Basic Computer Architecture

Typical Machine Layout

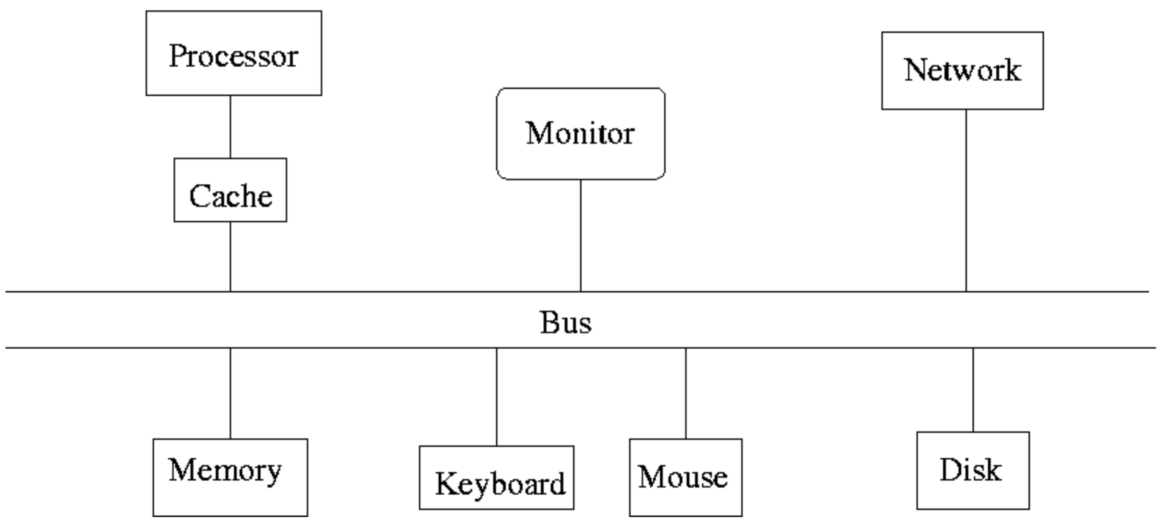


Figure based on M. L. Scott, *Programming Language Pragmatics*, Figure 5.1, p. 205

Structure of Lab Workstations

Processor and Cache

```
luke@l-lnx200 ~% lscpu
Architecture:          x86_64
CPU op-mode(s):        32-bit, 64-bit
Byte Order:            Little Endian
CPU(s):                8
On-line CPU(s) list: 0-7
Thread(s) per core:   2
Core(s) per socket:   4
Socket(s):             1
NUMA node(s):          1
Vendor ID:             GenuineIntel
CPU family:            6
Model:                 94
Model name:            Intel(R) Core(TM) i7-6700 CPU @ 3.40GHz
Stepping:              3
CPU MHz:               3895.093
CPU max MHz:           4000.0000
CPU min MHz:           800.0000
BogoMIPS:              6816.00
Virtualization:        VT-x
L1d cache:             32K
L1i cache:             32K
L2 cache:              256K
L3 cache:              8192K
NUMA node0 CPU(s):     0-7
Flags: ...
```

- There is a single *quad-core* processor with *hyperthreading* that acts like eight separate processors
- Each has 8Mb of L3 cache

Memory and Swap Space

```
luke@l-lnx200 ~% free
              total        used        free      shared  buff/cache   available
Mem:       32866464      396876    27076056       33620     5393532    31905476
Swap:      16449532           0    16449532
```

- The workstations have about 32G of memory.
- The swap space is about 16G.

Disk Space

Using the `df` command produces:

```
luke@l-lnx200 ~% df
luke@l-lnx200 ~% df
Filesystem                    1K-blocks      Used Available Use% Mounted on
...
/dev/mapper/vg00-root          65924860  48668880  13884156  78% /
/dev/mapper/vg00-tmp            8125880     28976   7661092   1% /tmp
/dev/mapper/vg00-var           75439224  13591304  57992768  19% /var
/dev/mapper/vg00-scratch      622877536     33068 622844468   1% /var/scratch
...
netapp2:/vol/grad             553648128 319715584 233932544  58% /mnt/nfs/netapp2/grad
...
netapp2:/vol/students         235929600  72504448 163425152  31% /mnt/nfs/netapp2/students
...
```

- Local disks are large but mostly unused
- Space in `/var/scratch` can be used for temporary storage.
- User space is on network disks.
- Network speed can be a bottle neck.

Performance Monitoring

- Using the `top` command produces:

```
top - 11:06:34 up  4:06,  1 user,  load average: 0.00, 0.01, 0.05
Tasks: 127 total,   1 running, 126 sleeping,   0 stopped,   0 zombie
Cpu(s):  0.0%us,  0.0%sy,  0.0%ni, 99.8%id,  0.2%wa,  0.0%hi,  0.0%si,  0.0%st
Mem:  16393524k total,   898048k used, 15495476k free,   268200k buffers
Swap: 18481148k total,        0k used, 18481148k free,   217412k cached

  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
 1445 root      20   0  445m  59m  23m S  2.0  0.4   0:11.48 kdm_greet
    1 root      20   0 39544 4680 2036 S  0.0  0.0   0:01.01 systemd
    2 root      20   0     0    0    0 S  0.0  0.0   0:00.00 kthreadd
    3 root      20   0     0    0    0 S  0.0  0.0   0:00.00 ksoftirqd/0
    5 root       0 -20     0    0    0 S  0.0  0.0   0:00.00 kworker/0:0H
    6 root      20   0     0    0    0 S  0.0  0.0   0:00.00 kworker/u:0
    7 root       0 -20     0    0    0 S  0.0  0.0   0:00.00 kworker/u:0H
    8 root      RT   0     0    0    0 S  0.0  0.0   0:00.00 migration/0
    9 root      RT   0     0    0    0 S  0.0  0.0   0:00.07 watchdog/0
   10 root      RT   0     0    0    0 S  0.0  0.0   0:00.00 migration/1
   12 root       0 -20     0    0    0 S  0.0  0.0   0:00.00 kworker/1:0H
   13 root      20   0     0    0    0 S  0.0  0.0   0:00.00 ksoftirqd/1
   14 root      RT   0     0    0    0 S  0.0  0.0   0:00.10 watchdog/1
   15 root      RT   0     0    0    0 S  0.0  0.0   0:00.00 migration/2
   17 root       0 -20     0    0    0 S  0.0  0.0   0:00.00 kworker/2:0H
   18 root      20   0     0    0    0 S  0.0  0.0   0:00.00 ksoftirqd/2
...
```

- Interactive options allow you to kill or *renice* (change the priority of) processes you own.

- The command `htop` may be a little nicer to work with.

- A GUI tool, **System Monitor**, is available from one of the menus. From the command line this can be run as `gnome-system-monitor`.

- Another useful command is `ps` (process status)

```
luke@l-lnx200 ~% ps -u luke
  PID TTY          TIME CMD
 4618 ?        00:00:00 sshd
 4620 pts/0    00:00:00 tcsh
 4651 pts/0    00:00:00 ps
```

  There are many options; see `man ps` for details.

Processors

Basics

- Processors execute a sequence of instructions
- Each instruction requires some of
  - decoding instruction
  - fetching operands from memory
  - performing an operation (add, multiply, ...)
  - etc.
- Older processors would carry out one of these steps per clock cycle and then move to the next.
- most modern processors use *pipelining* to carry out some operations in parallel.

Pipelining

A simple example:

$s \leftarrow 0$
for $i = 1$ to n **do**
  $s \leftarrow s + x_i y_i$
end

Simplified view: Each step has two parts,

- Fetch x_i and y_i from memory
- Compute $s = s + x_i y_i$

Suppose the computer has two functional units that can operate in parallel,

- An *Integer* unit that can fetch from memory
- A *Floating Point* unit that can add and multiply

If each step takes roughly the same amount of time, a pipeline can speed the computation by a factor of two:

	Cycle 1	Cycle 2	Cycle 3	Cycle 4	Cycle 5	...
Int:	Fetch x_1 y_1	Fetch x_2 y_2	Fetch x_3 y_3	Fetch x_4 y_4	...	
FP:		$s = s + x_1 y_1$	$s = s + x_2 y_2$	$s = s + x_3 y_3$	$s = s + x_4 y_4$	...

- Floating point operations are much slower than this.
- Modern chips contain many more separate functional units.
- Pipelines can have 10 or more stages.
- Some operations take more than one clock cycle.
- The compiler or the processor orders operations to keep the pipeline busy.
- If this fails, then the pipeline *stalls*.

Superscalar Processors, Hyper-Threading, and Multiple Cores

- Some processors have enough functional units to have more than one pipeline running in parallel.
- Such processors are called *superscalar*
- In some cases there are enough functional units per processor to allow one physical processor to pretend like it is two (somewhat simpler) logical processors. This approach is called *hyper-threading*.
  - Hyper-threaded processors on a single physical chip share some resources, in particular cache.
  - Benchmarks suggest that hyper-threading produces about a 20% speed-up in cases where dual physical processors would produce a factor of 2 speed-up
- Recent advances allow full replication of processors within one chip; these are *multi core* processors.
  - Multi-core machines are effectively full multi-processor machines (at least for most purposes).
  - Dual core processors are now ubiquitous.
  - The machines in the department research cluster have two dual core processors, or four effective processors.
  - Our lab machines have a single quad core processor.
  - Processors with 6 or 8 or even more cores are available.
- Many processors support some form of vectorized operations, e.g. SSE2 (Single Instruction, Multiple Data, Extensions 2) on Intel and AMD processors.

Implications

- Modern processors achieve high speed though a collection of clever tricks.
- Most of the time these tricks work extremely well.
- Every so often a small change in code may cause pipelining heuristics to fail, resulting in a pipeline stall.
- These small changes can then cause large differences in performance.
- The chances are that a "small change" in code that causes a large change in performance was not in fact such a small change after all.
- Processor speeds have not been increasing very much recently.
- Many believe that speed improvements will need to come from increased use of explicit parallel programming.
- More details are available in a talk at

  `http://www.infoq.com/presentations/click-crash-course-modern-hardware`

Memory

Basics

- Data and program code are stored in memory.
- Memory consists of *bits* (binary integers)
- On most computers
  - bits are collected into groups of eight, called *bytes*
  - there is a natural *word size* of W bits
  - the most common value of W is still 32; 64 is becoming more common; 16 also occurs
  - bytes are numbered consecutively, $0, 1, 2, \ldots, N = 2^W$
  - an *address* for code or data is a number between 0 and N representing a location in memory, usually in bytes.
  - $2^{32} = 4,294,967,296 = 4\text{GB}$
  - The maximum amount of memory a 32-bit process can address is 4 Gigabytes.
  - Some 32-bit machines can use more than 4G of memory, but each process gets at most 4G.
  - Most hard disks are *much* larger than 4G.

Memory Layout

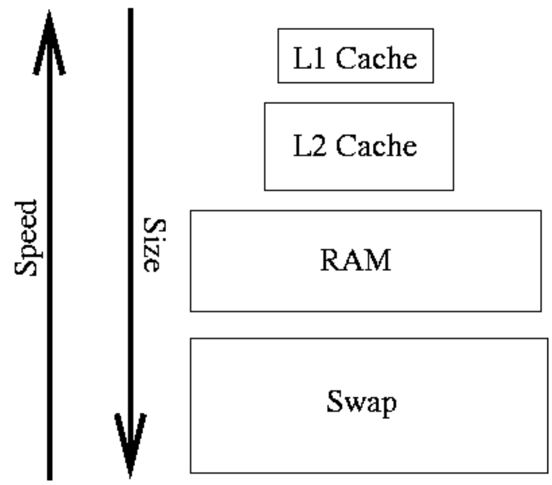
- A process can conceptually access up to 2^W bytes of address space.
- The operating system usually reserves some of the address space for things it does on behalf of the process.
- On 32-bit Linux the upper 1GB is reserved for the operating system kernel.
- Only a portion of the usable address space has memory allocated to it.
- Standard 32-bit Linux memory layout:

- Standard heap can only grow to 1G.
- `malloc` implementations can allocate more using memory mapping.
- Obtaining large amounts of contiguous address space can be hard.
- Memory allocation can slow down when memory mapping is needed.
- Other operating systems differ in detail only.
- 64-bit machines are much less limited.
- The design matrix for n cases and p variables stored in double precision needs $8np$ bytes of memory.

	$p = 10$	$p = 100$	$p = 1000$
n = 100	8,000	80,000	800,000
n = 1,000	80,000	800,000	8,000,000
n = 10,000	800,000	8,000,000	80,000,000
n = 100,000	8,000,000	80,000,000	800,000,000

Virtual and Physical Memory

- To use address space, a process must ask the kernel to map physical space to the address space.
- There is a hierarchy of physical memory:



- Hardware/OS hides the distinction.
- Caches are usually on or very near the processor chip and very fast.
- RAM usually needs to be accessed via the bus
- The hardware/OS try to keep recently accessed memory and locations nearby in cache.

- A simple example:

```
msum <- function(x) {
    nr <- nrow(x)
    nc <- ncol(x)
    s <- 0
    for (i in 1 : nr)
        for (j in 1 : nc)
            s <- s + x[i, j]
    s
}
m <- matrix(0, nrow = 5000000, 2)
system.time(msum(m))
##   user  system elapsed
##  1.712   0.000   1.712
fix(msum) ## reverse the order of the sums
system.time(msum(m))
##   user  system elapsed
##  0.836   0.000   0.835
```

- Matrices are stored in *column major order*.
- This effect is more pronounced in low level code.
- Careful code tries to preserve *locality of reference*.

Registers

- Registers are storage locations on the processor that can be accessed very fast.
- Most basic processor operations operate on registers.
- Most processors have separate sets of registers for integer and floating point data.
- On some processors, including i386, the floating point registers have *extended precision*.
- The i386 architecture has few registers, 8 floating point, 8 integer data, 8 address; some of these have dedicated purposes. Not sure about x86_64 (our lab computers).
- RISC processors usually have 32 or more of each kind.
- Optimizing compilers work hard to keep data in registers.
- Small code changes can cause dramatic speed changes in optimized code because they make it easier or harder for the compiler to keep data in registers.
- If enough registers are available, then some function arguments can be passed in registers.
- Vector support facilities, like SSE2, provide additional registers that compilers may use to improve performance.

Processes and Shells

- A *shell* is a command line interface to the computer's operating system.
- Common shells on Linux and MacOS are `bash` and `tcsh`.
- You can now set your default Linix shell at `https://hawkid.uiowa.edu/`
- Shells are used to interact with the file system and to start processes that run programs.
- You can set process limits and environment variables the shell.
- Programs run from shells take command line arguments.

Some Basic `bash`/`tcsh` Commands

- `hostname` prints the name of the computer the shell is running on.
- `pwd` prints the current working directory.
- `ls` lists files a directory
  - `ls` lists files in the current directory.
  - `ls foo` lists files in a sub-directory `foo`.
- `cd` changes the working directory:
  - `cd` or `cd` moves to your home directory;
  - `cd foo` moves to the sub-directory `foo`;
  - `cd ..` moves up to the parent directory;
- `mkdir foo` creates a new sub-directory `foo` in your current working directory;
- `rm`, `rmdir` can be used to remove files and directories; **BE VERY CAREFUL WITH THESE!!!**

Standard Input, Standard Output, and Pipes

- Programs can also be designed to read from *standard input* and write to *standard output*.
- Shells can redirect standard input and standard output.
- Shells can also connect processes into *pipelines*.
- On multi-core systems pipelines can run in parallel.
- A simple example using the `bash` shell script `P1.sh`

  ```
  #!/bin/bash

  while true; do echo $1; done
  ```

  and the `rev` program can be run as

  ```
  bash P1.sh fox
  bash P1.sh fox > /dev/null
  bash P1.sh fox | rev
  bash P1.sh fox | rev > /dev/null
  bash P1.sh fox | rev | rev > /dev/null
  ```

The `proc` File System

- The `proc` file system allows you to view many aspects of a process.