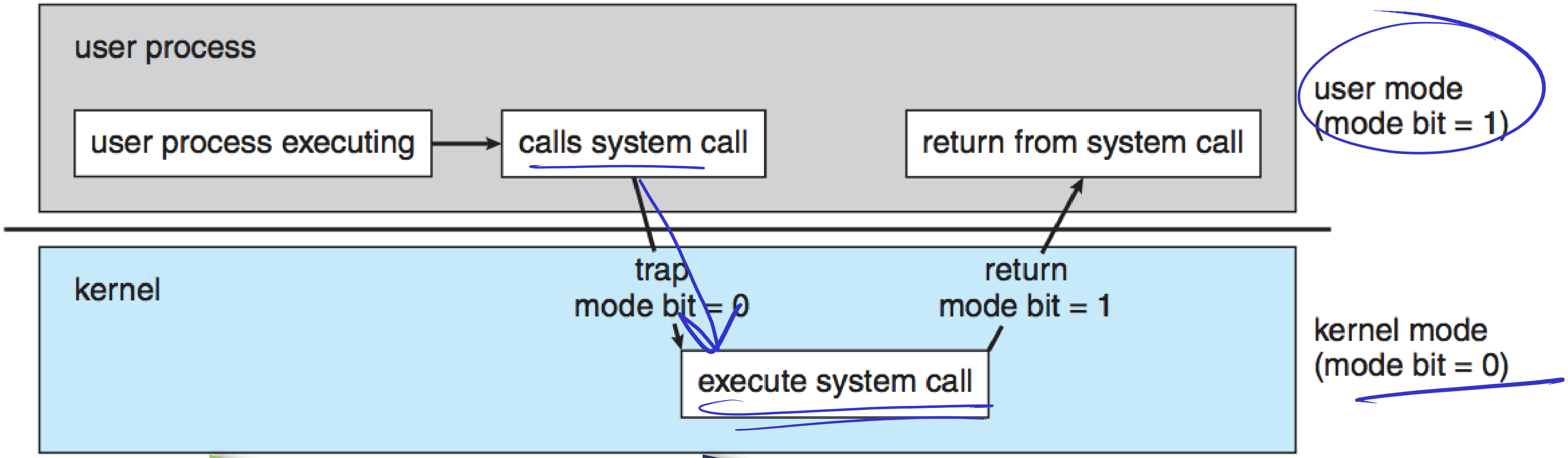
Time Sharing

- Time **sharing**: Several jobs are kept simultaneously in memory
- CPU **Scheduling**: Process of deciding which job is brought to memory to be executed when space is an issue
- Reasonable response time is of utmost importance:
  - Processes are **swapped** in and out of main memory to the disk
  - **Virtual memory** is used. This technique allows the execution of a process that is not completely in memory!

Time Sharing

- Virtual memory scheme enables users to run programs that are larger than actual **physical memory**
- It abstracts main memory into a large, uniform array of storage, separating **logical memory** as viewed by the user from physical memory
- This frees programmers from concern over memory-storage limitations!

Dual Mode



User Mode

user program executes in user mode

certain areas of memory are protected from user access

certain instructions may not be executed

Kernel Mode

monitor executes in kernel mode

privileged instructions may be executed

protected areas of memory may be accessed

Why do we need Dual Mode?

- MS-DOS: Intel 8088 architecture which has no mode bit
- User program can therefore wipe out the whole OS
- Programs are able to write to a device
- In dual mode: Hardware detect errors that violate modes and handle them with the help of OS

Why do we need Dual Mode?

- Stops user program attempting to execute an illegal instruction or to access memory of other users
- When detected:
  - OS must terminate the program
  - OS gives error message
  - Produces memory dumps by writing to a file. This can be checked by a user (if you’re brave!)

System Calls

- System calls are the mechanism through which services of the operating systems are sought.
- What language?
  - Typically C and C++
  - Sometimes assembly language
  - System call for reading data from one file and writing to another file:

```
$ cp file1 file2
```

System Calls

- ✓ **Open** file1
- Possible error (print, abort)
- ✓ **Create** file2 (if exists, rewrite/rename)
- Start **read** and **write** (errors: diskspace, memory stick unplugged)
- Read/write **complete**
- **Close** files
- **Are these accessed directly?** No - via API

System Calls : API Wrapper

- Application Programming Interface (API)
- Specifies a set of functions that are available to an application programmer
  - Includes the parameters passed to each function
  - Also includes the return values that the programmer can expect
- Programmer accesses API via a library of code provided by the OS
- **Example:** Windows API for Windows systems
  - **CreateProcess()** invokes the NTCreateProcess() system call in the Windows kernel. This return value 0 or 1, if error thrown.

System Calls : API Wrapper

- **POSIX API** for POSIX-based systems (UNIX, Linux & macOS)
  - Programmer accesses an API via a library of code provided by the OS
- Example: read() ✓ v = read(fd, buf, n);
- Input:
  - int fd: file descriptor to be read
  - void *buf: pointer into buffer to be read into
  - size_t count: maximum number of bytes to read
- Output:
  - number of bytes read, if successful
  - -1 if failed

System Calls : API Wrapper

Example:

- Java API for programs that run on the Java virtual machine
- getParentFile()
- invoked on a file object
- Output**:**
  - Returns the abstract pathname of this file’s parent, or null if this pathname does not name a parent directory.
  - JVM uses the OS system calls.

Why use an API?

- Why not invoke system calls directly?
  - Program portability: program can compile and run on any system that supports the API
  - System calls can often be more detailed and difficult to work with
  - Give access to high level objects (Java API)
- For example, think of interfaces in Java
- Using system calls is like implementing an interface

The Effect of a System Call

- The caller only needs to know the signature!
- Method call and parameters are passed into registers
- Values saved in memory
  - In a table or stacks
  - Addresses in registers
- Stack is preferred, as no limit to number of parameters

Summary

What elements of the OS have we looked at?

- The Role of Memory
- Multitasking / Time Sharing
- System calls
- APIs

References / Links

- Chapter # 1: **Operating System Concepts** (9th edition) by Silberschatz, Galvin & Gagne