Algorithms and Data Structures

Christian J. Rudder

October 2024

Contents

This page is left intentionally blank.

Preface

These notes are based on the lecture slides from the course:
BU CS330: Introduction to Analysis of Algorithms

Presented by:
Dora Erdos, Adam Smith
With contributions from:
S. Raskhodnikova, E. Demaine, C. Leiserson, A. Smith, and K. Wayne

Please note: These are my personal notes, and while I strive for accuracy, there may be errors. I encourage you to refer to the original slides for precise information.
Comments and suggestions for improvement are always welcome.

Prerequisites

— 1 —

Memory Management

1.1 CPU Arichitecture

This section provides a high-level overview of the CPU to provide context/motivation for the following algorithms and data structures.

Definition 1.1: Central Processing Unit (CPU)

The **CPU (Central Processing Unit)**, is a hardware component that *computes* instructions within a computer. Abstract models that define interfaces between hardware and software for a CPU are called **instruction set architectures (ISA)**.

Possible operations are detailed as **opcodes** (operation codes), which are numeric identifiers for each instruction. Moreover, the ISA defines supported data types, **registers (temporary storage locations)**, and addressing modes (ways to access memory).

ISA's are defines the instruction set, which allows for flexibility in hardware performance needs. This various categories:

- **CISC (Complex Instruction Set Computing):** Large number of complex instructions (multiple operations per instruction).
- **RISC (Reduced Instruction Set Computing):** Small set of simple/efficient instructions.
- **VLIW (Very Long Instruction Word):** Enables instruction parallelism (simultaneous execution).
- **EPIC (Explicitly Parallel Instruction Computing):** More explicit control over parallel execution.

Smaller more theoretical architectures exists such as **MISC (Minimal Instruction Set Computing)** and **OISC (One Instruction Set Computing)**, which are not used in practice. Popular CPU architectures include x86_64, and ARM64 (64-bit), originating from x86 and ARM (32-bit).

The implementation of a CPU on a circuit board is called a **microprocessor**. Multiple CPUs on a single circuit board are **multi-core processors**, where each *core* is a fully functional CPU.

Definition 1.2: CPU Anotomy

The CPU is comprised of three main components:

- **ALU (Arithmetic Logic Unit):** Performs arithmetic and logical operations (e.g., addition, subtraction, AND, OR).
- **Control Unit (CU):** Directs the operation of the CPU, fetching and decoding instructions, and controlling the flow of data.
- **Memory Unit (MU):** Manages data storage and retrieval, including registers and cache memory.

All these components have volatile memory, lost when the computer is turned off.

Definition 1.3: CPU Execution Flow

The **CPU execution flow** is the sequence of operations that the CPU performs to execute a program. It typically follows these steps:

1. **Fetch:** Fetches the next instruction from memory.
2. **Decode:** Decodes the fetched instruction to associated opcode and operands.
3. **Execute:** Perform decoded operation using the ALU or other components.
4. **Store:** Save results of the operation back into memory or registers.

This cycle is repeated until the program completes or an interrupt occurs.

Definition 1.4: Registers

Registers are small, high-speed storage locations within the CPU that hold data temporarily during execution. Common types of registers include:

- **General-Purpose Registers (GPR):** Hold general data storage and manipulation.
- **Special-Purpose Registers:** For specific functions, such as a reference to the current line of code.
- **Floating-Point Registers:** Floating-point arithmetic (e.g., decimal numbers).

Registers are faster than main memory (RAM) and are used to store frequently accessed data during program execution.

The following is an example of the primary registers in the x86-32 (IA-32) architecture, which is a CISC architecture.

Register	Size	Purpose
EAX	32-bit	Accumulator (arithmetic / return value)
EBX	32-bit	Base register (data pointer)
ECX	32-bit	Counter (loops, shifts)
EDX	32-bit	Data register (I/O, multiply/divide)
ESI	32-bit	Source index (string / memory ops)
EDI	32-bit	Destination index (string / memory ops)
EBP	32-bit	Base/frame pointer (stack-frame anchor)
ESP	32-bit	Stack pointer
EIP	32-bit	Instruction pointer (program counter)
EFLAGS	32-bit	Flags / status register (ZF, CF, OF...)

Table 1.1: Primary registers of the x86-32 (IA-32) architecture. **Note:** Registers are prefixed with 'E' for 32-bit, 'R' for 64-bit in x86-64.

Definition 1.5: Machine Code & Compilation

Code is separated into two main areas of memory management, the program itself, and the data in transit during execution. The program itself is broken up such as follows:

- **Text Segment:** The part of the program which contains the executable code.
- **Data Segment:** The part of the program which contains global and static variables.
- **Machine Code:** The compiled code of the program, which is executed by the CPU.

Once the code compiles, our data segment is further divided into two parts in memory:

- **Initialized Data:** Data given a value before the program starts (global variables).
- **Uninitialized Data:** Data yet to be assigned (local variables), which are zeroed at program start.

By memory we mean the **RAM (Random Access Memory)** hardware component, which stores temporary data, constantly communicating with the CPU or external storage (e.g., hard drive, SSD). Each memory cell is IDed by a unique monotonic **address**, often in hexadecimal format(e.g., 0xF00, 0xF01, etc).

Definition 1.6: Operating System (OS)

Implemented ISAs only provide an interface to the CPU; Programmers must design how their systems utilize the CPU (e.g., file and memory management), such software is called an **operating system (OS)**.

Tip: In an analogous sense, say we have a train riding service. The ISA would be the specifications of the trains, rails, routes, and stations needed. The physical implementation of trains, rails, and stations would be the CPU. The OS would be the train schedule system, managing external factors such as workers and other tasks effecting the train service.

Definition 1.7: The Kernel

The **kernel** is a **process** (a program) vital for OS operation, always running with the highest priority. It is the only program that can directly interact with the CPU and various hardware components.

Other processes running on the system are called **user processes**. This is where applications and other user-level programs run. If a user wishes to perform a task that requires hardware access (e.g., writing/reading files), they must request the kernel called a **system call (syscall)**. System calls provide an **Application Programming Interface (API)** for user processes to interact with the kernel.

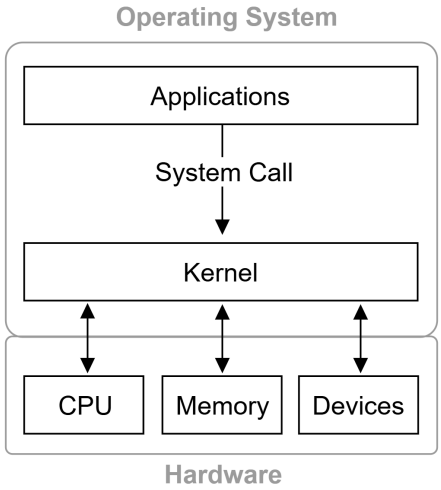

Figure 1.1: User-level applications make syscalls to the kernel to access hardware resources.

Definition 1.8: Bus

A **bus** is a collection of physical signal lines (wires or pins) and protocols that carry data, addresses, and control signals between components inside a computer (e.g. CPU, memory, I/O devices) or between multiple boards and peripherals. There are two main types of buses:

- **Serial Bus:** Transfers data one bit at a time over a single channel (e.g., USB).
- **Parallel Bus:** Transfers multiple bits simultaneously over multiple channels (e.g., PCI).

Definition 1.9: Device Drivers

The kernel exposes generic interfaces to various sub-systems (e.g., file system) that user processes can use to perform tasks; **Device drivers** implement such interfaces, translating generic system calls into hardware-specific operations for specific devices (e.g., disk drives, network cards, etc.). Drivers must be loaded into kernel space.

This text does not concern assembly code, so **do not** get caught in the specifics of this Example:

Example 1.1: Assembly Code

An assembly example demonstrating initialized (.data) and uninitialized (.bss) data sections:

```
section .data                   ; Initialized data section
    num1    dd  7               ; num1 is initialized to 7
    num2    dd  3               ; num2 is initialized to 3

section .bss                    ; Uninitialized data section
    temp    resd 1              ; temp is reserved (uninitialized)
    result  resd 1              ; result is reserved (uninitialized)

section .text                   ; Code section
    global _start

_start:
    mov eax, [num1]             ; Load num1 into eax
    mov [temp], eax             ; Store num1 in temp
    mov ebx, [num2]             ; Load num2 into ebx
    add eax, ebx                ; Add num2 to eax (eax = num1 + num2)
    mov [result], eax           ; Store the sum in result
; Exit syscall removed for simplicity
```

In this example, 'num1' and 'num2' are initialized before execution, while 'temp' and 'result' are uninitialized and only receive values during program execution. ■

1.2 Code Security

At a high-level, vulnerabilities exploited by hackers stem from flaws that the original programmer forgot to consider (i.e., bugs). To learn more on cybersecurity, consider our other text here.

Definition 2.1: Proper Encapsulation

Proper encapsulation is the practice of hiding implementation details and exposing only necessary interfaces to prevent unauthorized access or modification.

Example 2.1: Student Class

Consider a simple 'Student' class in an object-oriented programming language:

```
public class Student {
    private String name; // Private field, not accessible outside
        the class
    private int age;     // Private field, not accessible outside
        the class

    public Student(String name, int age) {
        this.name = name;
        this.age = age;
    }

    public String getName() { // Public method to access name
        return name;
    }
    // Other methods...
}
```

Upon creating a new student instance `new Student("Alice", 20)`, the name and age are private, preventing direct access via **dot notation** (e.g., `student.name`). The only way to access the name is through the public method `getName()`. Here we do not have a method for accessing age. ■

Definition 2.2: Risks of Accessing Main Memory

Programs accesses main memory (RAM) to read and write data; It's critical that such references to RAM are abstracted to avoid malicious or accidental access of data.

For example, in Java when users print objects, instead of printing the object's memory address, it prints the `toString()` method, which by default prints the class name and hash code of the object.

Bibliography