CA170: Week 10
OS and hardware

OS and hardware

- The fundamental job of an OS is to make the hardware usable for programs.
- To do this, it needs to understand the properties of its different forms of storage (in particular the access speeds and volumes).
- This is the "memory hierarchy" (see below).

Hardware v. Software

- First we consider what is a program anyway, and how can a program be implemented.

What is a Program?

- A list of instructions. A precise definition of what is to be done, as opposed to an English language description of what is to be done. A precise description is an algorithm, a recipe that can be followed blindly by a machine. A list of instructions to be executed one by one by a machine that has no memory of past instructions and no knowledge of future instructions.
- There are many processes that we observe in nature (animal vision, language use, consciousness) and invent in culture (calculating prime numbers, sorting a list). Some we have converted to algorithms. Some remain in the domain of an English language description.
- The Church-Turing thesis claims that "Any well-defined process can be written as an algorithm".

How is this Program, this list of instructions, to be encoded?

- What (if any) is the difference between the machine and the program?
- Many possibilities:
    1. Hardware (mechanical/electronic) is specially constructed to execute the algorithm. All you do is turn the machine on.
       e.g. Searle's "Chinese Room" thought experiment.
       (Hardware does not have to be general-purpose. Hardware can encode any algorithm at all.)
    2. Hardware is specially constructed to execute the algorithm, but there is some input data which is allowed to vary. This input data is read at run-time by the machine.
       e.g. Lots of machinery in factories and transport, calculators, clocks, (old) watches, (old) mobile phones.
       e.g. Many computer networking algorithms in routers (such as error-detection and error-correction) are done in hardware since they are run billions of times.
       Application-specific integrated circuit (ASIC)
       Bitcoin-specific ASICs are used to mine bitcoin.

    3. Hardware is a general-purpose device, and the program, as well as the input data, is read at run-time by the machine.
       e.g. Every normal computer.
- A general-purpose device implements a number of simple, low-level hardware instructions (the instruction set) that can combine (perhaps in the millions) to run any algorithm.
- Human programmer might write program in the hardware instructions themselves.
- Or might write program in High Level Language. Compiler translates this into low-level instructions for the particular hardware.

-
- The complete Intel x86 instruction set (the start of the popular x86 CPU family).
- For full reference guide, see Intel manuals for the x86-64 architecture.
- If you write direct in the low-level instructions, program likely to be much more efficient. But you can only run the program on that particular hardware.
- If you write in a HLL, the same program can be recompiled into the low-level instruction set of a different machine, which is an automated process, and much easier than having to re-write the program from scratch.
- One HLL instruction like x := x+y+5 may translate into many low-level instructions:
    - find the memory location represented by "x"
    - read from memory into a register
    - do the same with "y" into another register
    - carry out various arithmetic operations
    - retrieve the results from some further register
    - write back into a memory location

CISC v RISC

- There is a body of theory, Computability theory, showing what set of low-level instructions you need to be able to run any algorithm.
- We can have pressure to:
    - Increase the number of hardware instructions - CISC model.
      Observe the system in operation. Anything done many times gets its own dedicated instruction in hardware. (This is the broad history of desktop Operating Systems).
      Example: Intel x86 family.
    - Decrease the number of hardware instructions - RISC model.
      Simpler CPU runs faster. May not matter if executes higher number of low-level instructions if runs much faster.
      Also lower power needs, less heat. Suitable for mobile, portable devices.
      Examples: ARM architecture.
- As Operating Systems have evolved over the years, they constantly redefine the boundary between what should be hardware and what should be software.

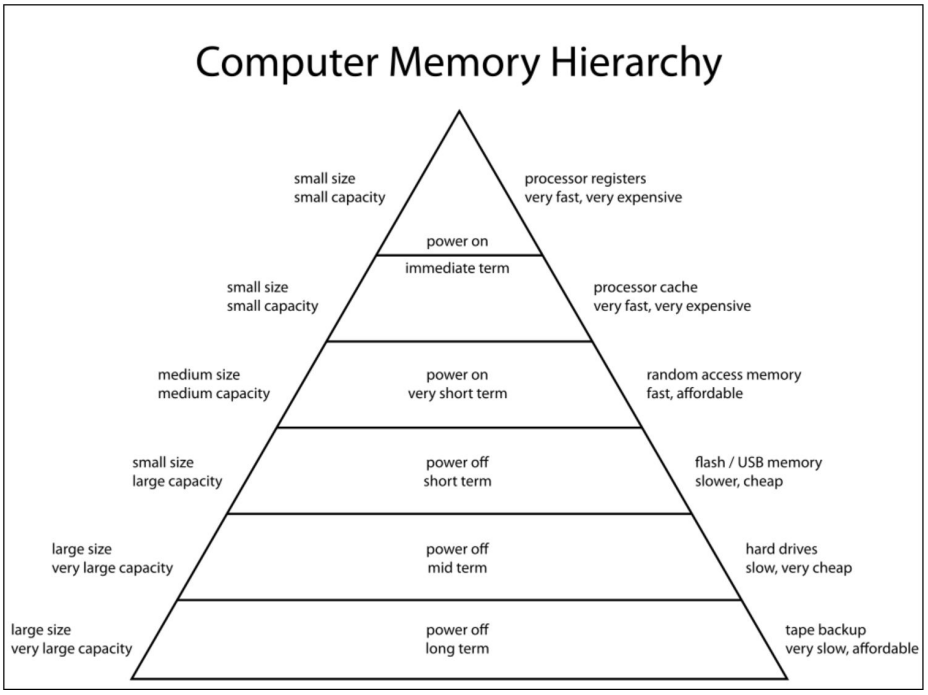
The standard model for a Computer System

- We have different types of Computer memory.
- Permanent v. Temporary.
    1. Program is kept until needed on some permanent (or "non-volatile") medium (hard disk, flash drive, DVD, backup tape).
       "Permanent" means retains data after power off.
    2. To run the program, it is loaded into some temporary (or "volatile") but faster medium (RAM).
       Temporary data structures and variables are created, and worked with, in this temporary medium.
    3. In fact the CPU may not work directly with this medium, but require that for each instruction, data is read from RAM and loaded temporarily into an even faster, volatile medium (registers).
       e.g. To implement `X := X+1`, where program variable X is stored at RAM location 100:

```
MOV  AX, [100]
INC  AX
MOV  [100], AX
```

    4. The results of the program are output either to some temporary or permanent medium.
    5. When the program terminates, its copy (the instructions) in the temporary medium is lost, along with all of its temporary data and variables. The long-term copy survives however, on the permanent medium it was originally read from.

Memory hierarchy



- "Memory" = RAM.
    - An unstructured collection of locations, each of which is read-write, randomly-accessible.
    - = "working memory"
    - = the fast, temporary medium that we actually run programs in. Reset every time we restart the computer.
        - RAM may be (usually is) cached for speed. See Cache memory.
- "Disk" = Hard disk (moving parts), Solid state drive (flash memory, see below), USB flash drive, etc.
    - The permanent (non-volatile) medium.
    - A more structured space than memory, formatted with a file system in place, where different zones of the disk have names and are separated from each other.
    - The permanent medium that we store programs and data files in.

- "Read-only" memory needed for "bootstrapping" code:
    - There has always been a need for some "Read-only" (or at least hard-to-rewrite) non-volatile memory built-in to the computer to hold the basic "bootstrapping" code: (the basic code to load an operating system, find devices).
    - ROM - Read-only memory - used for this.
    - Some forms of ROM can only be written once.
    - EEPROM can be repeatedly re-written.
    - Flash memory (developed from EEPROM) used for bootstrapping now.
    - Flash memory can also be used as the main disk. You can format and structure a flash memory space, and keep a file system on it. See solid-state drives and USB flash drives.

Up and down the memory hierarchy: Higher speed - Lower volume

- A typical computer: MacBrook Pro.
- Spec (highest values):
- 4 M cache memory.
- 8 G RAM memory.
- 1 T disk.
- Shows the memory hierarchy of speed and volume.

The multi-tier model is a practical necessity, not a mathematical necessity

- The multi-tier model is necessary to make machines work in practice, with the storage hardware that exists.
- Mathematically, a multi-tier system is not needed to run a program.
- Consider the following.
- If RAM was as fast as registers
    - If RAM was as fast as registers, the CPU could just operate on the RAM directly. No registers needed.

```
INC  [100]
```

    - Not very likely: This is not likely. A large, expandable memory space unlikely to ever be as fast as a small, dedicated memory space tightly integrated with the CPU.
    - Note that it is not just about speed: The circuitry does not exist to do arithmetic and logical operations on RAM. Only registers have this circuitry. RAM is currently designed only as a medium to send data to and receive data from registers. Adding this circuitry to every single RAM location would be a massive overhead, which is why the current, multi-tier architecture has evolved.
- If disk was as fast as RAM
    - If disk was as fast as RAM, then no need to load instructions into RAM from disk to be run. Run them direct from disk.
    - Not very likely: This is not likely. It seems likely though that a volatile medium will always be faster than a permanent one.
    - However, disk can be used an overflow of RAM: Paging and Swapping.
- Thought experiment
    - The above is really just a thought experiment, to think about the possibilities for different architectures.
    - For the whole history of computers, we needed all the speed we could get, which is why these complex, multi-tiered machines have evolved.
    - Mathematically, this multi-tier system is not necessary. Only a single medium is needed.
    - From the engineering point of view, it is doubtful if a single-tier system will ever come into existence, for the reasons above. But it is useful to think about the possibilities for different architectures, especially when later we consider using disk as an overflow of memory, and memory as a cache for disk.
    - 2015 article on "The Machine", a Hewlett-Packard plan to get rid of the multi-tier system. "The Machine is designed to [scrap] the distinction between storage and memory. A single large store of memory based on HP’s memristors will both hold data and make it available for the processor. Combining memory and storage isn’t a new idea, but there hasn’t yet been a nonvolatile memory technology fast enough to make it practical".

Os Structure

OS structure

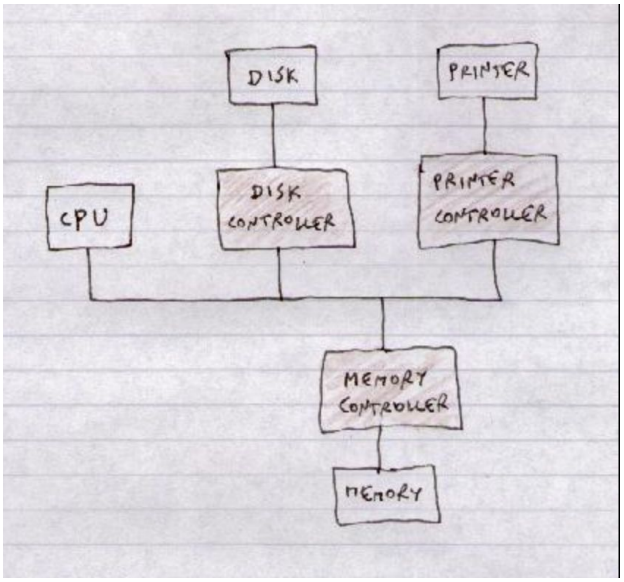
- "Process" = A program running. Could be multiple copies of the same program running. (Multiple processes.)

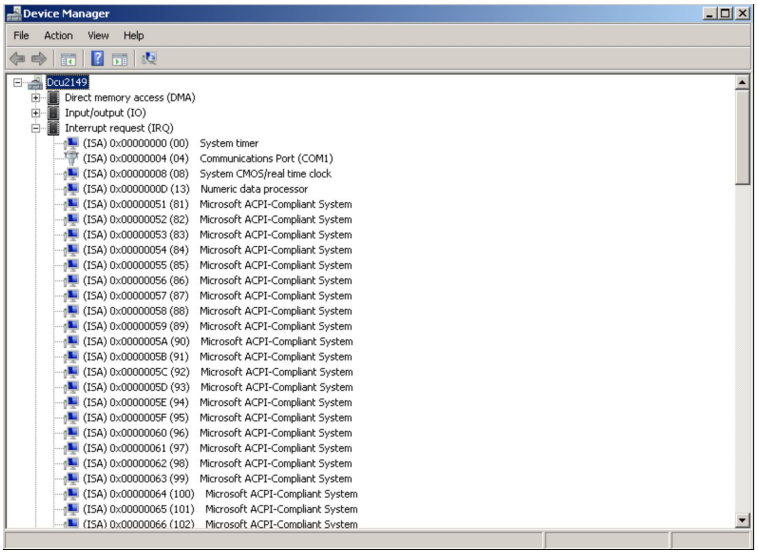
Device speed

- See History of OS.
- What do we mean "I/O devices are slower than CPU" ?
- Easiest to see with user input. User types say 1 character per second average. The CPU takes 2 micro-sec to put character into a memory buffer. For 998 out of every 1000 micro-sec, the CPU is doing nothing.
- This holds even for non-user I/O. e.g. Read from disk is at very slow speed compared to speed at which CPU can work.
- Even read from RAM is at slow speed compared to speed at which CPU can work. CPU could be doing other jobs while the read is going on.

Device controllers and Interrupts

- Device controller
    - Memory controller
    - Disk controller





- Device controllers operate in parallel.
- Can read/write devices and memory in parallel. Tell CPU when done.
- Instead of OS constantly polling device to see if done, device controller sends an interrupt.
- OS when it gets interrupt runs an interrupt handler.
- CPU is kept free (responsive to user, able to run other code) while device I/O is going on.
- Modern OS driven by interrupts.
- Interrupts are to do with the massive asymmetry between CPU speeds and device speeds. They are how the OS program runs on hardware with slow devices.

Examples of Interrupts

- Completion of user input (every key press) causes interrupt.
- Sometimes each key press is handled just by hardware device controller, which builds up a buffer, and even handles line editing itself - and only when you (say) hit return at end of command-line is interrupt sent to OS.
- Imagine a typical command-line: Shell process may have waited hours for completion of that Input.
- Completion of any type of Input or Output.
- Deliberate system call in a program causes interrupt (essentially, the start of all I/O). System calls:
    - Process - exit, load, execute, wait, signal, alloc
    - Info - query date, set date, query-set system info
    - File - create, open, delete, read-write
    - Device - request, release, read-write
    - Communications - create connection, send-receive messages
- Also errors - Divide by zero causes interrupt.
- Bad memory access.
- Timer interrupts.

Does an infinite loop cause an interrupt?

- Does an infinite loop quickly (or eventually) cause an interrupt?
- The Halting Problem (Turing, 1936).
- Might just be a long loop. We have no way of knowing.
  But still, even if long loop, control must switch occasionally - time-slicing.
  It is a timer interrupt that switches control. But loop runs forever, time-sli
- Unsolved problems in mathematics.
    - Note that if you could detect an infinite loop in general, then could solve all problems of the form:

```
Does there exist a solution to f(n)
for n > t?
```

    - by asking the OS if the following:

```
repeat
n := n+1
 test solution
until solution
```

    - is an infinite loop, or just a long loop? Then our OS could solve many of the world's great mathematical problems.
    - Many mathematical problems can be phrased as infinite-loop problems.
- Note that many "infinite loops" actually terminate with a crash, because they are using up some resource each time round the loop. e.g. This program:

```
f ( int x )
{
f(x);
}

f(1);
```

- will eventually crash with a stack overflow.
- This program however:

```
while true { }
```

- will run forever, time-sliced.

Interrupts - Keeping the OS in charge

- Interrupt idea is a way of periodically keeping the OS in charge so nothing runs forever without reference to the OS. e.g. In time-slicing, periodic timer interrupt to give OS a chance to do something else with the CPU.
- Remember, the CPU is just looking for a stream of instructions to process, whether they come from the "OS" or from "user programs". When the OS "runs" a program, it points the CPU towards a stream of instructions in the user program and basically hands over control. How do we know program cannot now control CPU for ever?
- Note: Interrupt does not necessarily mean that OS immediately attends to the process that sent it (in the sense of giving it CPU time). It just means the OS is now aware that that process wants attention. The OS will note that its "state" has changed.

CPU modes

- There must be a concept of an "OS-type program" (which can do anything, including scheduling ordinary programs) and an "ordinary program" (which is restricted in what it can do on the machine).
- The restrictions on the "ordinary program" are normally not a problem - they are basically just: "Other programs have to be able to run at the same time as you".
- This obviously makes sense in a multi-user system, but in fact it makes sense on a PC as well. When you're writing a program and you make an error, you don't want your crashed program to be able to crash the whole OS. You still want to be able to turn to the OS (which should still be responsive) and say "terminate that program".
- Code is executed either in kernel/system/monitor mode or in user mode.

```
boot in system mode, load OS
when run program, switch to user mode

when interrupt, switch to system mode
 and jump to OS code

when resume, switch back to user mode
 and return to next instruction in user code
```

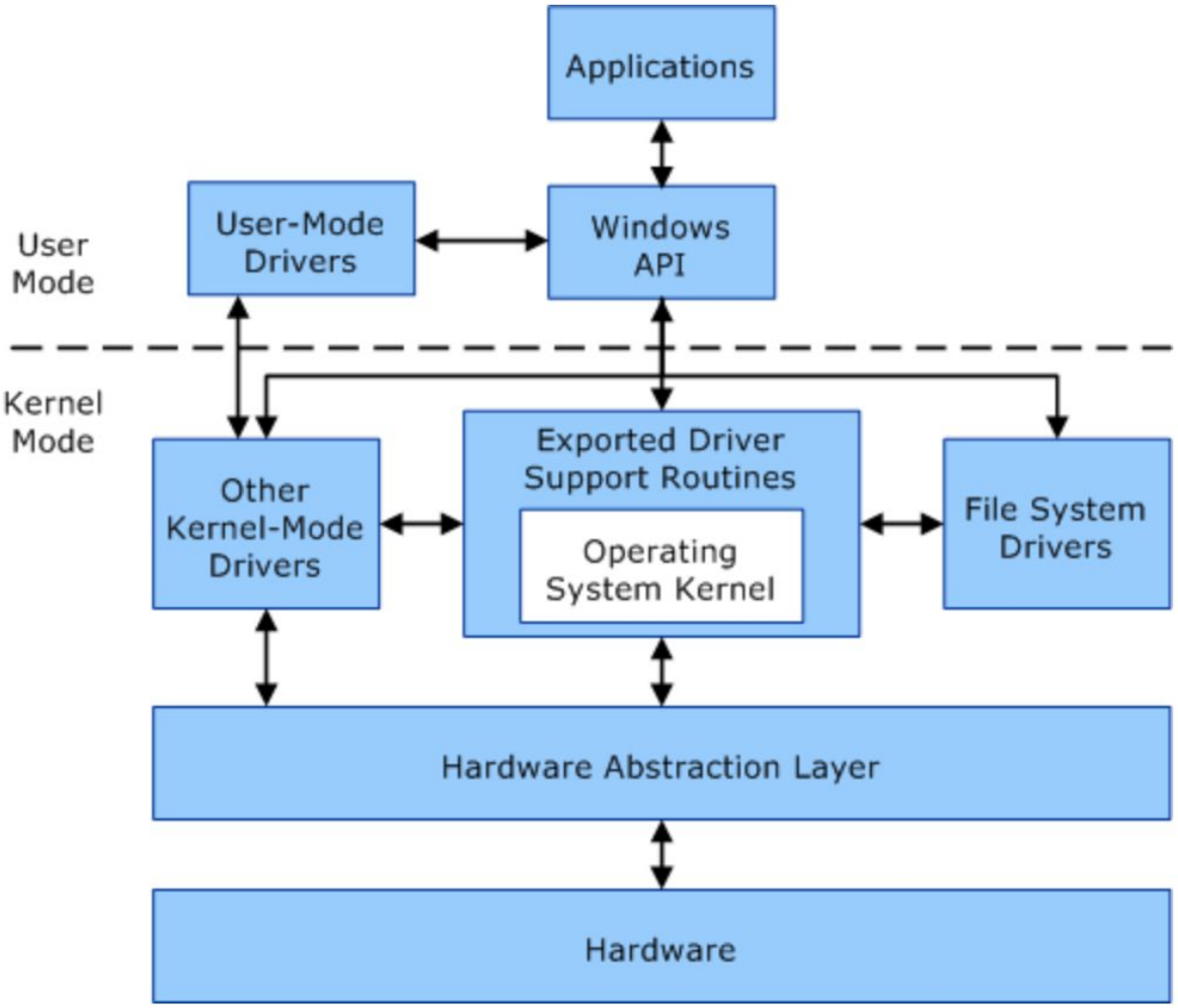
- Privileged instructions can only be executed in system mode.
- e.g. Perhaps any I/O at all - user can't do I/O, user has to ask OS to do I/O for it (OS will co-ordinate it with the I/O of other processes).
- On Unix/Linux, see these commands:
    - su
    - sudo

The kernel is the part of OS that is not scheduled

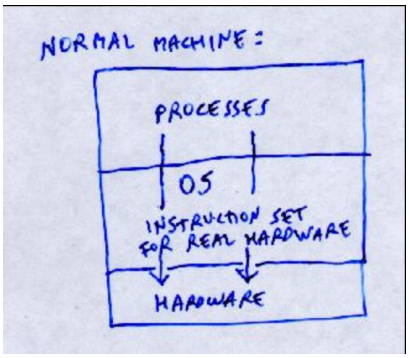
- Consider the different types of code, OS or applications:
    1. Parts of OS that are not scheduled themselves. Are always in memory. Operate in kernel/system mode.
       This part of the OS is called the kernel.
       Includes:
        - memory management
        - process scheduling
        - device I/O
    2. Parts of OS that can be scheduled. Come in and out of memory. Operate in user mode.
       Includes:
        - Command-line
        - GUI
        - OS utilities

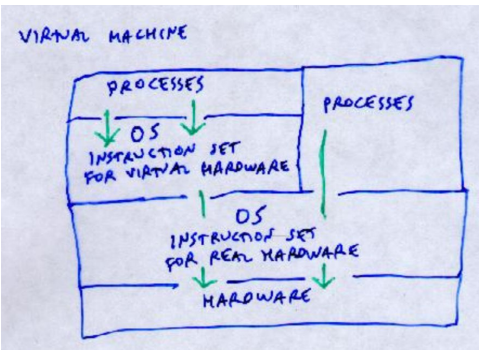
       Parts of OS that could be in either of the above:
        - file system
        - network interface

    3. Applications. All scheduled. Come in and out of memory. Operate in user mode.

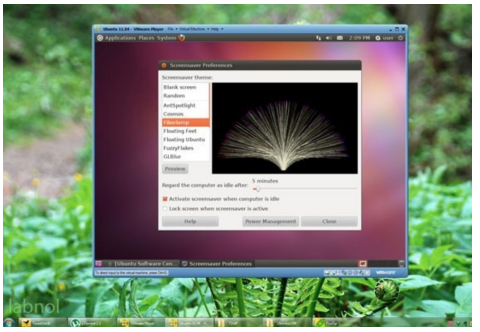


Virtual Machine (VM)





- Virtual machine
- IBM VM "Virtual Machine" OS, since 1972.
- VM idea built-in. Can run a multi-user OS in the user space of a single user of VM.
- Emulator - Programs from Operating System 1 or CPU 1 running on Operating System 2 or CPU 2.
- VMware - Run Linux inside Windows. Run another Windows inside Windows. etc.



- I use this kind of thing in DCU labs to teach system administration of an imaginary multi-machine network all running on one machine.

Java Virtual Machine

- Java is a language designed for a machine that does not exist, but that can be emulated on top of almost any machine. Java runs on a virtual piece of "hardware", the Java virtual machine.
- Promises:
    1. Portability. Write an application once, run everywhere.
    2. Run code from Internet on client-side. Even though from a site with different hardware, they will run on your hardware.
- Java is a HLL. It is "compiled" to a "virtual" instruction set called Java bytecodes. These run on Java VM, where they are actually mapped to native hardware instructions.
    - Java applets (client-side) - Written in Java, compiled bytecodes transmitted.

What language is OS written in?

- OS'es are written mainly in HLL, with some Assembly in Kernel.
- Assembly fast but hardware-specific.
- HLL much more portable, much easier to write/debug/change/maintain.
- UNIX / Linux
    - Mainly written in C.
    - Some parts in Assembly.
- Windows
    - Mainly written in C, C++.
    - Some parts in Assembly.
- C / C++ allows you directly embed Assembly instructions within it.
- Comparison of open-source operating systems shows kernel programming language.
- Comparison of mobile operating systems shows languages programmed in.

Is Assembly needed?

- Assembly is faster. But not needed everywhere.
- Perhaps only 5 % of the code is performance-critical. Things executed constantly - Interrupt handler, Short-term CPU scheduler.
- Other 95 % can be HLL for portability. 5 % can be machine-specific - needs a rewrite for each new hardware.
- Best to find out what is important to performance rather than pre-judge. (In which loops does the system really spend its time? You might be surprised.)
- Performance profiling
-
- Another reason to stick with HLL:
- Modern compilers have smart optimisers:
- Compiler often has a -optimise switch to analyse the HLL code to generate faster Assembly code.

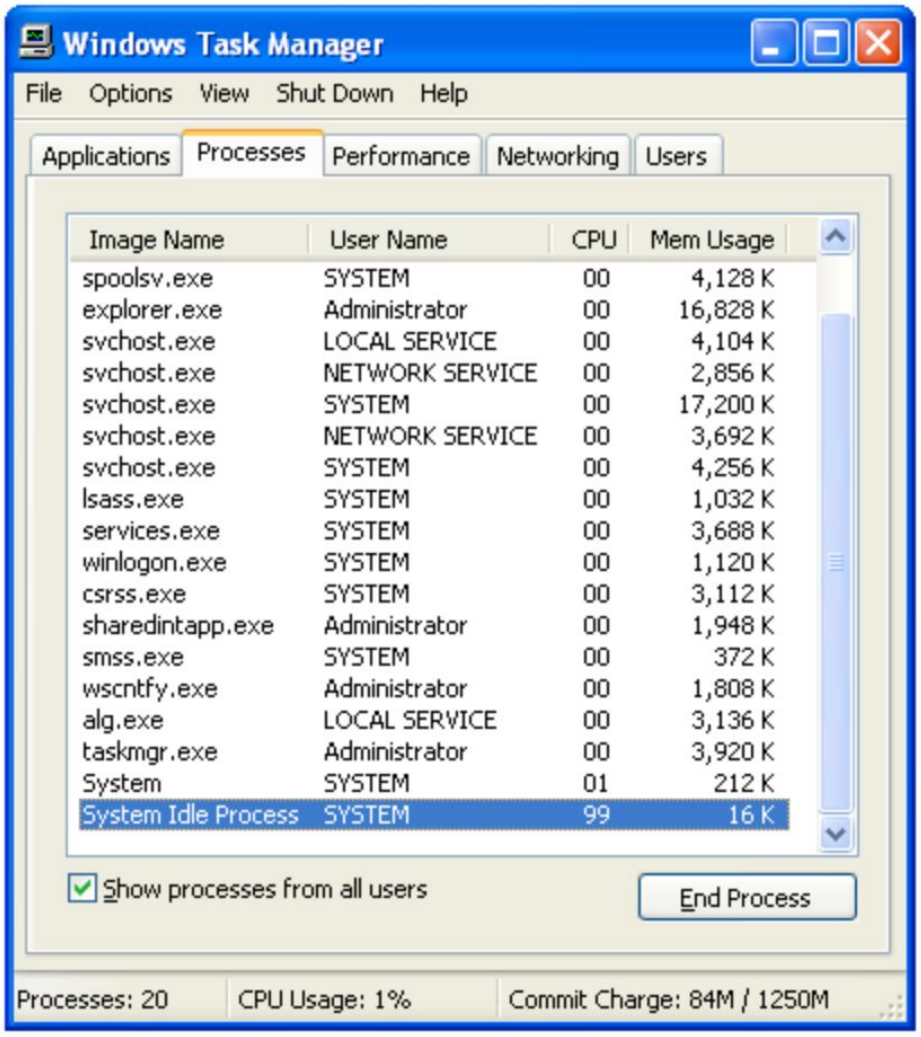
```
int __init_or_module do_one_initcall(initcall_t fn)
{
        int count = preempt_count();
        int ret;
        char msgbuf[64];

        if (initcall_blacklisted(fn))
                return -EPERM;

        if (initcall_debug)
                ret = do_one_initcall_debug(fn);
        else
                ret = fn();

        msgbuf[0] = 0;

        if (preempt_count() != count) {
                sprintf(msgbuf, "preemption imbalance ");
                preempt_count_set(count);
        }
        if (irqs_disabled()) {
                strlcat(msgbuf, "disabled interrupts ", sizeof(msgbuf));
                local_irq_enable();
        }
        WARN(msgbuf[0], "initcall %pF returned with %s\n", fn, msgbuf);

        return ret;
}
```

Processes

Processes

- Programs sit on disk. Maybe not run for years.
- Process - An instance of the program running.
- 1 user with 2 instances of text editor running - 1 program, 2 processes, each with their own memory space.
- 100 users with 100 instances of web browser running, all running the same copy from `/usr/bin/firefox` - 1 program, 100 processes, each with their own memory space.
- Processes (ps) in Linux

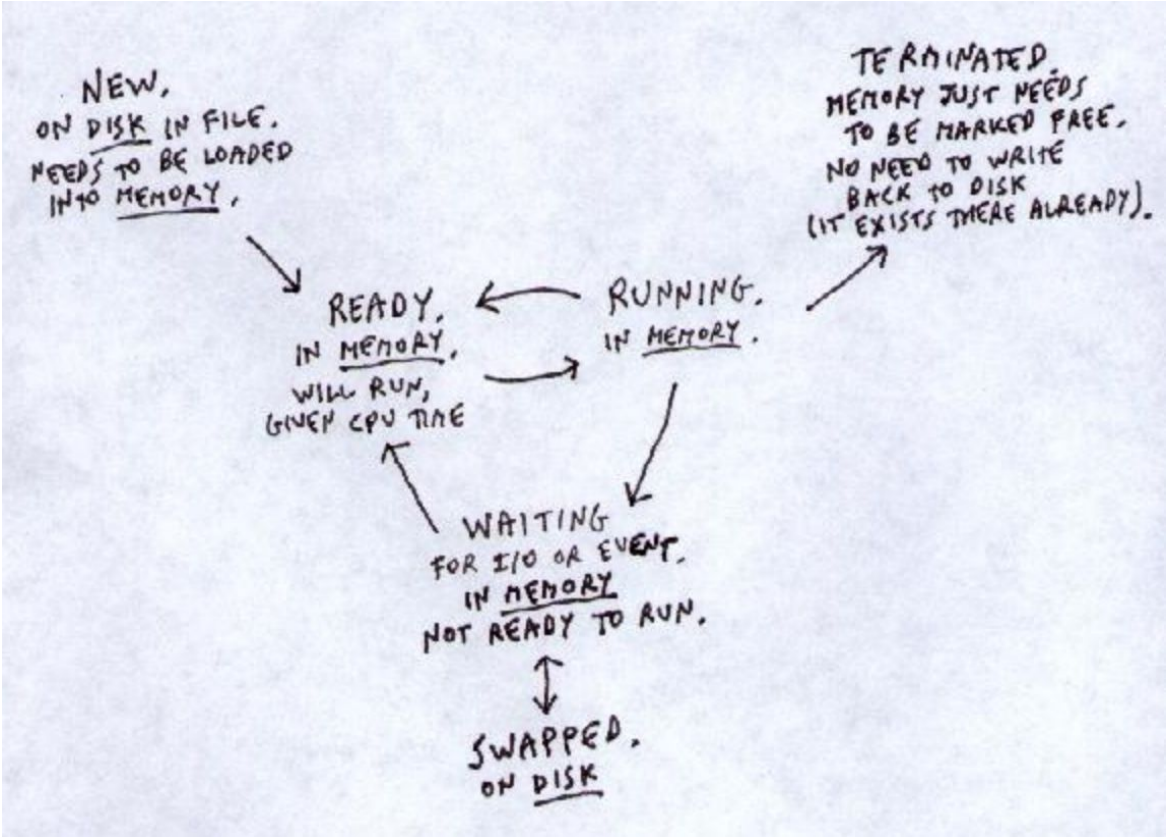
Windows processes (Task Manager)



- List processes in Task Manager.
- This is in fact Windows XP. From here.
- When the system is running slow, look for software updates running.
    - Some examples:
    - trustedinstaller.exe - Windows update
    - msiexec.exe - Windows update
    - ARMsvc.exe - Adobe update
- Sysinternals page at Microsoft has many tools to download, including:
    - Process Explorer - link in notes

Process State

- Process state
- Loaded from disk (permanent, power-off storage) into memory (RAM).
- Recall Memory hierarchy
- Starts running.
- May pause when doing disk I/O, waiting on user event, etc. Does not always have work for CPU to do.
- May pause for long time.



- READY to RUNNING - Short-term (CPU) scheduler decides if you can run now.
- RUNNING to WAITING - I/O or event wait.
- WAITING to READY - I/O or event completion.

CPU scheduling (Time-slicing)

- RUNNING to READY - Why stop a process that is happily running?
- To allow the OS do something else with the CPU. e.g. To allow the OS give CPU time to other process for a while.
- This time-slicing by the Short-term (CPU) scheduler is how we run many processes at the same time.
- Most processes look like:

```
Short period of calculation (short "CPU burst")
Long period of I/O
Short period of calculation
Long period of I/O
..
```

- i.e. Most processes regularly "vote themselves" off the Running list anyway. - We don't have to force them to go.

Swapping

- It is often the case that processes are idle for a long time. e.g. A GUI process waiting on user input, that the user has not looked at in hours.
- If running short on RAM, and process has not run for a long time, it may be removed from RAM to disk. (Or parts of it may.)
- This is called Swapping to disk.
- It is the running copy on disk. Not the same thing as the original program file.
- If you click on the program again, there may be a delay as the OS gets it back from disk.
- Swapping is integrated with Paging

Threads

- Threads
- 2 (or more) flows of control in the same process. 2 (or more) instruction counters. Same memory (same data and code space).
- e.g. Web server - Each request is a separate thread, all running in parallel until request completed.
- e.g. Web browser - multiple windows downloading at same time. Each is a separate thread in same process.
- On Linux, ps can list threads.

```
see all processes and threads:
 ps -ALf

LWP = thread id
note multiple threads within same process id PID
NLWP = no. of threads in this process

sort by processes with the most threads:
 ps -ALf | sort -k6 -n

show threads for one process:
 ps -Lf -p (PID)
```

- Scheduling of threads: OS may do the CPU scheduling between threads (kernel-level threads). Or program designer may implement their own (user-level threads).
- Up to program designer to make sure threads don't interfere with each other (or at least, only interfere in the ways he intends). Maybe nice to have OS protect us against incompetent thread design, but how can it? Note OS does not (can not) protect against lots of types of incompetence, e.g. infinite loop.
- Protection:
  Threads - 1 user - friendly (though might be incompetent).
  Processes - multiple users - hostile.

Multiple cores

- Modern computers have multiple cores (multiple processing units, executing instructions in parallel).
- Modern OS will schedule threads and processes across multiple cores.
- Processor affinity - The idea that once a process or thread starts running on one core, it is often more efficient to leave it there (and its data in the core's cache memory) rather than change cores.
- On Linux we can look at multiple core use from command-line using ps and top and other commands
- See next page

Event logging

- Event logging: The OS logs (in a file) events connected to processes.
  - Event logging
  - Windows Event Viewer.
- Unix/Linux logs
  - Unix syslog.
  - On Unix/Linux, various text and binary format logfiles are in:

    ```
    /var/log
    ```
  - fail2ban logfile is plain text

    ```
    tail -20 fail2ban.log
    ```
- Log of user logins on Unix/Linux
  - Record of user logins, stored in binary file:

    ```
    /var/log/lastlog
    ```
  - Can see them with lastlog command. (Same name as file!)
  - All users who logged in in last 2 days:

    ```
    lastlog -t 2
    lastlog -t 2 | sort -k4
    ```

Multiple cores

```
Display number of cores available:
====================================
$ nproc
4

Run process using core n (where n is 0 to 3 here):
=================================================
$ taskset -c n prog


See what processes are currently using what cores:
=================================================
$ ps -Ao user,pid,ppid,comm,psr

USER       PID  PPID COMMAND         PSR
root         1     0 init              0
root         2     0 kthreadd          2
root         3     2 ksoftirqd/0       0
root        25     2 watchdog/3        3
root       328     1 udevd             3
humphrys 29858 29855 sshd              0
humphrys 29859 29858 csh               0
humphrys 32373 29859 ps                1
humphrys 32374 29859 grep              1

# PSR will be 0 to 3 here


Query if process is bound to a core:
=====================================

$ taskset -p 3
pid 3's current affinity mask: 1

$ taskset -p 25
pid 25's current affinity mask: 8

# 0001 (1) - bound to core 0 (see process 3 command /0)
# 0010 (2) - bound to core 1
# 0100 (4) - bound to core 2
# 1000 (8) - bound to core 3 (see process 25 command /3)


$ taskset -p 1
pid 1's current affinity mask: f

# 1111 (f) - can run on all 4 cores, not tied to any core



See load on different cores:
```

```
==============================
$ top
(and then press 1)
```