COMP 264 Fall 2015 Lecture Notes

R. I. Greenberg

August 24, 2015

1 Administrivia

- Fill out student information sheet now, especially name and email address.
- Read syllabus, e.g., comments about turning in homework and about collaboration.
- Make sure you can log on to our Loyola Linux setup: (`shannon`, appended with `.cs.luc.edu`); any students who did not have a Linux account before and have not registered last minute should now have an account with ID the same as your Loyola UVID. Password is pxxxxx with xxxxx the last 5 digits of student PID on ID card). Change your password using `passwd` command. If you have problems, email `mye@luc.edu`. (You can conenct to Loyola Linux from other Loyola lab machines using `Loyola Software` → `Internet Tools` → `Putty` off the start menu, or you can connect from elsewhere using the LSA (Loyola Secure Access) replacement for the VPN. With LSA, you will still need a terminal emulator, and you can dowload putty from `http://www.chiark.greenend.org.uk/~sgtatham/putty/download.html`; you'll probably want the link marked `putty-0.63-installer.exe`. Loyola also uses Filezilla, accessible at `http://filezilla-project.org/download.php?type=client` for file transfer. Another option for Windows to replace both Putty and Filezilla is at `http://www.filewatcher.com/m/SSHSecureShellClient-3.2.9.exe.5517312-0.html` .)
- An alternative method to connect to shannon without LSA is as follows. (I have usually had success using Firefox and Chrome, but mileage may vary depending on exact browser version.) In your web browser, visit `https://ada.loyolachicagocs.org/guacamole/#/login/` or the alias `http://cs264.cs.luc.edu` with same initial username and password as indicated above. (Once you start changing passwords, though, the authentication for Guacamole and Loyola Linux can be different.) After you log in to Guacamole, you will see a connection c264 that you can just click on after which you do your Loyola Linux login.

2 A Quick Introduction of Some Important Principles

During the first two classes, we'll get a little overview of some of the topics to be covered by looking at selected portions of notes for a CMU (home institution of the authors of our textbook) version of the course: *01-overview slides*. Without getting bogged down in the assembly code or memory performance examples, we will look at the 5 "great realities" there: slides 3–4, 8–9, 12–15, 18–21.

3 (Digital) Computer Representation of Information

- *digital* computer storage: a sequence of *bits* (binary digits), each 0 or 1.
- Contrasts with analog devices using continuous range of values.
- Information = Bits + Context
  - Different bit sequences can represent specific numerical values, but depends if interpreted as integer or floating-point and depends on specific conventions employed in the representation.
  - A bit sequence can also represent textual data. Usually, each character represented by a chunk of 8-bits (called a *byte*) using ASCII encoding standard. e.g.,:

character	bit pattern	decimal equivalent
⋮	⋮	⋮
A	0100 0001	65
B	0100 0010	66
C	0100 0011	67
D	0100 0100	68
⋮	⋮	⋮
a	0110 0001	97
b	0110 0010	98
c	0110 0011	99
⋮	⋮	⋮

    Various views of encoding shown by "`man ascii`" on a UNIX system. Our text Figure 1.2 uses an approach that assigns a decimal number to each bit pattern (as illustrated in table above).
- The 512 bits of textbook Figure 1.2 interpreted as a text file represent the `hello.c` C code for the `hello` program in textbook Figure 1.1.

4 Program Translation

Traditionally, programming languages divided into three categories:

- *High-level language* (e.g., Java, C++, FORTRAN, LISP, Pascal) hides details of computer and operating system on which will run. Said to be *platform-independent*, since same program can be converted to run on computers with different microprocessors and operating systems. (Actually, isn't always 100% true, as we'll discuss in class.))
- *Assembly language* specific to a particular microprocessor. Directly manipulate content of programmer-accessible registers in the microprocessor and in other memory components within the computer system. (A company making a new microprocessor may make it *backward compatible* so that it will run assembly language programs written for earlier versions of that microprocessor. But the decreased use of assembly language programming as the computer field has matured has led to increased freedom to develop radically new microprocessor designs.)
- The lowest level is a *machine language*. Binary encodings of assembly language instructions. Machine language instrs. directly read and executed by the microprocessor. Machine language almost same as assembly language; result of a fairly mechanical translation from a symbolic form to an encoding with just 0s and 1s.

One may also consider a lower level, in which machine language instructions are executing by running a program of even lower-level instructions referred to as microcode. In addition, terms have been introduced for higher levels, e.g., "fourth-generation" languages where the idea is to provide a higher-level language than programmers typically use that can be translated into a typical high-level language.

Sample C program `hello.c` is translated into an *executable object program* `hello` via UNIX command

```
gcc -o hello hello.c
```

Four phases in the *compilation system* as per *compilation.ppt*, namely

- preprocessing
- compilation
- assembly
- linking

See overview in text of how we'll gain benefits later in the course from understanding how compilation systems work:

- Optimizing program performance.
- Understanding link-time errors.
- Avoiding security holes.

A good exercise for now is as follows:

- Log on to one of the Loyola Linux machines. (See instructions in last set of lecture notes.)

  If you didn't have a Loyola Linux account before, you should have a new one now with ID same as Loyola UVID. The initial password is in the form "pxxxxx", where "xxxxx" is to be replaced by the last 5 digits of your Loyola personal ID number. Change password right away using `passwd`.
  (You may want to edit your `.bashrc` file to include things like `alias rm='rm -i'`, `alias mv='mv -i'`, and `alias cp='cp -i'`. If you don't have a `.bashrc` file and a `.bash_profile` file that invokes it, you can copy these two files from the `~rigreenberg` directory.)

- Enter our sample program into a file `hello.c` using the editor of your choice. I'll demo in class using emacs. (You can get an emacs tutorial inside emacs with `C-h t`, and more in-depth information with `C-h i`, where `C-h` means hold the "control" key while typing `h`.) (A particularly simple editor one could also use is pico (or nano).)

- Skim `man gcc` to see how to obtain the results of each of the four stages for our sample program, i.e., `hello.i`, `hello.s`, `hello.o` and `hello`. (Look at `-E`, `-S`, and `-c` options.)

- Produce these four results and look at them. Careful, `hello.o` and `hello` are binary files that will look mostly like gibberish and could mess up your terminal if you print them directly via, e.g., `cat hello.o`. Instead, you can see a print representation and page through it by doing `cat -v hello.o | more`

In demonstrating some of the above in class, we'll use a few UNIX commands; you can get documentation on any of them by using the `man` command (as discussed above in connection with `gcc`). Some other useful UNIX commands are `ls`, `mkdir`, `cd`, `cat`, `cp`, `mv`, `rm`, `diff`, and `more`. Protect directories in which you are doing homework with `chmod go-rwx .` (with the period being part of that command).

5 Basic Computer Organization and Program Execution

5.1 Major Components

- **Buses**.
  - A bus is a shared communication link between subsystems.
  - Usual physical implementation is a set of wires.
  - Advantage: Less space and power than many direct connections between subsystems, and fewer pins on chips comprising the system.
  - Disadvantage: Can be a communications bottleneck.
  - Generally transfer information in chunks referred to as *words*; most often 4 or 8 bytes to a word. We'll assume one 4-byte word at a time.
  - Different computers use different bus organizations. An example (modeled on Intel Pentium) in *hardware.ppt*.
- **Input/Output (I/O) Devices**. Each connects to I/O bus via a *controller* or *adapter*. Examples:
  - keyboard
  - mouse
  - display
  - disk
  - printer
- **Main Memory**.
  - Main storage for running programs and associated data.
  - Almost always *dynamic random access memory (DRAM)* technology.
  - Typically addressed by byte.
  - Instructions have a variable number of bytes in some machines; fixed in others.
  - Size of data items depends on type; a typical arrangement for C programs:
    - `short int`: 2 bytes
    - `long int`: 4 bytes
    - `float`: 4 bytes
    - `double`: 8 bytes
- **Central Processing Unit (CPU) or just Processor.** Typically three subparts:

- *register section* containing
  - word-sized registers that can be used by assembly language program for temporary storage.
  - "internal" registers used by the CPU but not directly accessible to the programmer. Two examples: CPU keeps track of address of next instr. in the *program counter (PC)*. When an instr. is fetched, generally stored in the *instruction register (IR)*.
- *arithmetic/logic unit (ALU)* implements the major arithmetic and logical operations performed by the CPU, e.g., adding two data values or performing arithmetic calculations needed to determine the memory location (effective address) where data is located.
- *control unit* generates internal control signals to properly orchestrate the movement of data and information within the CPU as well as to provide control signals to the I/O and memory subsystems.

5.2 Running the hello Program

Once the `hello` program has been compiled (`gcc` command above), the program can be executed by typing "`./hello`" from the same directory where the compilation was done.

(The "`.`" is a UNIX shorthand for "current directory". Typing "`./hello`" guarantees that the operating system will find the executable file. With an appropriate setting of the *environment variable* named "`PATH`" that tells the operating system what directories to search for executables, it would suffice to type just "`hello`".)

When you type the `./hello` command, you are typing to a command-line interpreter called a shell. The default shell you will get in the Linux lab is called `bash` ("Bourne again shell"), a play on the name of the early shell program `sh` referred to as the "Bourne shell". An alternative is `csh` and an enhanced version `tcsh`. There are also other extensions/combinations of these shells, e.g., `ksh`, `zsh`, and `pdksh`. A good overview regarding the different shells is at
`http://rig.cs.luc.edu/~rig/courses/c264/general/UnixShellTypes.pdf`
(originally obtained from
`http://www-group.slac.stanford.edu/cdsoft/presentations/UnixShellTypes.pdf`),
though I would say use "bash" rather than "sh" for programming.

keyboardread.ppt illustrates the reading and storage of the `./hello` command by the shell.

After this command has been entered, the shell loads the executable file `hello`. This loading can be set up by the processor to proceed by *direct memory access (DMA)* so that the information goes directly from disk to main memory without further involvement of CPU. Illustrated in *helloload.ppt*

displaywrite.ppt illustrates execution. Each instruction of the program is read from main memory into the CPU, where it is executed, part of which involves sending the `"hello, world\n"` string to the display.

6 Memory Hierarchy

Most modern (general-purpose) computers incorporate a hierarchy of memory components as illustrated in *memhier.ppt*. Major components:

- *physical memory* (or *main memory*) probably most familiar. E.g., may be referred to by an advertisement for a PC with "512 MB of RAM".
- Also mentioned often is the (hard) disk size (e.g., 80 GB (gigabytes)), used for the *virtual memory* level of the hierarchy.
- Less familiar to naive users but extremely important to performance: *cache memory*. Classically: CPU, then one level of cache, then main mem. Now, two or three levels of cache typical, often including one on same chip as CPU.
- "Remote secondary storage" of *memhier.ppt* not part of classical hierarchy, but now commonly utilized.
- Typically, cache uses *static random access memory (SRAM)*, and main memory uses DRAM. More on memory technologies later.

The reasoning for using this sort of hierarchy is based on the following facts:

- Smaller memories are usually faster, more costly per byte.
- Locality of Reference
  - Likely to reuse recently used data and instructions (temporal locality).
  - Likely to use items with nearby addresses at nearby times (spatial locality).

The idea of using a memory hierarchy is that by having several levels of memory, each smaller, faster, & more expensive per byte than the level below (farther from the CPU), we hope to achieve a cost almost as low as the cheapest mem. level and speed almost as high as the fastest level. Each level of the hierarchy holds some recently used data and/or instructions from the level below that can be accessed more quickly than by going to the level below.

Later we'll learn more about how the different levels of the memory hierarchy are organized and how the movement of content from one level to another is orchestrated.

7 The Operating System

Software interposed between application program and hardware, e.g., to facilitate access to keyboard, display, disk, and main memory. Key abstractions:

- processes
- virtual memory
- files

7.1 Processes

- A running program plus the state or context needed to continue it. .
- Can have many running concurrently. OS transfers control from one to another by doing a *context switch*.Illustrated in *switch.ppt*.

7.2 Threads

Instead of having a single control flow, a process may actually consist of multiple execution units called *threads*. Mostly beyond our scope ("Concurrent Programming" chapter).

7.3 Virtual Memory

Uses main memory as a cache for the disk so that processes can refer to a *virtual address space* much larger than main memory.

Linux virtual address space illustrated in *rtimage.ppt*. Divided into several sections that we'll come back to later, some in more detail than others:

- Program code and data.
- Heap.
- Shared libraries.
- Stack.
- Kernel.

7.4 Files

A sequence of bytes; used as an abstraction for any I/O device, e.g., disk, keyboard, display, etc.

8 Networks

Can think of as just another I/O device as per *nethost.ppt*. Text illustrates use of telnet to run `hello` program remotely as per *telnet.ppt*. We did that (and more) in class, using ssh, which is just a more secure version of telnet.

9 Important Themes

9.1 Abstraction

A critical idea throughout computer science (e.g., abstract data type, application-program interface (API), etc.). *abstractions-multi.ppt* cites additional abstractions (e.g., instruction set architecture as an abstraction of a machine running one machine-code instruction after another).

9.2 Amdahl's Law

Idea: Make the common case fast.

For example, when adding two numbers, overflow is rare. It pays to optimize for the usual case, even if that will cause extra delay in the case of overflow.

Idea: Performance gain achievable with an enhancement is limited by the fraction of time spent on tasks where the enhancement can be used.

Definitions:

$$\begin{aligned} \text{Speedup} &= \text{(old exec. time)/(exec. time using enhancement when possible)} \\ &= \text{(new performance)/(old performance)} \end{aligned}$$

$$\text{Fraction}_{\text{enhanced}} = \text{fraction of old exec. time during which enhancement can be used}$$

$$\text{Speedup}_{\text{enhanced}} = \text{how many times faster we can operate while using the enhancement}$$

Amdahl's Law:

$$\text{Speedup} = \frac{1}{\left(1 - \text{Fraction}_{\text{enhanced}}\right) + \frac{\text{Fraction}_{\text{enhanced}}}{\text{Speedup}_{\text{enhanced}}}} \ .$$

Corollary:

$$\text{Speedup} \leq \frac{1}{1 - \text{Fraction}_{\text{enhanced}}} .$$

- Example:

  Given: $\text{Speedup}_{\text{enhanced}} = 1000$ and $\text{Fraction}_{\text{enhanced}} = 1/2$.

  $$\text{Speedup} <= 1/(1 - 1/2) = 2 .$$

  (The full formula gives Speedup $= 1/(1 - 1/2 + 1/2/1000) = 1.998$.)

- Another Example:

  Given a program that does square root 20% of the time, with the total time spent on floating point (FP) being 50%. Suppose we have enough money to redesign the computer in one of two ways below. Which is better?

  Options:
  (1) Speed up square root by a factor of 10.
  (2) Speed up all FP by a factor of 2.

  Speedups:
  (1) 1 / (1 - .2 + .2/10) = 1.22
  (2) 1 / (1 - .5 + .5/2) = 1.33

  In this example, it is better to do the more modest speedup that can be applied more often, i.e., the choice agrees with the rule of thumb of making the common case fast, though more unusual numbers could lead to a different choice.

9.3 Concurrency and Parallelism

Our text uses *concurrency* for the general concept of a sysem with multiple, simultaneous activities, and *parallelism* for the use of concurrency to make a system run faster. Parallelism can be exploited at multiple levels of abstraction, e.g.:

- Thread-Level Concurrency. Can have multiple control flows in a single process. In older days, just a *uniprocessor system* with one processor to switch among different tasks. Now *multiprocessor systems* becoming more commoneplace. *Multi-core* processors have multiple CPUs. *Hyperthreading* or *simultaneous multi-threading* involves duplicating some hardware within a CPU so it can execute multiple flows of control.

- Instruction-Level Parallelism. Processors can execute more than one instruction at a time.

- Single-Instruction Multiple-Data (SIMD) Parallelism. Can have hardware allowing a single instruction to cause multiple operations to be performed in parallel.

In this course, we'll especially look at ILP. Other forms of parallelism get more attention in our text's "Concurrent Programming" chapter and the web aside "OPT:SIMD".