Lecture 9 - Computer Memory

Eric A. Autry

Last time: Midterm

This time: Computer Memory

Next time: Review of Data Structures

Midterm will be returned today after lecture.

A Binary Tape

Now that we have developed the theory of automata, how might we actually build one?

Electrical wires can track high and low voltage.

This corresponds to a binary tape of 0s and 1s.

- Store numbers in binary on the tape.
- Convert other symbols to numbers for storage on the tape (unicode/ascii).

A Binary Tape

When we have multiple pieces of data on the tape, how to tell when one ends and the other begins?

Before we could use a mark like $\#$.

But we don't have a third symbol on our binary tape.

Instead, we define a fixed length for certain objects. Ex: a byte is 8 bits, an int is 32 bits.

Machine Registers

To build our computer, we start with a central processor that will control the read/write head and the states of our machine.

We add a very small amount of fixed storage space at the front of the tape reserved for operations of the processor.

This storage space is called the machine's registers, has very fast access, and can allow for basic addition of registers, multiplication of registers, swapping of registers, etc.

Storing an Array

What if I want to store an array of values? Say I want the array of bytes:

$$a = [1,\ 2,\ 3].$$

Each byte is 8 bits, and a is 3 bytes long, giving us 24 bits total:

$$\sqcup\sqcup\sqcup\sqcup\ \sqcup\sqcup\sqcup\sqcup\ \ \dot{0}000\,0001\ 0000\,0010\ 0000\,0011\ \ \sqcup\sqcup\sqcup\sqcup\ \sqcup\sqcup\sqcup\sqcup$$

We access this data by storing the location of the front of the array on the tape, which is a pointer to the array.

We can index into the array using the knowledge that it is an array of bytes:

- $a[0]$: shift location by 0 bits, count of the next 8 bits
- $a[1]$: shift location by $1 \times 8 = 8$ bits, count of the next 8 bits
- $a[2]$: shift location by $2 \times 8 = 16$ bits, count of the next 8 bits
- $a[3]$: shift location by $3 \times 8 = 24$ bits, count of the next 8 bits - out of bounds on the array!

Storing Programs

We can store programs on the tape as compiled binary functions.

So the list of instructions are stored at specific locations on the tape.

The machine has a special reserved instruction pointer (aka program counter) that tracks which instruction is next.

Malicious attacks can try to overwrite the instruction pointer so that the machine begins running outside code.

The Stack

Where can each function store its local memory (the local variables it uses during computation)?

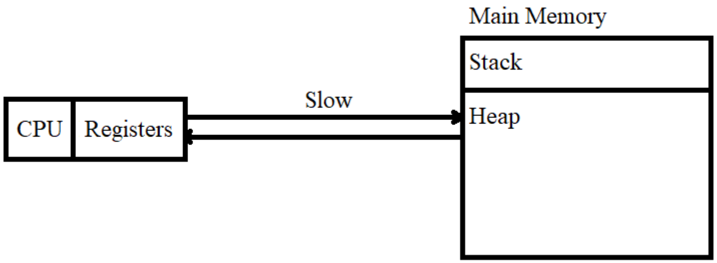
- Each function is allocated space on the stack, a reserved area near the top of the tape.
- The stack is Last In First Out order for allocation/deallocation. This requires storing just one pointer for where we are in the stack. Fast allocation.
- Want to change a variable size dynamically? You cannot (would collide with another function's space on the stack).

The Heap

Store dynamically (changing in size) allocated memory on the heap, a large section of the tape further from the head.

- Each dynamically allocated object could change in size.
- So need complex memory management to ensure data is not overwritten, and garbage collection to free up space on the tape.
- Slower memory access due to the complex bookkeeping.

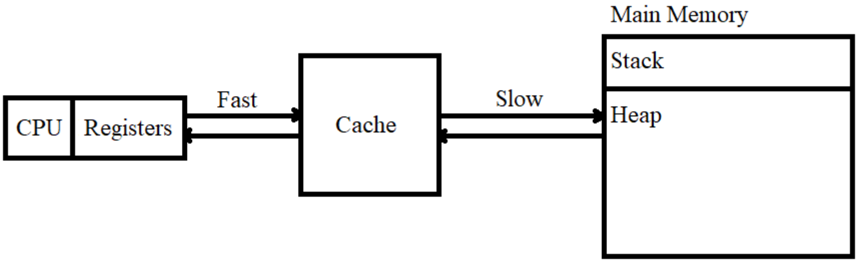
Caching



Can we have faster memory access?

Idea: put a cache of memory in between.

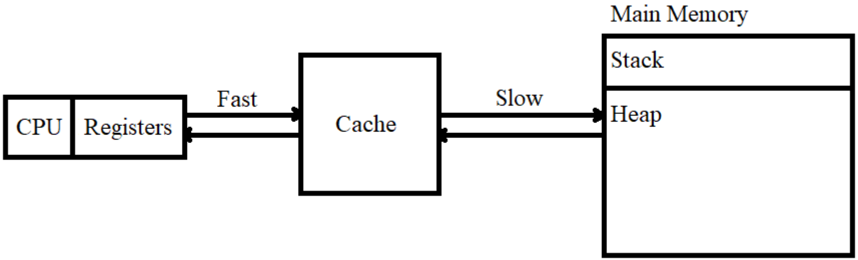
Caching



If some piece of data gets used a lot, we prefer to have it stored in the fast access Cache instead of the slow access Disk.

We load ‘blocks’ from Disk into Cache, then load individual ‘words’ from the Cache to the Registers.

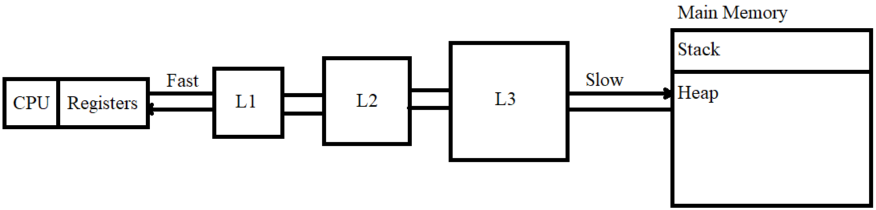
Caching



Least Recently Used (LRU) Algorithm: when overwriting a block stored in Cache, always replace the block in the Cache that has been unused the longest.

This requires hardware that can track which block was last used. Another option is to simply dump and refill the entire Cache.

Caching



We can even allow for multiple levels of Caching, with increased access times as we get closer to the CPU.

Caching

Why is this important?

Let's look at a matrix stored as a 2D array:

$$A = [\,[1,2,3],\ [4,5,6],\ [7,8,9]\,]$$

Consider the following code:

```
for r in rows:
    for c in columns:
        print A[r][c]
```

```
for c in columns:
    for r in rows:
        print A[r][c]
```

Which will run faster? What gets loaded into Cache when?

The code on the right wipes the Cache an loads a new row on each iteration of the inner loop!

Midterm Return

Grading Philosophy: Exam grades **do not represent** the overall course grade.