hardware and operating system basics

On two occasions I have been asked,—"Pray, Mr. Babbage, if you put into the machine wrong figures, will the right answers come out?" In one case a member of the Upper, and in the other a member of the Lower, House put this question. I am not able rightly to apprehend the kind of confusion of ideas that could provoke such a question.

--Charles Babbage (1864), *Passages from the Life of a Philosopher*, ch. 5 "Difference Engine No. 1"

1 of 11

Computer systems can be loosely classified as such:

client/workstation: A general-purpose system primarily for interactions with a human user, such as a PC.

server: A system primarily used to respond to requests from other systems across the network.

handheld: Like a client/workstation, but miniaturized, like in a cell phone.

embedded sytsem: A sytem designed to perform one or a few dedicated functions. It is *embedded* as part of a complete device, such as a car or toaster.

mainframe: a server with high-performance data-handling. Used primarily by large organizations for large data processing needs. Mainframes have largely declined in popularity since the 1980's.

supercomputer: a system with high-performance computation. Used primarily in scientific modeling, such as predicting the weather.

In all of these cases, the basic design is fundamentally the same: one or more CPU's, system memory, and I/O devices. Differences in the hardware between these classes stem mainly from differing power and performance considerations.

RAM (Random Access Memory) is characterized by:

- The CPU addresses each byte by its numeric address.
- Contents of RAM are volatile: they get scrambled when the RAM loses power.

- RAM is faster than storage devices like harddrives. Harddrives are especially slow when you read or write from locations scattered throughout the drive. RAM doesn't have this problem.
- The code of running programs are stored in RAM. Running programs also use RAM as their first place to keep data they need during their operation. (Any data we want to persist between runs of the program must of course be stored on some kind of non-volatile storage.)

2 of 11

instruction/binary instruction/machine instruction/native instruction: a sequence of bits understood by the processor to signal a certain action.

Different CPU's recognize different instructions, but all CPU's need instructions to do a few basic things:

- copy bits
- arithmetic
- bit logic
- jumps (including a conditional jump)

register: a small memory area in a processor. Modern CPU's tend to have between a few dozen and a few hundred registers, each usually 32 to 128 bits in size.

ISA (Instruction Set Architecture): the set of instructions and registers of a CPU. When two processors implement the same ISA, they can both run the same machine code. The most prevelant ISA today is x86, used in Intel and AMD processors in PC's. ISA's tend to evolve over time such that later processors extend the set of registers and instructions. So today's x86 processors can run code which older x86 processors cannot.

3 of 11

The memory used by a running program is typically divided into three sections. The code itself lives in one section while the data lives in two others, the call stack and the heap.

The code section is almost always treated as read-only because "self-modifying" code is generally frowned upon.

The call stack is where we store the local variables of the functions we invoke, while the heap is where we store any other data.

The call stack is so called because it is organized into a stack, a LIFO (last-in, first-out) data structure in which we only remove the last thing added. For each function call, we add to the stack a “frame”, which consists of the local variables for that call and the return address (the address to jump back to when the function returns). The parameter variables are given their initial values from the arguments to the function.

big-endian vs. little-endian: Whether a system is big-endian or little-endian determines how bytes get copied between memory and registers. In a big-endian system, the most significant byte is copied first, *e.g.* the instruction which copies address *A* into register R copies the byte at *A* into the most-significant byte of *R*, *A*+1 into the next byte of *R*, *A*+2 into the next byte of *R*, and so on; in a little-endian system, the least significant byte is copied first. The choice of endianness is generally regarded as arbitrary.

4 of 11

The storage locations of a computer are often though of in a heirarchy. At the top are the fastest but smallest locations, the registers of the CPU. At the bottom are the slowest but largest locations, hard drives and other forms of high-capacity long-term storage.

In between the registers and system RAM, we have the processor’s cache. The cache is controlled automatically by the CPU such that, when it reads bytes from memory, those bytes get stored in cache. As long as those bytes of memory remain copied in cache, the CPU can subsequently read the cache instead of having to go out to actual memory.

Today’s newer CPU’s have caches of a few megabytes in size. Computationally intensive programs tend to run significantly faster when they can use the cache to avoid going out to actual memory as much as possible.

5 of 11

I/O devices contain registers, and it is these registers which the CPU reads and writes to communicate with the device.

In a system with **port-mapped I/O**, these registers are written and read using input/output instructions which specify the register by *port* number. Ports are effectively a separate address space.

In a system with **memory-mapped I/O**, the registers are written and read using regular copy instructions which specify the device register by a regular address. For this to work, certain addresses are mapped to devices rather than to memory.

polling: code running on the CPU periodically checks device registers to see if the device needs the CPU to do something. The problem with polling is that doing this constant checking can be wasteful.

interrupt: A signal sent from an i/o device to the CPU which causes the CPU to jump execution to an address designated in the interrupt table (a table of addresses in a special place in memory). When the interrupt handling code is finished, the processor generally resumes whatever it was doing when the interrupt came in.

hardware exception: Like an interrupt, but triggered by something in the CPU, *e.g.* a divide by zero triggers an exception.

6 of 11

word size: The “natural”, most efficient size to copy to and from memory. Usually corresponds to the size of the general purpose registers.

address size: The number of bits used for each address. Usually the same as the size of the general purpose registers.

processors vs. cores: A system can have multiple CPU’s. In the last decade, multi-processor systems have become common with multi-core chips, in which more than one independent execution unit is placed in the same CPU package. A new PC today has typically 2 to 4 cores, and the number of cores is expected to rise in the next few decades.

boot firmware: The code which the CPU runs upon power on. This code usually resides in a ROM chip on the motherboard, from which the CPU is hardwired to read its first instructions. On PC’s, this chip is called the BIOS (Basic Input/Ouput System). The principle job of boot firmware is to begin the loading of the operating system, which usually resides on a hard drive.

7 of 11

operating system: the code responsible for managing the hardware and all the other running programs. An OS:

- loads and manages *processes* (a running program in an OS is called a *process*)
- provides “interaces” for processes to the hardware *via* system calls
- provides a filesystem, an interface for processes to the storage devices
- provides a basic interface for users to launch and manage programs

The most popular OS’s today are Windows and Unix. Unix is actually an umbrella term referring to OS’s which follow a set of conventions established by the orignal (now defunct) OS called Unix. Apple Mac OS X and Linux are two example Unixes (or *Unices*, if you prefer).

8 of 11

device driver: a plug-in to the OS which communicates with a particular i/o device.

When we have more processes running than we have CPU's, the processes can't literally all be running at the very same moment, so instead we need a mechanism to pre-empt a process as it runs on the CPU so that the OS can hand the CPU over to another process for it run a while. If this happens fast and often enough, human users get the illusion that the programs are running at the same time even though they are really taking turns.

Interrupts suspend the currently running process to run the interrupt handler and then run the *scheduler*, the OS code which decides which process gets to next use the CPU.

9 of 11

Virtual memory is what allows processes in a modern system to use memory without concern for other running code. Virtual memory is also what restricts processes from interferring with the memory used by other processes or by the OS itself.

Before the OS hands the processor over to a process, it configures the memory tables such that the process runs in a lowered state of privilege and an address specified in the process's code is translated by the CPU into a different physical address in RAM. So addresses in a process's code are actually virtual addresses, not physical addresses. As far as each process is concerned, it has the full range of addresses to use for itself.

This mapping from virtual addresses to physical is done in chunks called pages. On today's x86 systems, the usual page size is 4KB.

When a process uses an address in a page which has not been mapped for it, this triggers a hardware exception usually called a *pagefault* (or *segfault,* "segmentation fault": older processors used a memory scheme divided into segments, not pages, but the term segfault still persists).

When a page isn't being used, the OS can swap it from RAM onto a hard drive. The next time the page is needed, it is swapped back out to RAM.

10 of 11

A system call is basically a function in the OS code which processes invoke using a special instruction. In this instruction, you specify a system call number, and the CPU looks for the corresponding address in the system call table. Because the OS controls the contents of this table, it determines the locations in its code where processes can jump to. (You wouldn't want processes to be able to jump to just any address in the OS code.)

Because processes can't directly talk to the hardware, they need the OS to do it for them, and so they make requests to the OS *via* system calls.

In a process, the virtual addresses used for the code and call stack are automatically mapped to RAM by the OS, but addresses used for heap memory must be explicitly requested with system calls. When no longer used, a chunk of allocated memory should be "freed" with another system call, thereby allowing the OS to reallocate that memory to your process in a later system call. Failure to free memory can result in requests for more memory eventually being rejected, especially in a long running process.

As a process runs, it transitions back and forth between the state of running—actually executing on a CPU—and waiting to be scheduled. A process may also be put into a blocked state, meaning it is waiting to be unblocked by some circumstance in the OS, at which point it will go back to waiting.

The most common reason a process gets blocked is because it is waiting on I/O. For instance, a process usually blocks when invoking a system call to read a file off a hard drive. This makes sense because, until the data has been read into memory, the process can't continue, and this may take a long time in CPU terms, time which would be better spent letting other processes use the CPU. When the data is ready, the OS will unblock the process, thus allowing it to continue.

11 of 11

Each partition corresponds to a contiguous chunk of a storage device (though often, a single device is simply taken up by one big partition).

Each partition can store files and directories, each of which has an id number unique within that partition. Files are not necessarily stored contiguously, but this reality is disguised from programs that use files; as far as programs are concerned, a file is a *logical* sequence of some number of bytes.

A directory is a list of file/directory ids mapped to names. So files don't really have names, they just have ids. The same id can be listed multiple times with different names and can in fact be listed in multiple directories. So it's a bit of a fallacy to think of files and directories living in a particular directory.

A parition contains special directory, called the *root* directory. Ultimately, to refer to any file or directory on the partition by name, we refer to it *via* a *filepath* from this root, *e.g. <root>*/foo/bar/baz is the path referring to baz in the directory bar in the directory foo in the root directory (baz may be either a file or a directory).

In Windows, partitions themselves are mapped to drive letters: *c:*, *d:*, *e:* , *etc.* So the root directory of the partition mapped to c: is denoted by the filepath c:/

(Windows considers / and \ to be interchangeable in filepaths; Unix insists upon /.)

In Unix, one parition is *mounted* to the special path /, which is called root. So the file path / refers to the root directory of the partition mounted at /. Other partitions are mounted to directories of other already mounted partitions. So if we have a / partition, mounting partition *X* at /foo/bar means that directory found at path /foo/bar becomes a stand in for the root directory of partition *X*.