I. A Tour of Computer Systems

§1.1 Information Is Bits + Context

- All information in a system is represented as a bunch of bits
  - In different **contexts**, the same sequence of bytes might represent an integer, floating-point number, character string, or machine instruction.
- Need to understand machine representations of numbers
  - not same as integers and real numbers
  - finite approximations that behave in unexpected ways

§1.2 Programs Are Translated by Other Programs into Different Forms

- On Unix system, ***Compiler driver*** translates from source file to object file

```
linux> gcc -o hello hello.c
```

- Translation is performed in the sequence of **4 phases**
  1. **Preprocessor** (cpp) (.c -> .i)
     - Preprocessing phase
       - modifies the original C program according to directives that begin with the '#' character
         - e.g. #include <stdio.h>
       - inserts header files directly into the program text

  2. **Compiler** (cc1) (.i -> .s)
     - Contains ***assembly-language program***
       - e.g.

```
main:
  subq  $8, %rsp
  movl  $.LC0, %edi
  call  puts
  movl  $0, %eax
  addq  $8, %rsp
  ret
```

- each line describes one low-level machine language instruction in a **textual form**
- provides a **common output language** for different compilers for different high-level languages

3. **Assembler** (as) (.s -> .o)
    - translates .s into machine-language instructions and packages them in a form ' *relocatable object program* ' **(binary)**

4. **Linker** (ld) (.o + .o + .o + ... -> executable object program)
    - **merges** several precompiled object files
    - creates an executable object file that is ready to be loaded into memory and executed by the system

§1.3 It Pays to Understand How Compilation Systems Work

- Optimizing program performance
- Understanding link-time errors
- Avoiding security holes
    - *buffer overflow vulnerabilities & stack discipline*

§1.4 Processors Read and Interpret Instructions Stored in Memory

- **Shell** (Command-line interpreter)
    - prints a prompt
    - waits for a command line
    - performs the command
    - **IF** the first word of the command line does not correspond to a built-in shell command, then the shell assumes that it is an **executable file** that it should load and run

¶1.4.1 Hardware Organization of a System

- **Buses**
  - carry bytes of information between the components
  - transfer fixed-size chuncks of bytes (words)
    - either 4 bytes (32 bits) or 8 bytes (64 bits)
- **I/O Devices**
  - system's connection to the external world
  - connected to the I/O bus by a **controller** or an **adapter**
    - controller: chipsets in the device itself or the system's main circuit board
    - adapter: card that plugs into a slot on the motherboard
- **Main Memory**
  - **temporary storage device** that holds both a **program** and the **data** it manipulates while the processor is executing the program
  - Collection of *dynamic random access memory* (**DRAM**)
  - organized as a linear array of bytes with its own unique address
- **Processor**
  - *central processing unit* (**CPU**)
    - *program counter* (**PC**): word-size storage device at CPU's core
      - points at some machine-language instruction in main memory
    - ***register file***: a small storage device that consists of a collection of word-size registers each with its own unique name
    - *arithmetic/logic unit* (**ALU**): computes new data and address values
  - **interprets** (executes) instruction stored in main memory
  - repeatedly **executes the instruction pointed at by the program counter** and updates the program counter to point to the next instruction
  - operates according to CPU's ***instruction set architecture***
  - Examples of the simple operations
    - **Load** : copy a byte or a word from main memory into a register, overwriting the previous contents of that location
    - **Store** : copy a byte or a word from a register to a location in main memory, overwriting the previous contents of that location
    - **Operate** : copy the conents of two registers to the ALU, perform an airthmetic operation on the two words, and store the result in a register, overwriting the previous contents of that register
    - **Jump** : extract a word from the instruction itself and copy that word into the

program counter (PC), overwriting the previous value of the PC

§1.5 Caches Matter

- system spends a lot of time moving information
    - originally, disk -> main memory -> processor
- larger storage devices are slower than smaller storage devices (due to physical laws)
- faster devices are more expensive to build
- processor-memory gap continues to increase
    - easier & cheaper to make processors run faster than make main memory run faster
- **Cache Memories**: temporary staging areas for information
    - added to deal with the processor-memory gap
    - ***Static random access memory***
        - **L1 cache**: holds x0,000 bytes & can be accessed nearly as fast as the register file
        - **L2 cache**: ~x,000,000 bytes, takes 5 times longer for the processor to access L2 than to L1, but still 5-10 times faster than accessing the main memory
            - connected to the processor by a special bus
        - Newer systems may have L3 cache as well
    - ***Locality***
        - very large memory + speed
        - Set caches to hold data that are likely to be accessed often

§1.6 Storage Devices From a Hierarchy

- similar to pyramid
- Top -> Bottom
    - slower, larger, less costly per byte
    - **Register file** - (**L0**)
    - **Caches** - **L1 - L3**
    - **Main memory** - **L4**
    - **Local Disks** (Local secondary storage) - **L5**
    - **Remote secondary storage** (distributed file systems, web servers) - **L6**
- Storage at one level **serves as a cache for storage at the next lower level****
    - register file is a cache for the L1 cache..

§1.7 The Operating System Manages the Hardware

- When the shell ran the program, neither program accessed the keyboard, display, disk, or main memory directly
- **Operating System**:
  - a layer of software interposed between the application program and the hardware
  1. to protect the hardware from misuse by runaway applications
  2. to provide applications with simple and uniform mechanisms for manipulating complicated and different low-level hardware devices
  - uses fundamental abstractions: ***processes, virtual memory, files***

Layered view of a computer system	
Application programs	SW
Operating system	SW
Processor & Main memory & I/O devices	HW

```
--------------------------- Processes ------------------------------
|                                                                   |
|                    -------------- Virtual memory -----------------|
|                   |                                               |
|                   |                        --------- Files -------|
|     Processor     |      Main memory       |     I/O devices      |
--------------------------------------------------------------------
```

¶1.7.1 Processes

- ***Process***: operating system's abstraction for a running program
- Multiple processes can run **concurrently** on the same system with exclusive use of the hardware
  - instructions of one process are interleaved with the instructions of another process
- In general, there are more processes to run than there are CPUs to run them
- Traditional systems could only execute one program at a time

  <--> Newer ***multi-core*** processors can execute several programs simultaneously
- ***Context switching***: used to perform interleaving

- OS keeps track of all the state information that the process needs in order to run
  - **Context** = current values of the PC / register file / contents of main memory
- OS performs **context switch** by saving the context of the current process, restoring the context of the new process, and then passing control to the new process.
  - Invokes a ***System call*** : passes control to the OS
- Transition from one process to another is managed by the ***OS kernel*** (always resident in memory)
  - When an application program requires some action by the OS (e.g. read, write), executes a special ***system call*** instruction, transferring control to the ***kernel***
  - Kernel is not a separate process

¶1.7.2 Threads

- In modern systems, a process can consist of multiple ***threads*** (execution units)
  - each thread runs in the context of the process while sharing the same code and global data
- Threads are required for **concurrency** in **network** servers
  - easier to share data between multiple threads than between multiple processes
  - threads are more efficient than processes

¶1.7.3 Virtual Memory

- ***Virtual memory***: provides each **process** with the illusion that it has **exclusive use of the main memory**
  - ***Virtual address space*** : process has the same uniform view of memory

Figure. Process virtual address space
Kernel virtual memory
User stack (created at run time)
^, v
Memory-mappeed region for shared libraries
^
Run-time heap (created by malloc / calloc)
Read/write data
Read-only code and data

  - In Linux,
    - topmost region = code, data common to all processes
      - lower region = code, data defined by user's process
- Virtual address space (bottom -> top)
  - ***Program code and data***
    - begins at the same fixed address for all processes
      - followed by data locations to global C vars
  - initialized directly from an executable object file
  - ***Heap***
    - **expands** and **contracts** dynamically at run time as a result of calls to C standard library routines
      - malloc, free, etc..
  - ***Shared libraries***
    - e.g. C standard library, math library
      - #include , #incldue
  - dynamic linking
  - ***Stack***
    - used to implement **function calls**

        - **expands** and **contracts** dynamically during the execution of the program
            - grows each time we call a function
        - contracts each time we return from a function
    - ***Kenrel virtual memory***
        - Application programs **are not allowed** to read or write, or to directly call functions defined in the kernel code
    - must invoke the kernel to perform operations

¶1.7.4 Files

- ***File***: sequence of bytes
- Every I/O device (disks, keyboards, displays, networks ...) is modeled as a file
- All input and output in the system is performed by reading and writing files
    - ***Unix I/O*** system calls
    - Provides applications with a uniform view of all the varied I/O devices
    - enables the same program to run on different systems that use different disk technologies

§1.8 Systems Communicate with Other Systems Using Networks

- Modern systems are linked to other systems by **networks**
- **Network** can be viewed as another **I/O device**
- Example) Using telnet to run *'hello'* remotely
    1. User types "hello" at the keyboard
    2. Client sends "hello" string to telnet server
    3. Server sends "hello" string to the shell, which runs the *'hello'* program and passes the output to the telnet server
    4. Telnet server sends "hello, world\n" string to client
    5. Client prints "hello, world\n" string on display

§1.9 Important Themes

¶1.9.1 Amdahl's Law

- effectiveness of improving the performance of one part of a system
- when we speed up one part of a system, the effect on the overall system performance depends on both **how significant** this part was and **how much it sped up**

Suppose,

T_{old} = time required to execute some application

α = fraction occupied by some part of the system

k = performance improve factor

Then,

$$T_{new} = (1-\alpha)T_{old} + (\alpha T_{old})/k$$
$$= T_{old}[(1-\alpha) + \alpha/k]$$

Hence,

$$S = \frac{1}{(1-\alpha)+\alpha/k}$$

Special case exists when $k = \infty$ (sped up to the point at which it takes a negligible amount of time)

$$S_\infty = \frac{1}{(1-a)}$$

¶1.9.2 Concurrency and Parallelism

- ***Concurrency*** : general concept of a system with multiple, simultaneous activities
- ***Parallelism*** : use of concurrency to make a system run faster

1. **Thread-level Concurrency**
    - With threads, can have multiple control flows executing wihtin a single process
    - ***Uniprocessor system***
        - thread-level concurrency was ***'simulated'*** by having a single computer rapidly switch among its executing processes
            1. multiple users can interact with a system at the same time
            2. single user can engage in multiple tasks concurrently
    - ***Multiprocessor system***
        - system consisting of **multiple processors** under the control of a **single operating system** kernel
        - **Multi-core processors** (e.g. Intel i7)
            - chip has 4 CPU cores
                - each has its own L1 and L2 caches
                    - 2x L1 caches - instruction, data
                - share higher levels of cache & interface to main memory

- **Hyperthreading**
    - *simulatneous multi-threading*
    - allows a **single CPU to execute multiple flows** of control
    - having multiple copies of some of the CPU hardware (program counters, register files), while having only single copies of other parts of the hardware
    - decides which of its threads to execute **on a cycle-by-cycle basis**
        - conventional processor requires ≈20,000 clock cycles to shift between threads
        - takes better advantage of its processing resources
    - e.g. Intel i7 processor can have each core executing two threads ==> 2 x 4 = 8 threads in parallel
- Multiprocessing can improve system performance by...
    1. reducing the need to simulate concurrency when performing multiple tasks
    2. running a single application program faster (but only if the program is expressed in terms of multiple threads that can effectively execute in parallel)

2. **Instruction-Level Parallelism**

- modern processors can **execute multiple instructions** at one time
    - 2-4 instructions per clock cycle (<--> early microprocessors required 3-10 clock cycles to execute a single instruction)
    - can process ≈100 instructions at a time
- ***Pipelining***
    - **actions** required to execute an instruction are **partitioned into different steps** and the processor **hardware** is organized as a **series of stages**, each perfroming one of these steps
    - **stages can operate in parallel**
        - on different parts of different instructions
- Processors sustaining execution rates faster than 1 instruction per cycle = ***superscalar processors***

3. **Single-Instruction, Multiple-Data (SIMD) Parallelism**

- modern processors have special hardware that allows a **single instruction** to cause **multiple operations to be performed in parallel**
    - e.g. Intel, AMD processors can add 8 pairs of single-precision floating-point

numbers in parallel

    - used to speed up applications that process image, sound, and video data

H5 ¶1.9.3 The Importance of Abstractions in Computer Systems

- <u>**Program side**</u>
    - **simple application program interface (API)**
        - allows users to use the code without having to delve into its inner workings
- <u>**Processor side**</u>
    - ***instruction set architecture***
        - provides an abstraction of the actual processor hardware
        - a machine-code program behaves as if it were executed on a processor that performs one instruction at a time
- <u>**Operating system side**</u>
    1. **files** <- I/O devices
    2. **virtual memory** <- program memory
    3. **processes** <- running program
    4. ==**virtual machine**== <- entire computer