Brief tour of hardware

Only the basic aspects are covered here. The hardware is divided into entities (“things”) and features (“actions” also called “mechanisms”); the main distinction is that you can see the former but not the latter.

The hardware entities covered are: CPU, MMU (main memory controller), secondary memory controllers, timers, buses and i/o devices.

The hardware features covered are: interrupts, protection modes and privileged instructions.

Processors

A processor is device that executes machine instructions one after another, in a perpetual sequence.

The processor is made of an **ALU** which does the work and a number of registers which act as local storage. These registers include the Stack Pointer (SP), the Program Counter (PC) and the Processor Status (PSW).

The execution of an instruction constitutes a **machine cycle** which is made of a number of steps the last of which being the triggering of interrupts if needed. Certain interrupts (memory) can also be triggered at a middlepoint of the machine cycle.

Interrupts

When a hardware device wants to signal an exceptional condition to the world, it triggers a CPU interrupt (devices other than the CPU can also be interrupted in a similar fashion).

An interrupt causes the CPU to stop doing whatever it is doing and perform the following actions:

1. Store the current values of the PC and PSW registers in memory (in a predefined location, usually relative to the stack pointer).
2. Load new values of the PC and PSW registers from a static table in memory (fixed location). The interrupt number (or level) serves as index in this table, so that different values of PC and PSW registers are loaded for different interrupt types.
3. Start a machine cycle.

Step (2) must be **atomic**, i.e. not interruptible.

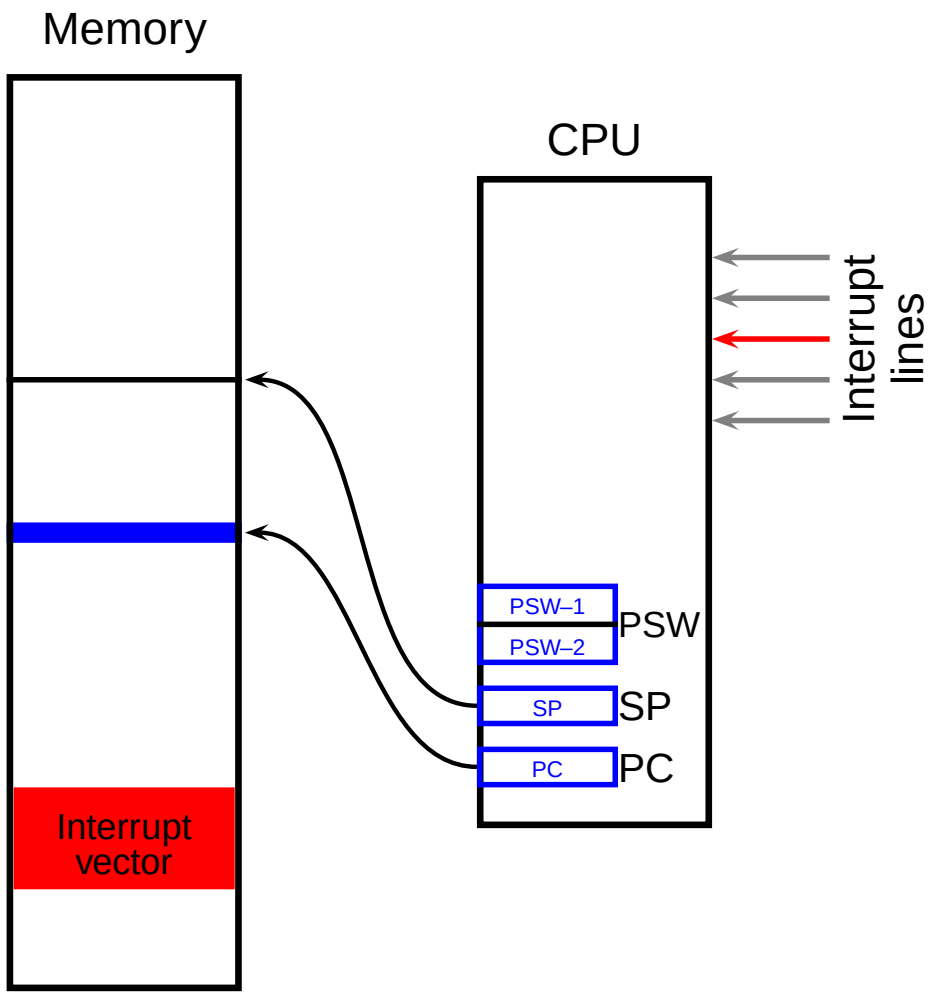
Steps (1) and (2) are called a **context switch**; a context switch can be performed in several other contexts not involving interrupts (a special machine instruction exists to do it “in software”).

Step (3) executes the first machine instruction from a different sequence, because the contents of the PC changed (equivalent to a goto instruction).

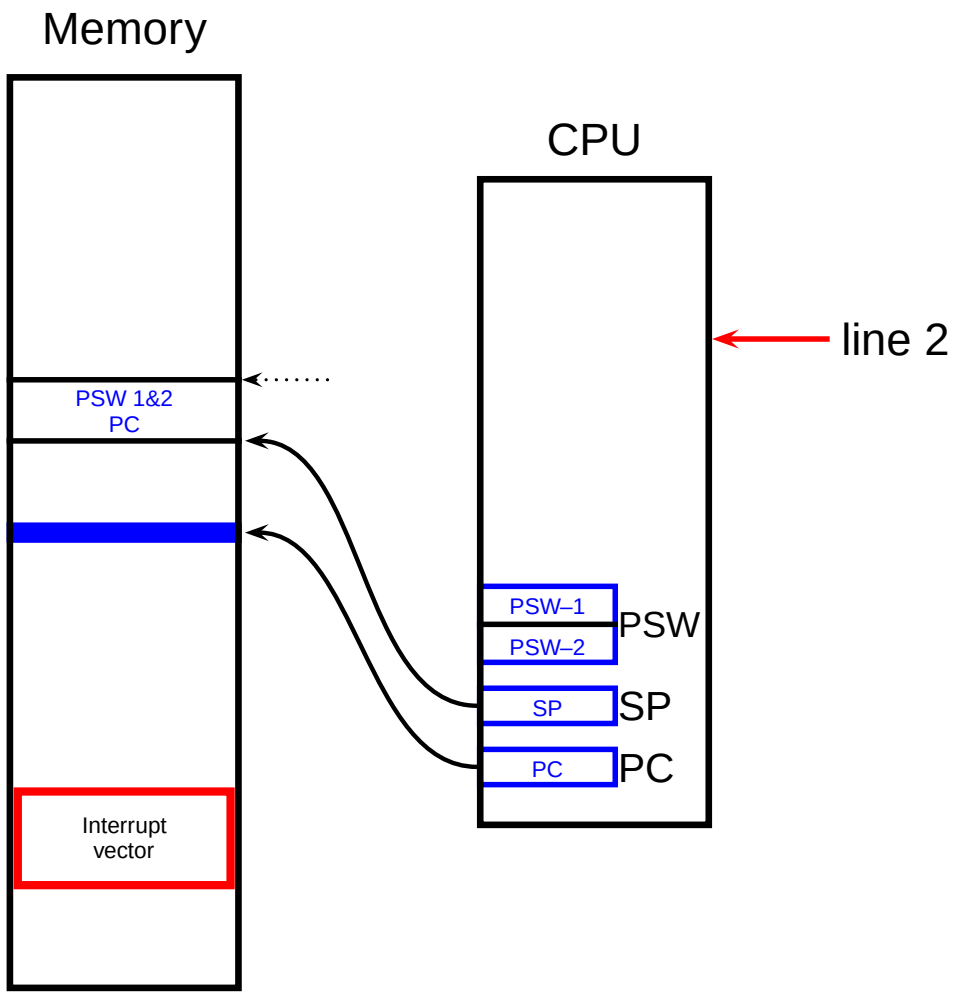
Note that some interrupts can be **masked** which means that the CPU can be told to refuse to accept them (an interrupt mask in the PSW is used for this).

Hardware interrupts have nothing in common with software interrupts known as **signals**.

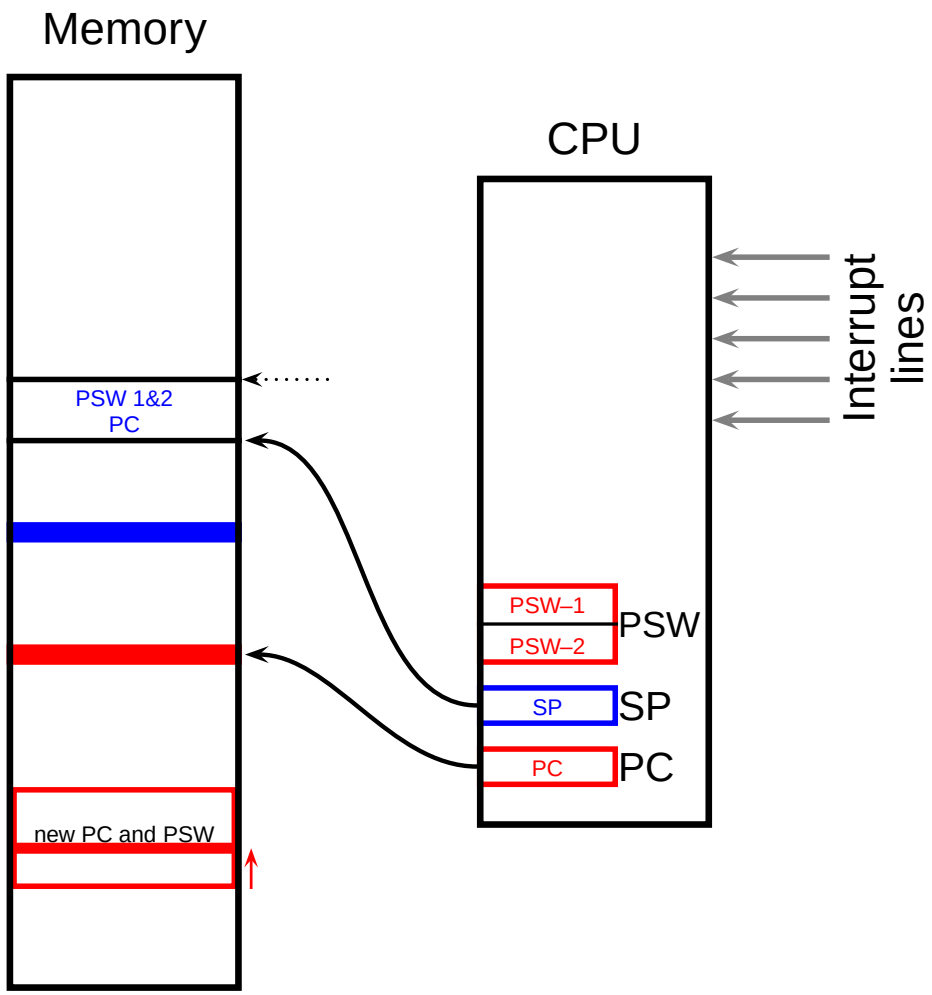
Processor and memory before an interrupt on line 2:



Processor and memory after step (1) of an interrupt:



Processor and memory after step (2) of an interrupt:

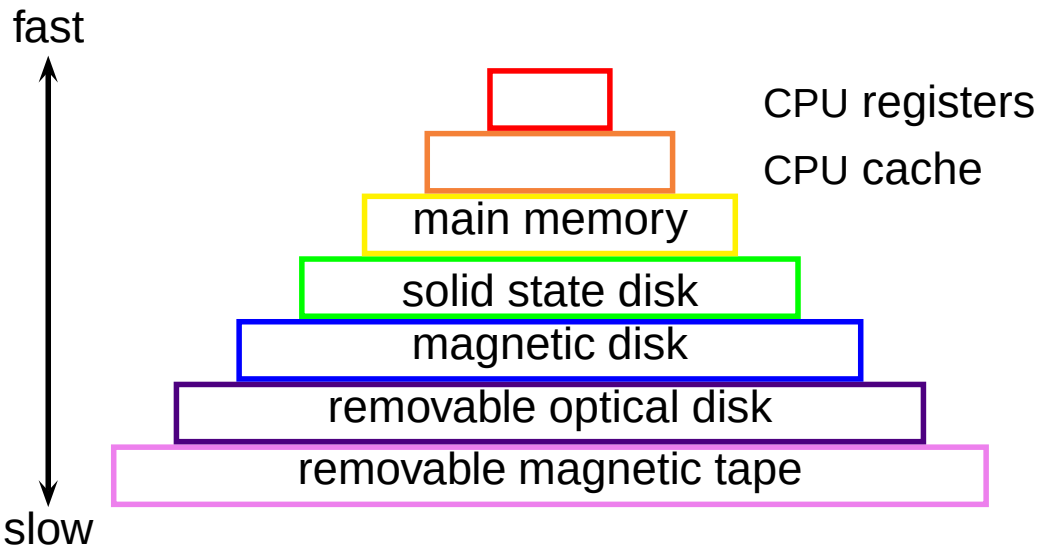


At this moment, the CPU resumes its normal operation and executes the next instruction (the one pointed to by the PC).

Interested in more details? Try LDD3 or D.A. Rusling (old but not incorrect).

Memory

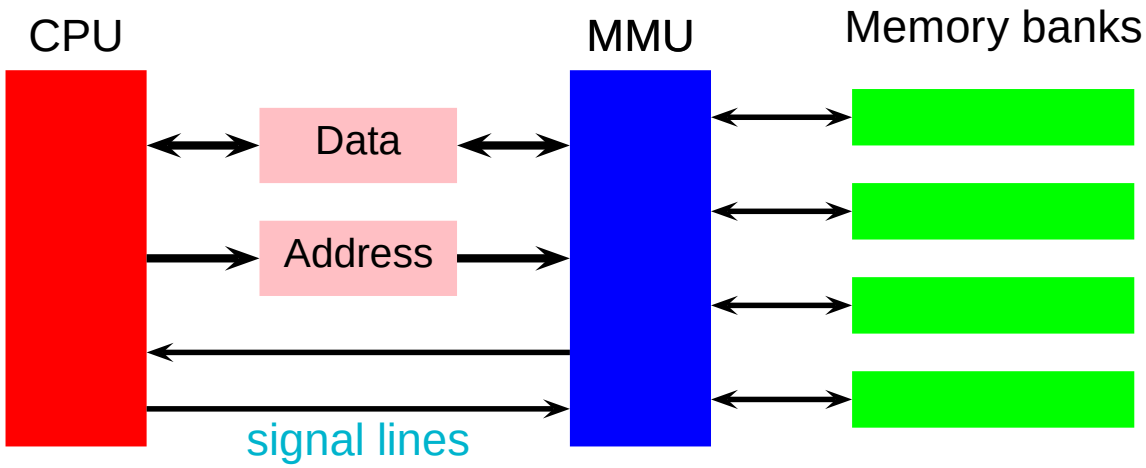
Price *vs.* performance considerations resulted in a hierarchy of storage devices. an additional factor–the address length–plays an important role in the size of devices at each level.



The small–large ranking is similar, but the slowest two memory types are not the largest; they are removable and are used for backup (neither size nor speed are not essential for this procedure).

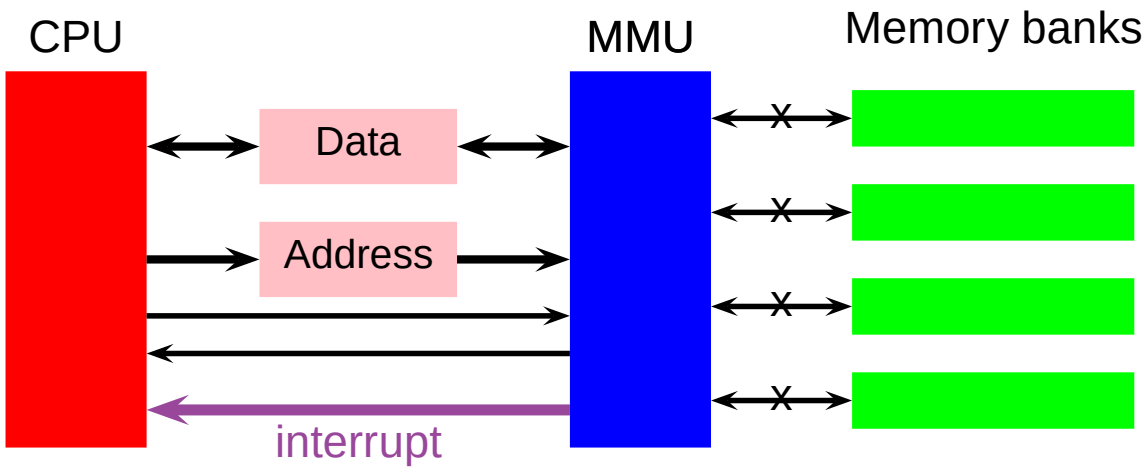
Main memory

This is a fast device designed to contain the data that the processor needs. A specialised processor called the Memory Management Unit (MMU) which serves as memory controller. The MMU is a device handler that is implemented in hardware because of the high speed requirements; it is often placed on the CPU chip to enhance speed even further.



The main design requirement for main memory is speed and that limits its size. Hence most OSs create an artificial, very large, virtual memory that satisfies the needs of memory–hungry applications.

The size of virtual memory is much greater than the size of real memory banks attached to the MMU. Consequently, some memory requests originating from the CPU cannot be satisfied by the MMU and result in **memory faults** (Page faults or Segmentation faults[a]) which are interrupts triggered by the MMU when it has no direct access to a requested memory location.



[a]Not related to Segmentation violation interrupts.

Secondary memory

All the storage devices beyond main memory are considered to be “secondary” memories. Their purpose is to provide bulk storage, hence they all are block devices, i.e. data transfers are done in large blocks as opposed[a] to words or bytes. Most block devices are connected to a DMA bus[b] which allows transfers of data blocks between the device and main memory (without involving the CPU).

The size of a block varies and is often software selectable. Hard disks have block sizes (called sectors) of between 512B and 2048B.

[a] This is becoming fuzzy as the size of memory–cache transfers increases.

[b] ISA, PCI, etc.

Transfer rate

The transfer rate is a fundamental parameter describing the performance of secondary storage devices. Hard drives have a transfer rate varying from 50 MB/s to 300 MB/s (SATA) and more (320 MB/s for Ultra–SCSI); other devices are much slower (USB 2.0 has a transfer rate of 60 MB/s). Note that these transfer rates are maximum rates; in real life the actual rates may be much lower due to the speed of interconnecting buses).

Disk drives

For practical purposes, the secondary storage device of relevance is a hard disk (magnetic or solid–state); other types are used mainly for backup or for recreational use.

For the most common magnetic drives, the transfer rate is not the most relevant parameter describing the disk performance; more relevant is the **access time** which takes into account the mechanic motions involved. The average access time has not changed much in the last 30 years (5–10 ms); the main change is in the disk capacities (from 10^8B in the 70s to more than 10^{12}B today).

While the access time to disk storage remained unchanged, the access time to main memory went down by orders of magnitude, making disk accesses a relative bottleneck in the computer system's performance. This led to several innovations such as RAM disks, disk cache, and, at the OS level, the **buffer cache**.

Clocks and timers

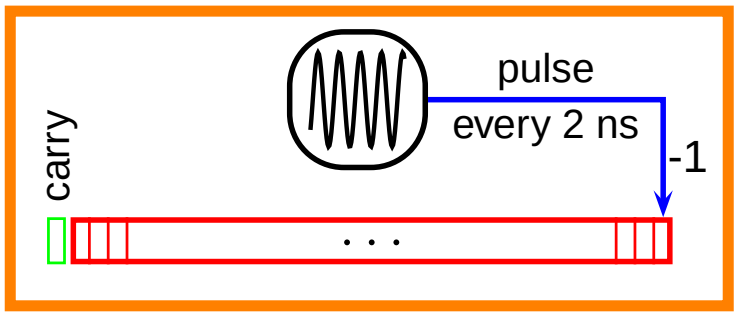
No OS can function without a hardware timer also called clock. The timer is needed to synchronise events within the computer system; if there is no time–of–day clock on board, a timer can be used to keep track of the current time.

A timer requires some source of periodic signal with a fixed frequency. Originally the power supply was used, giving a periodic signal with a frequency of 60Hz or 50Hz. Nowadays, crystal oscillators are used (as in “quartz” watches, but more accurate).

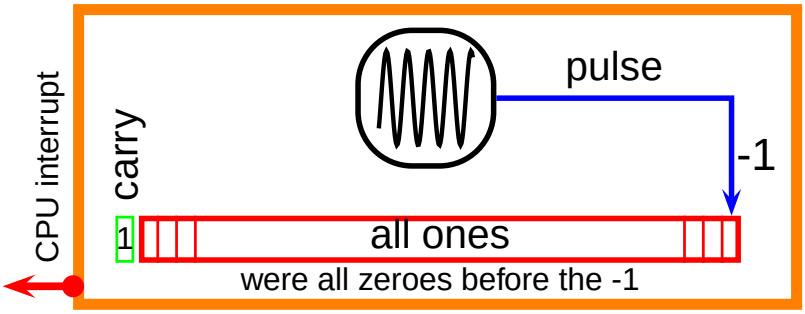
A properly prepared crystal emits a periodic signal with little variability. This signal is fed into the clock hardware which interrupts itself on every pulse (so that a 500MHz clock interrupts every 2 *ns*). Every pulse causes the clock hardware to subtract 1 from a special counter; when carry occurs (the value was 0), the clock chip triggers a CPU interrupt.

The timer has nothing to do with the CPU clock that drives the instruction cycle.

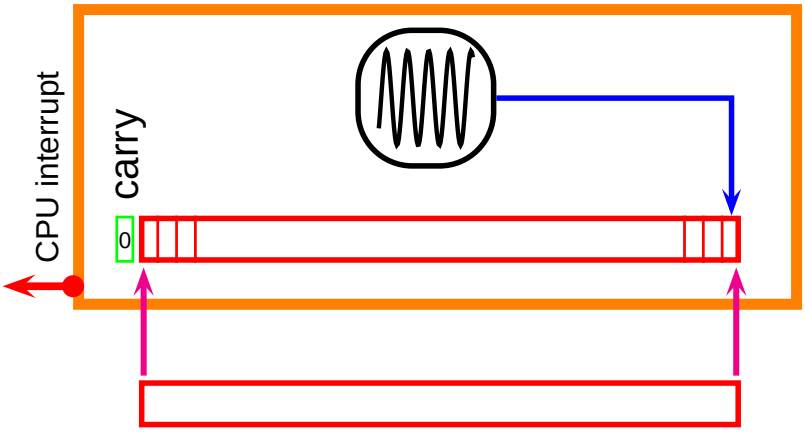
Timer interrupt



The chip interrupts the CPU when the carry bit is set.



The timer interrupt also forces the chip to reload the counter with a new initial value taken from a special register.



Soft timers

As timer hardware is cheap, it is normal to find several hardware “clocks” in a contemporary computer.

Even when several hardware timers are available, an OS will occasionally resort to soft timers which are virtual timers implemented totally in software.

This is done for two reasons:

1. Not enough hardware timers are available.
2. The overhead in handling a timer interrupt may be unacceptable for some applications.

The second case deserves some attention. Handling an interrupt takes some time (2 context switches), especially in systems with an MMU, a TLB or an instruction pipeline. If a hardware device triggers interrupts very frequently, it ends up reducing the system performance with no tangible benefits.

These interrupts can be replaced by a less accurate but less time–consuming mechanism. When a soft interrupt is needed at time t_s, the value t_s is simply stored in memory.

Whenever the OS is about to exit kernel mode, it checks the hardware time to see if the current time is greater than t_s. If it is, the OS performs whatever action necessary to serve the interrupt without having to go through a context switch.

A typical example is Gigabit Ethernet. At 1Gb/s, the maximum size (1500B) Ethernet frame is sent in $12\mu s$.

If the NIC interrupts every time it finishes a transmission it will chew up 5–10% of the CPU just on signalling. Replacing the hardware “device ready” interrupt with a soft timer will avoid it completely at a moderate cost (the Ethernet transmitter may be kept idle from time to time).

I/O devices

A large variety of Input/Output devices can be attached to the various buses in a computer system. From an OS perspective these devices are all similar with the exception that some may be faster than others.

The devices are connected to device controllers which are, in turn, connected to a bus. The OS talks to a controller and the details of the device (behind the controller) is often unknown to the OS.

Buses

Several buses coexist in a computer system, as shown in the figure depicting an old Pentium system. The key bus is a PCI bus.

PCI–X 133 MHz (8 bytes per cycle, with a maximum transfer rate of 1014 MB/s as there is some signalling overhead).

PCI–X 2.0 266 MHz (or even 533 MHz).

PCI–E (E = Express) version 1.1 has a capacity of up to 8 GB/s (more likely 4 GB/s in reality).

PCI–E 2.0 has a transfer rate of up to 16 GB/s.

PCI–E 3.0 will have a transfer rate of up to 32 GB/s.

USB 2.0 has a transfer rate of 60 MB/s although a rate of more than 30 MB/s is seldom possible because the USB uses the host processor and not its own controller for some activities.

USB 3.0 will have a transfer rate of 600 MB/s.

1394a up to 50 MB/s.

1394b 393.16 MB/s.

Kernel mode

A CPU operates in one of two distinct modes:

Kernel mode in which the CPU can execute any machine instruction and access any location in memory. Kernel mode has synonyms: “supervisor mode” or “system mode” or “privileged mode.”

User mode in which only untrusted instructions can be executed and full memory protection is in place.

Hardware details

From an Operating Systems perspective it is important what the hardware does, not how it does it.

Therefore such details as the memory type (e.g. DDR3) or processor architecture (*i7*) are not relevant.

Privileged instructions

Quoted from *Microsoft Help and Support*

The Intel architecture defines ”privileged” instructions and ”sensitive” instructions. The privileged instructions may only be executed when the Current Privilege Level is zero (CPL = 0). Attempting to execute a privileged instruction when CPL $\neq$ 0 will generate a general protection (GP) exception. Windows traps GP exceptions caused by executing privileged instructions and usually generates an application error.

The sensitive instructions (also called IOPL-sensitive) may only be executed when CPL $\leq$ IOPL (I/O Privilege Level). Attempting to execute a sensitive instruction when CPL $>$ IOPL will generate a GP exception. This should usually not cause a fatal error. The Windows Virtual Machine Manager (VMM) traps GP exceptions caused by executing sensitive instructions and (depending on the instruction) either simulates the instruction’s behavior in the VM in which the instruction was executed, or dispatches it to a virtual device driver, which simulates the instruction’s behavior.

The privileged instructions include:

CLTS - Clear Task-Switched Flag	LMSW - Load Machine Status
HLT - Halt Processor	LTR - Load Task Register
LGDT - Load Global DT Register	MOV CRn - Move Control Register
LIDT - Load Interrupt DT Register	MOV DRn - Move Debug Register
LLDT - Load Local DT Register	MOV TRn - Move Test Register

The sensitive instructions in protected mode include:

IN - Input	OUTS - Output String
INS - Input String	CLI - Clear Interrupt-Enable Flag (IF)
OUT - Output	STI - Set IF

Software entities

The OS is a collection of software modules united by two peculiar properties:

1. An OS module has no owner other than possibly the OS itself.
2. An OS module has no final goal to achieve; hence it lives forever never reaching a termination point.

The boundary of an OS is not well-defined, as illustrated by the utility **cron**, which is called by the kernel but probably should not be classified as part of it.

The software modules forming the OS are either **processes** or **handlers**. Application software consists of processes only.

Process

The key concept in OS is that of a **complete program in execution** called a **process**.

the relationship between a program and a process is the same as between a **class** and an **object** in OO programming languages:

Program or class is a blueprint describing the behaviour of its instances (processes or objects). It holds no resources and does not have the ability to perform anything.

Process or object is an instantiation of the above executing the code defined in the blueprint and holding the resources as described by the blueprint. Several processes/objects can exist concurrently for the same program/class.

Process resources

When a program is **invoked** (*instantiated*) by a shell or a process, a new process is created. This new process inherits some resources from its parent; it acquires additional resources during startup and execution.

The process holds:

Memory which is traditionally divided into 3 parts: the static part (fixed size and location), the stack (dynamic with a fixed starting point and a LIFO architecture) and a heap (dynamic with a fixed starting point and a semi–random architecture).

Files that the process opened (or created) including possibly **devices** masquerading as files (in Unix, all devices are pretending to be special files as much as possible, e.g. `/dev/tty`).

Environment descriptor containing several entries describing the impact of the environment on the process, including a list of pending signals, the working directory, the owner id (**uid**), the id of the process itself (**pid**), of the parent (**ppid**), of the process group (**gid**), etc.

Handler

A handler is a piece of code that is not an independent entity but plays a role similar to that of a function within the code of the OS. Similarly to functions, handlers are entered, in this case through an interrupt[a] which is a hardware equivalent of a function call, and exit through a return, in this case called return from interrupt.

Examples:

Alarm handler entered when a suitable timer triggers an alarm. it does its job (passes the alarm to the processes that care) and returns to wherever the CPU was before the interrupt.

Mouse handler (or **driver**) entered when a mouse button is pressed. It passes the information on and disappears.

[a] Other possibilities exist and will be described at a later time.