Computer Architecture

6.100 LECTURE 25

SPRING 2026

Announcements

- Pset 7 due next tonight at 10 pm
- checkoff due next Tuesday 5/12 during office hours

- Exam 3
  - overall class performance improved
  - same letter grade ranges
  - will release scores & solutions this evening
  - regrade requests open tomorrow 5/7 at 6 pm until Sunday 5/10 at 6 pm

Memory & Processor

Storing and retrieving data

- **To store a bit,** use capacitor, store charge
- **To retrieve a bit,** measure voltage
  - measuring drains the charge, need to rewrite bit
- Abstraction of a memory cell
  - control line – read or write command
  - data-in line – low (0) or high (1) voltage
  - data-out line – low (0) or high (1) voltage
  - often more efficient to read/write a contiguous block of memory cells at once

Steps to process instructions

- Recall **Turing machine**
  - read data on tape
  - depending on internal state
    - write data
    - move tape
    - update state
- In terms of modern architecture
  - tape is like **memory**
  - internal state is like **processor state**

Executing a sequence of instructions

- Program is actually stored in memory (separate from stack and heap) as a **sequence of instructions**
- Typical instructions
  - read this memory address
  - add/multiply these two numbers
  - write this memory address
  - **move/jump** to this next instruction
- Typical processor pipeline
  - fetch instruction from memory
  - **control unit** interprets instruction type
  - populate **registers** with data (e.g., contents at a memory address or memory addresses themselves)
  - **arithmetic/logic unit** operates on registers
  - control unit writes result back to memory

Operating System

Where data comes from and goes to

- **Input devices**
  - storage
  - keyboard
  - network, wireless
  - camera, microphone
  - usb devices
- **Output devices**
  - storage
  - display, sound
  - network
  - printer
- How is a program or a processor supposed to interact with them?
  - each of these has specialized circuitry
  - **operating system** abstracts them away

(Very) simplified view of one aspect of OS kernel

- When a program says **"print this data to screen"**
  - programming language translates that into a **system call**
  - translate each character into pixels
  - pixels are updated in a **frame buffer**
  - every 1/60th of a second, buffer gets **copied/flushed/serialized across wire** to display device
  - display receives new frame, unpacks, applies voltages to pixels
- When a program says **"read this file from disk"**
  - translate into **system call**
  - trace file path to identify file on **filesystem** (path in a graph!)
  - map that file to a **location/sector/address on disk**
  - prepare command to disk device, send over wire
  - disk receives command, unpacks, runs its own controller to seek and read data
  - serializes that data over the wire again, arrives in a location in memory
  - **system call returns with pointer to that place in memory**

Device drivers

- A lot of **"low-level programming"** goes into talking to hardware/devices
  - compute addresses
  - pack data into lists/arrays
  - look up command codes
- Nominally falls on the processor to perform these tasks
  - so need subprograms to accomplish them
  - these subprograms are called **drivers**
- A driver is specific to a given **(device, processor, OS) combination**

Running Programs

Translating code into processor instructions

- **Example: multiply matrices**
  - Python code doesn't translate directly to processor instructions
  - show equivalent **C++ code** for now
- Compiler converts code to linked object file of processor instructions
  - **`$ g++ matrix_multiply.cpp`**
  - produces executable **`a.out`** file
- When we run the .out file, the OS loads the instructions into memory and points the processor to start running those in sequence
- **Assembly code** is "human-readable" representation of processor instructions
  - **`$ g++ -S matrix_multiply.cpp`**
  - produces human-readable **`matrix_multiply.s`** file
  - color-coded inspection: **Compiler Explorer** https://godbolt.org/
- While overall computer architecture principles are common, specific implementations differ across operating systems and hardware
  - hence why **Windows, macOS, iOS, Android, etc.** need different object/assembly code for the same program

How Python programs run

- Python belongs in a class of languages that are supplied with an intermediate **"virtualized" machine**
  - **interpreter** is a compiled program that simulates a machine
  - release **different interpreter variants** for **different architectures**
  - **Python code behaves the same on each variant**
- Matrix multiplication is much slower
  - processor isn't running Python code directly
  - it's running the Python interpreter, which is a lot of machinery
  - actual running of Python code is small portion of the time
- **NumPy** tries to address this by implementing common computations tasks in C/C++
  - access these library functions through **Python wrappers**
  - even faster than our C++ implementation due to **parallelism**
    - **vectorized** operations
    - writing to different parts of memory simultaneously, **multiprocessing**
- Much of Python is written this way for **built-in types**
  - operations are fast
  - but interpreting them in the context of Python is slow

Caching memory

- **Memory access is slow** compared to processor cycles
  - not efficient to fetch one bit or even one byte at a time
  - an `int` is typically 4 or 8 bytes
  - memory access needs to travel wires
  - goes through additional drivers, chip controllers, etc.
- When running C++ program to multiply matrices, lists/arrays are laid out contiguously
  - easier to fetch **entire "pages" of memory** in on the order of kilobytes
  - easy to build circuitry to support that
  - **store fetched page in a cache close to processor/registers**
  - write entire pages back
- We can directly "perceive" the memory accesses in C++ code
- But Python code is running a separate **interpreter program**
  - **data for the interpreter** may live far away from the **actual Python data** our Python program is operating on
  - results in **thrashing** as pages get repeatedly swapped in and out of cache

Next time

- **Today**
  - what hardware and low-level software supports the execution of a program
- **Next Monday**
  - landscape of other programming languages
  - how to learn them given your knowledge from 6.100