Lecture #6: Computer Hardware (Memory)

CS106E Spring 2018, Young

In this lecture we explore computer memory. We begin by looking at the basics. All memory is numbered, with each byte having its own address. We take a look at the Random Access Memory (RAM) that composes most of main memory. We also consider some other items that may appear in the main memory address space, such as special access to Input/Output devices using Memory Mapped IO.

Next, we explore the sections of memory that are used by a typical program. We take a closer look at the Call Stack used to store parameters and local variables when functions are called and we study the Heap which is used to store objects. As we will see, improper use of the heap may lead to Memory Leaks. Some languages avoid this by managing memory for a programmer and use Garbage Collectors to ensure no memory is leaked.

Modern CPUs in addition to the registers we studied last lecture contain some additional memory – Cache memory. We take a quick look at CPUs' L1, L2, and L3 Caches and also how some Solid State and Hard Drives use similar Memory Caches.

We end by taking a look at how Read-Only Memory (ROM) can be used in the startup sequences of computers and how it provides Firmware instructions for various consumer devices.

Hexadecimal Numbers

Hexadecimal Numbers are often used in Computer Science to represent binary numbers.

- Hexadecimal is based on 16, whereas as we've previously seen Binary is based on 2 and Decimal is based on 10.
- As we don't actually have 16 digits to use for Hexadecimal, we use the regular Decimal digits 0-9 and then add the letters A, B, C, D, E, and F.

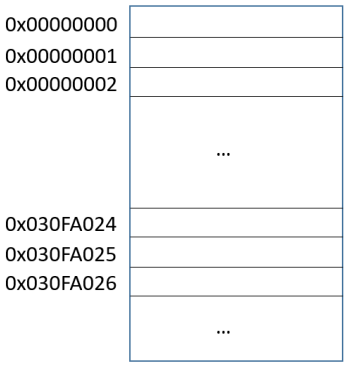
- I've put out a separate handout specifically discussing the user of Hexadecimal Numbers. Read it for more information on how they work and why Computer Scientists like using them.
- In this document, I'll be using Hexadecimal to represent memory addresses. The fact that these numbers are preceded by a 0x indicates that they are Hexadecimal.

Memory Addresses

- Computer Main Memory consists of sequential numbered bytes.
- The numbers for each byte is called an ***Address***. Internally addresses are represented in binary, of course, which is why computer memory is always purchased and installed in powers of 2.

- You'll often see abstract diagrams of computer memory drawn something like this.
    - What we're seeing is the sequence of bytes in the computer.

- The first byte has the address 0x00000000, the second byte has the address 0x00000001, and so on.
- The ellipsis in the middle of the block indicates that there are a lot of bytes, and after that sequence of many, many bytes, we reach a set of bytes somewhere in the middle of memory that we are interested in or where something is being stored.
    - In this particular case, that memory location we are interested in is located at 0x030FA024. I've chosen to show that that memory location is followed by 0x030FA025 and 0x30FA026, although these may or may not be shown on a typical memory diagram, and they would likely not be given addresses, since the assumption would be that the viewer would understand that they followed sequentially after 0x030FA024.
- Note that the addresses here consist of 8 hexadecimal digits, which correspond to 32-bits. So this computer is using an address size of 32-bits.



Random Access Memory (RAM)

Most of Main Memory consists of RAM, which is short for Random Access Memory.

- The term Random Access Memory refers to the fact that we can immediately access any element in RAM memory by passing in its memory address.
    - That's in contrast with a device such as a magnetic tape, where even if we know the location of a particular byte, we have to roll through the entire tape to get to that byte. This type of access is referred to as *Sequential Access*.
- However, the "random-access" capability of RAM isn't really what distinguishes it from other types of memory. You can think of this as a vestige of older times that's now permanently embedded in the name.

- The key characteristics from our standpoint are that we can access individual bytes by immediately using their address and that *we can change the values found at that memory address*. That's in contrast with ***Read-Only Memory (ROM)*** where we can access any byte by address, but as the name implies, we cannot change the value stored at that address.

- Many different types of RAM exist, which differ in their electronic characteristics and may be faster or slower. So you may run into, for example, SDRAM, DDR SDRAM, or RDRAM. You do need to get the exact right type for your computer if you do an upgrade, but from the high-level Computer Science perspective, they all fulfill the same purpose.

Other Items in Main Memory Space

As I've alluded to, while the bulk of Main Memory is RAM, other items can be placed in the Main Memory address space these include:

Read-Only Memory (ROM) – Some computers contain memory whose values, unlike RAM memory, cannot be changed. This memory is non-volatile – its values are permanently set to a specific bit sequence. For example, the original Macintosh computers included 64kbytes of ROM, which included instructions to display windows and draw geometric shapes and text on the screen. The assumption was made that this code would always be needed by every program, therefore it was hardcoded into ROM memory chips, rather than being loaded from disk into RAM every time the computer started up.[1]

Memory-Mapped IO – As we've previously seen, the CPU doesn't have instructions dealing with IO devices such as Computer Mice, Displays, SSD drives etc. So how does it interact with and control these devices? One technique is to set aside certain memory addresses to interact with these devices. When the CPU writes to one of these special addresses, it's not actually changing the contents of RAM, instead it's communicating directly with an IO device. Whatever bits the CPU stores at that special address are actually control instructions being sent to the device, telling it what do. Similarly, the CPU can get data from such a device by reading bits from a special address.

I think it's also worth noting that the Graphics Processing Unit (GPU) has its own memory, this is separate from Main Memory. We will not be exploring the GPU memory further in this lecture.

An Aside on Functions and Methods for Java Programmers

For those of you who have programmed in Java, you may not be familiar with the name *function*. (Everyone else can skip this section).

We can think of a function as a set of code that we've given a name to and that we can call by using that name. Typically, we will be able to pass parameters in to the function and return a value from a function.

A function is very similar to a Java static method (also known as a class method). In contrast with regular methods, a function, like a static method, is not called in the context of a specific object.

I will be using the term function throughout this handout and throughout the rest of the class. You can think of Java methods as specialized functions, where the object that the method is being called on is an additional input to the function.

Sections of Main Memory

Let's consider what the RAM memory is used for from a program's perspective. While the exact breakdown may vary, we can generally divide memory into several different categories:

Code Segment – As we've discussed the actual instructions for a program must be transferred into main memory when a program executes. This section of main memory is where a program's code resides while the program is running.

Data Segment – Global variables for the program we are running are stored in this part of main memory.

[1] Originally, this was before the Macintosh had hard drives, so if the instructions had not been permanently hard-coded in ROM, they would have had to load in to RAM from a floppy disk drive.

Call Stack – When a function (or method) is called, the data associated with it is placed in the call stack.

Heap – When new objects are created they go into the heap.

Let's take a closer look at these last two sections.

Call Stack

- Modern programming languages allow a function to call itself. This is called recursion. A commonly used teaching example for showing recursion is calculating the factorial:

```
int factorial(int n) {
    if (n == 0) {
        return 1;
    }
    return n * factorial(n - 1);
}
```

- If we allow this type of code, we cannot just have a single storage location for the parameter n. Because the factorial function can call itself, we need a new copy of n for each copy of factorial that is called.
    - Suppose for example, I call factorial(3). In the middle of executing factorial(3) I call factorial (2), factorial(3) is still not done yet, so I need to remember its value of n, while factorial(2) is running.
    - But factorial(2) calls factorial(1), so now I need to remember the different values of n for factorial(3), factorial(2), and factorial(1).

- While our example factorial function has only a single parameter and no local variables, this problem will occur for both the parameters and the local variables in a recursive function.

- When a function is called, a structure called the Call Stack Frame is created. This structure includes space for all the parameters and local variables of the function this is being called. It may also contain other information such as a place for a return value, and a reference to the address where the function was called.
    - The call stack frame is placed in a structure called the call stack. It remains on the call stack until the function associated with it finishes executing.
    - Call stack frames all go into this section of memory.

- On a historic note, the earliest programming languages did not allow for recursion. They therefore did not have a call stack. Local variables and global variables could all go in the data segment.

Heap

- Modern programming languages allow us to dynamically create structures in memory.
- For example, when I write the code:

```
Student st = new Student("Molly");
```

  where is the actual data for the Student object stored? It is stored in the section of memory called the Heap.
- The *Heap Memory Manager* or *Heap Manager* keeps track of the Heap section of memory. When the program uses `new` in order to create a new object, the Heap Manager finds an unused section of memory the heap and allocates it for the new object.

- On important problem that comes from these structures is keeping track of whether or not the program still needs to remember a particular object. Consider for example the following two different scenarios:

    (1) I create the student object corresponding to "Molly". I perform some operations on this object, possibly passing it around between some different functions. After some time, I am done with this data and no longer need it.

    (2) I create the student object corresponding to "Molly". As before, I perform some operations on it. However, one of those operations is to place it in a list of students. After some time, I am done with the data for now, but the list still maintains a reference to the student object, and I may want to access it again later.

  How does the computer distinguish between these two scenarios? There are several different models, and typically the model is determined by the programming language used.

  Unmanaged Languages – In an unmanaged language such as C or C++, the programmer needs to explicitly make it clear when they are done with a particular object in the heap. These languages will provide a `delete` operation, which specifically tells the heap manager that the memory is no longer needed and that it can be freed and allocated for some other purpose.

  The problem with the unmanaged approach is that it depends on the programmer actually paying attention and freeing up objects in the heap when they are no longer needed. We have a lot of evidence that programmers are not very good at this.

  When a program repeatedly allocates memory for objects in the Heap and forgets to free them up, the program is said to have a ***Memory Leak***. When a program has a Memory Leak the program repeatedly allocates memory in the Heap for objects, does not free up the memory, and ultimately the heap runs out of available memory and the program crashes.

  Managed Languages – Managed languages work on the premise that programmers are very bad at managing their own memory, and that we should pass responsibility for that task on to the computer system. Rather than requiring a programmer to free memory, in a managed system, the programmer ignores whether or not they still need access to an object in the heap. Instead, the system runs a process called a ***Garbage Collector***. The Garbage Collector frees up unused heap memory and marks it as available for reallocation.

  There are several different techniques used by Garbage Collectors. A common method for Garbage Collecting is called the *Mark and Sweep* Method. In this method, every object in the Heap is marked as unreachable. The Garbage Collector then takes every active variable reference in the program and follows all references from those variables to other objects. When it finds an object, it changes the unreachable flag to reachable. After it has completed checking all variables, anything still marked as unreachable can't be reached by any existing variable and should be deleted.

  Let's consider how Mark and Sweep would work with our two different situations with the student object described previously. In situation (1) the object is created, used for a while, but ultimately we no longer need it, and no variables directly or indirectly reference it any longer. The Mark and Sweep algorithm would leave this object marked as

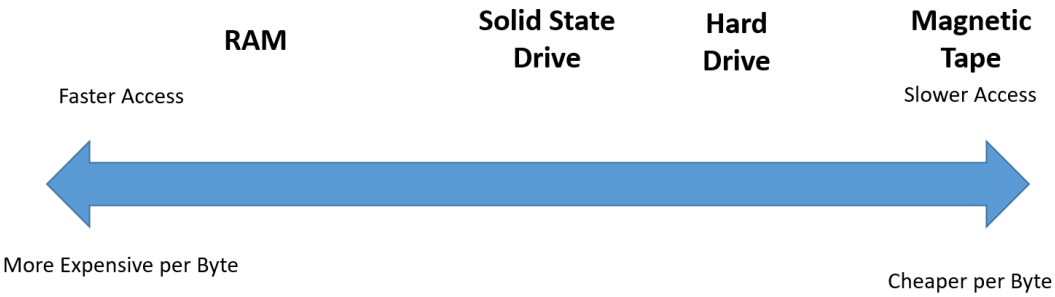
unreachable and it would be freed for later use. In contrast, in situation (2) we no longer have a direct reference to the object, however, we do have a direct reference to the list of students, and our student is contained within that list. When our Garbage Collector sees a variable referring to the list of students, it also follows references to each of the students on that list. In this case, our student would ultimately be reached, it would be marked as reachable, and it would not be garbage collected.

Theoretically having the programmer explicitly mark objects as deleted when they are no longer needed is much more efficient. In practice, as programmers are not very good at freeing memory when they should, a garbage collector can be very useful. The garbage collector does add to our overall processing overhead, as it takes CPU resources to do its job. However, as computers have become more powerful, the cost of this overhead is not that high, and therefore most newer languages are managed languages.

- I should mention that there is another use of the term Heap in Computer Science. Specifically there is a special tree data structure called a Heap Tree. These two Heaps are completely unrelated to each other.

The Memory Hierarchy

- We can place memory used by our computers on a continuum with more expensive, faster memory on the left and less expensive, but slower memory on the right. This is called the Memory Hierarchy. Let's start off with just a few different memory types. We'll add some more later in this lecture:



- As we can see, RAM is quite fast, but is also quite expensive to purchase per byte. A Solid State Drive (SSD) is much slower than RAM, but also cheaper per byte. If we have a lot of bytes to store, we might prefer buying a Hard Drive, which is slower than an SSD, but even cheaper per byte. Finally, if we really want to have a lot of storage space for low cost, we might use Magnetic Tape (Magnetic Tape is sometimes used for creating backup copies of a Hard Drive's contents).

- We'll discover that there are various techniques used by computers that make sense due to the relationship between storage media in the memory hierarchy.

Virtual Memory

- This is a technique in which the cheaper (per byte) storage on the SSD or Hard Drive is repurposed to augment the more expensive RAM in main memory.
    - As we've previously seen, instructions and data for a running program must be in main memory.
    - This means that the number of programs we can run simultaneously is limited by how much RAM we have in our computer.
    - If we want to run more programs than we have RAM, the clear answer is that we should go out and buy more RAM.
    - Virtual Memory lets us get around this.

- Let's take a close look at how Virtual Memory works.
    - The basic insight behind this technique is that all memory (primary, secondary, etc.) is designed to store bits. They might do so in different ways, electronically vs. magnetically, for example, but ultimately we think of what they are storing as 0s and 1s.
    - In Virtual Memory, we set aside some space in Secondary Memory to act as Main Memory.
    - When a program tries accessing an instruction or data item that's actually stored in the real main memory, everything acts normally.
    - When a program tries accesses something that it thinks is in main memory, but it actually isn't, that data is copied in from the SSD or HDD and something else in main memory is swapped out to the virtual memory section of the SSD or HDD in its place.
    - This whole process is transparent to the program running and is handled by the Operating System (special System Software that we'll be exploring in another lecture).

- The overall utility of Virtual Memory depends on how we're actually utilizing the programs that we have up and running on our computer.
    - If I have a lot of programs which I've started up on my computer, but I'm only interacting with a few at a time, then Virtual Memory works really well.
        - Suppose for example, I'm running Microsoft Word to work on a paper. I don't want to shut down Word, because it will be annoying to restart it and get back to the section of the paper I'm working on. However, although I have Word running, I'm really surfing the web and chatting with friends.
        - In this scenario, what will happen is that the instructions and data associated with Word will be stored in the section of the SSD that is pretending it is main memory. The instructions for my web browser and my chat software will be kept in real main memory.
        - My web browser and chat software will run nice and snappy, and I can switch back to Word at a later point, without having to restart the application.

    - If I'm working with a lot of programs simultaneously, this technique doesn't work well.
        - If I'm constantly switching between Word, Excel, my Web Browser, and my Chat Software, the Virtual Memory System won't be able to optimize which program instructions and data to put in real-memory vs. what to put in the slower fake SSD memory.
        - What actually ends up happening is that the computer is constantly copying instructions and data back and forth between the real RAM and the virtual memory on the SSD. This is called *Thrashing.* It's very inefficient and can slow down a computer considerably.

- All modern consumer computers and many smart phones run Virtual Memory.

Cache Memory

- Going from the CPU to Main Memory is slow. Main Memory is not only slow compared to a CPU's registers, but it's also on a different computer chip, and it takes time for the electrical signals to travel from one chip to another.
- Because of this, CPU designers add additional very fast memory into the CPU chip itself. This memory is called Cache Memory.
- There are several layers of this memory, and it's common to have an ***L1 Cache, L2 Cache***, and sometimes an ***L3 Cache***.

    - The L1 Cache is the very fastest memory outside of the registers themselves. It is also the smallest Cache.
    - L2 is larger, but slower.

- L3 is even larger, and for some chip designs, in contrast to L1 and L2, L3 may be shared between cores in a multi-core machine.

- Just to give you an idea of the size of these Caches an Intel i7 Core processor (which is a high-end processor sometimes found in consumer machines) has:

  - 64 kbytes of L1 Cache per Core
  - 256 kbytes of L2 Cache per Core
  - 2 Megabytes of L3 Cache per Core[2]

  As you can see the L1 and L2 Caches are extremely small and the L3 Cache, while larger, is still quite small compared to the total amount of main memory on the computer (8 Gigabytes to 16 Gigabytes is common on a consumer laptop as of this writing).

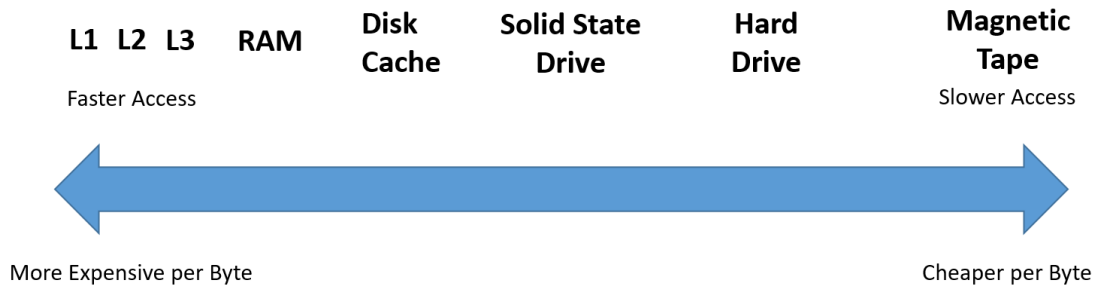
- Some computers even add an L4 Cache, which is slower than L3 and is not on the CPU chip itself, but rather on a separate chip. However, the L4 memory chip is much faster than RAM.

- Access to these Cache memories occurs automatically without any intervention by a programmer.
- The CPU designer needs to work to make sure that the data that is most likely to be requested gets transferred and stored into the Cache. Having data needed by the program in the Cache rather than having to be accessed in main memory will have a large impact on the execution speed of a program.

Disk Cache

- Similar to the CPU's Caches some higher end Solid State Drives and Hard Drives also have Cache Memory.
- These Disk Caches are RAM chips added to the SSD or HDD. Particularly with an HDD, if a request for data to the HDD comes in, and that data is in the drive's RAM memory cache, accessing it will be much faster, than spinning up the Hard Drive, and then moving the arm mechanism to get the read/write head over the exact section of the disk that our data is stored in.

- Here's what our revised Memory Hierarchy looks like.[3]



Additional Memory Related Issues

[2] The L3 is listed per Core, as i7 Core processors with more Cores will have more amounts of L3, but any of the L3 Cache may be accessed by any Core, whereas the L1 and L2 are only accessible by the associated Core.

[3] Note: the exact position of the various memory types from left-to right on my diagram reflects more my ability to fit everything in, rather than the exact difference in speeds between these different technologies – magnetic tape, for example, would be much, much further to the right in terms of its speed difference from the other technologies

Let's take a look at a few more memory related issues you may run into.

I mentioned previously that ROM or Read-Only Memory is memory whose contents cannot change. It is non-volatile and the data in it will be available as soon as a device is turned on. Because of this, ROM is commonly used for special control instructions, such as telling a computer what to do when it first starts up.

- On traditional PCs, the **BIOS** (Basic Input/Ouput System) would be stored in ROM chips and would be the first set of instructions executed by the CPU. Its instructions would begin the sequence of starting up or *booting up* the computer system.
- On some newer PCs and on Macintoshes this process is handled by the **EFI** (Extensible Firmware Interface) which may also be found on ROM chips.

- In general, the concept of having special control instructions permanently written into a device's ROM chips is referred to as ***firmware***. This term can be found both in computing and in consumer electronics where, for example, a television or even a microwave might have its control instructions provided permanently by ROM chips.

While we think of the data in ROM as permanently set to a specific sequence of 0s and 1s, in fact, modern ROM can often have its value changed through special electronic processes. Under normal operation, the ROM is indeed read-only, but if we really need to change its contents we can.

- This has led to the ability to update firmware. If a manufacturer finds that their original instructions hardcoded into the firmware have problems, they may instruct users to update their firmware.
- When updating firmware, make sure that the device is fully powered, if a device's battery were to die partway through a firmware update, it may become unusable, as its instructions for its startup sequence may be partly, but not completely updated, leaving a partially completed and therefore unusable program.