CMPE 220

System Software

– or –

History of Computing

System Software / History of Computing

Programming

- Machine language
- Loaders
- Assembly language
- Macro Processors
- Linkers
- Compilers
- Schedulers
- Memory Managers
- Servers

Architecture

- Microprogramming
- CISC
- RISC
- Distributed Systems
- Networked Systems

Rear Admiral Grace Murray Hopper

- December 9, 1906 – January 1, 1992
- PhD, Mathematics, Yale University, 1934
- One of the first programmers of the **Harvard Mark I** computer, she was a pioneer in computer programming who invented one of the first *linkers*.
- She popularized the idea of machine-independent programming languages, which led to the development of **COBOL**, an early high-level programming language still in use today.

My Background

Robert Nicholson

- BS, Computer Science - 1975
  CSU, Chico
- MS, Computer Engineering – 1978
  Stanford University
- Engineering and Management:
  Hewlett-Packard, Oracle, Silicon Graphics, Sun Microsystems,
  Red Herring Communications, NASA, various startups
- While in college, started a company building kit computers for local businesses

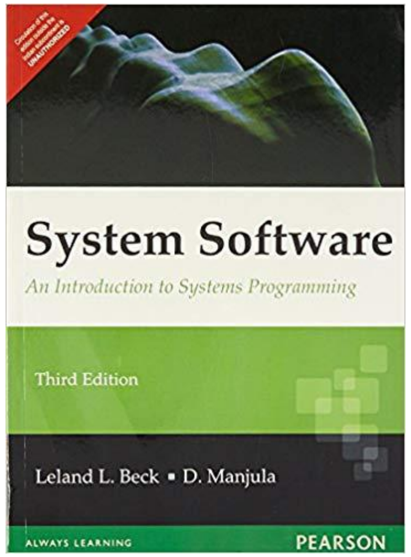
Housekeeping

- Class information in Canvas
- Email: robert.nicholson@sjsu.edu

Grading

- Assignments: 30%
- Midterm: 25%
- Final: 45%

Course grades will be based on a curve. Per CMPE Department policy, the median total score will earn a B+. Approximately one third of the class will earn higher grades, and another one third will earn lower grades.


System Software
An Introduction to Systems Programming
Third Edition
Leland L. Beck ▪ D. Manjula
ALWAYS LEARNING
PEARSON

Textbook (recommended):

System Software: An Introduction To Systems Programming, 3rd Edition
Leland L. Beck – 1996

Exams and Proctoring

- The Midterm and Final Exams will be in Canvas, administered during scheduled exam times.
- Exams are open book, open notes, open Internet

Examples and Assignments

- Examples will be based on Unix/Linux/POSIX – and if you don't know what that means, I'm about to explain.
- Programming assignments will be short and can be done on any system, but I strongly advise that you use a Unix/Linux/POSIX system, for several reasons:
  - It's going to be easier for you
  - I probably can't help with questions or problems you encounter on other systems
  - It's effectively an industry standard. From a career standpoint, you should be proficient in it!
- So, let's see why it's so important…

How Software Became Portable

- Before the 1970s, every computer manufacturer distributed bundled software with their systems – compilers, databases, editors, and so on. Once you chose a computer vendor, you were “locked in” to their systems.
- There was virtually no independent software industry.
- Three things changed that:
  - The development of system-independent programming languages
  - The development of system independent databases (SQL)
  - The proliferation of Unix/POSIX based systems

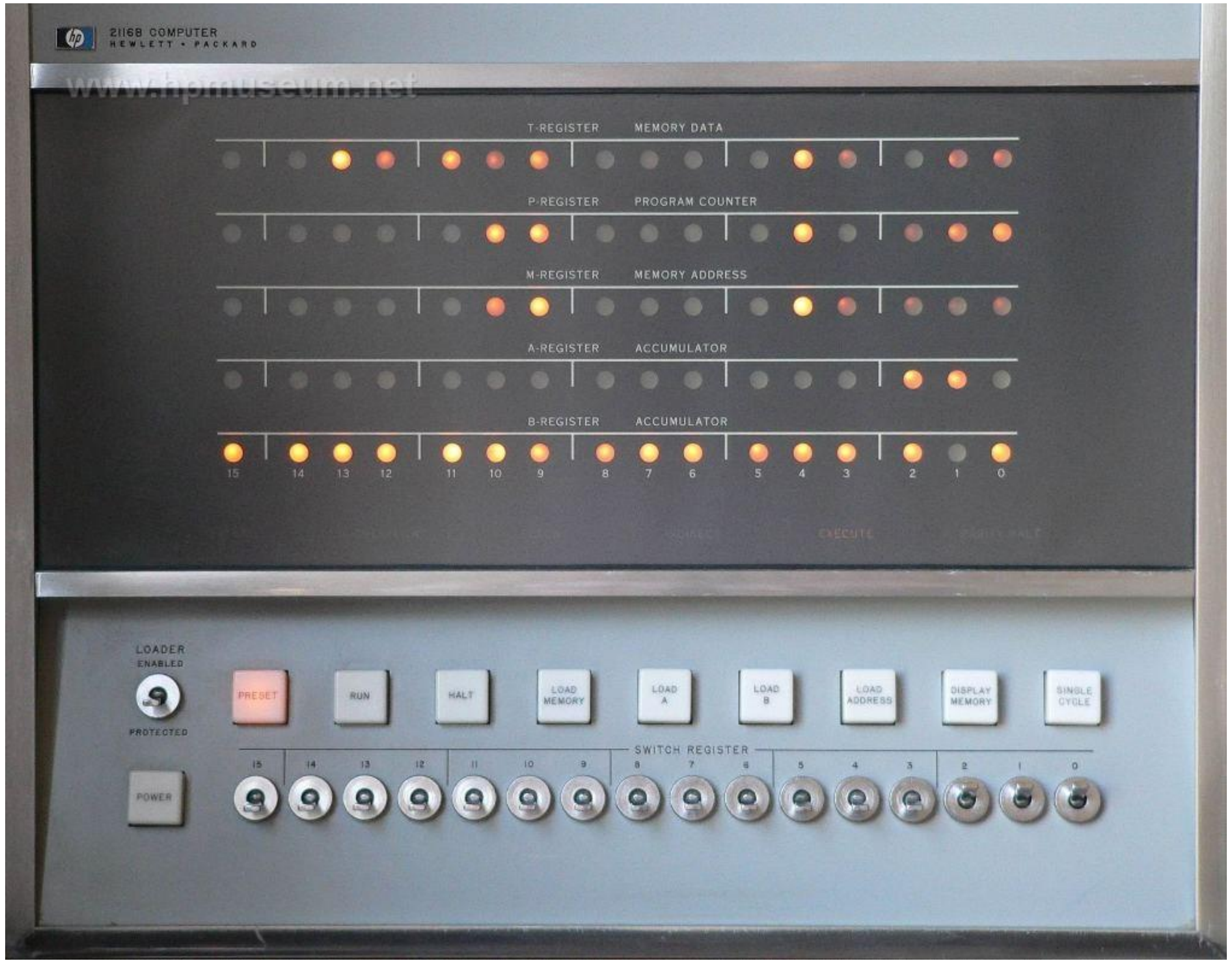
How “Unix” Became a De Facto Standard

- Unix was developed in the 1970s at Bell Labs by *Ken Thompson*, *Dennis Ritchie*, and others, but licensing was very restrictive
- An operating system – the *Berkeley Standard Distribution (BSD)* – based on Unix was developed at Berkeley and released in 1977. *Mac OS* is based on BSD.
- Another system based on Unix – *Linux* – was developed by Finnish software engineer Linus Torvalds in the early 1990s. Many Linux variants were created by other developers.
- The *IEEE Computer Society* released a family of standards – the *Portable Operating System Interface (POSIX)* – in 1997.
- Over 70% of all web servers, and approximately 98% of the top 1 million web servers, run POSIX compliant operating systems.

Machine Code – Common in 1940s

Instruction	Action
0101 1111 1111 0001	Load the value from the following address into the A register; advance the program counter by 2
0011 1110 1000 0101	(data address)
0111 1111 1111 0001	Subtract the following value from the A register; advance the program counter by 2
0000 0000 0000 1100	(value = 12)
0110 1111 1111 0001	Store the value from the A register into the following address; advance the program counter by 2
0011 1110 1000 0101	(data address)
0110 1010 1111 0001	Compare the following value to the A register; if A is less than or equal to the value, jump to address
0000 0000 0001 0100	(value = 20)
0100 1000 1000 0110	(program address)

HP 2116 – circa 1966



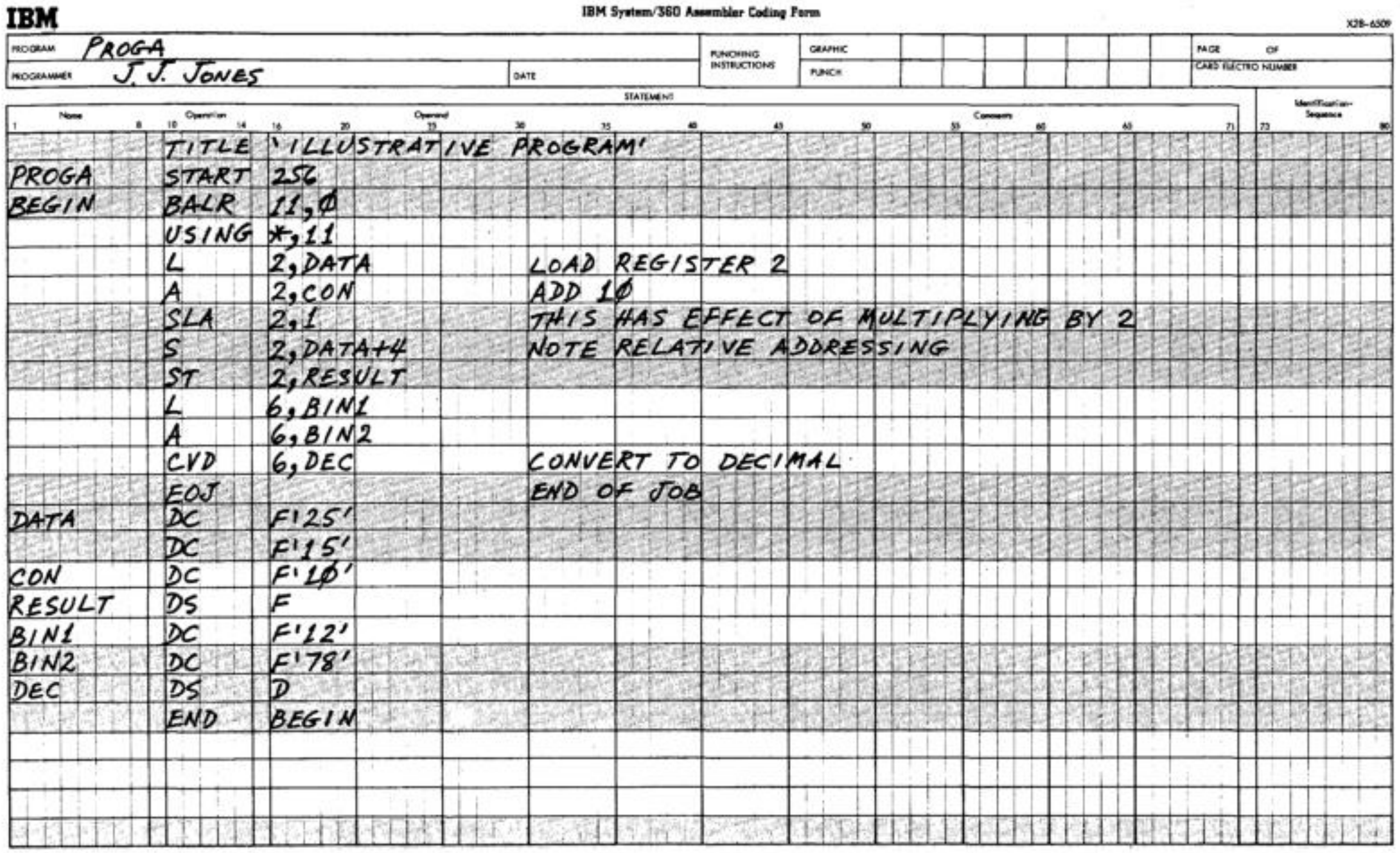
Problems with Machine Code Programming

- Really hard and error prone
- Really hard to “read” code
- Tied to machine architecture
- Need to remember what each address is used for
- Not relocatable – if you want to load things at a different place in memory, addresses change

Assembly Language

Instruction	Action
LDA inventory	Load the value from the specified location into the A register
SBA 12	Subtract a value (12) from the A register
STA inventory	Store the value from the A register into the specified location
CMPA 20, low_inventory	Compare the A register to a value (20); if A <= 20, go to the address “low_inventory”
• • •	
low_inventory:	

Assembly Language Coding Sheet



IBM

IBM System/360 Assembler Coding Form

X28-6509

PROGRAM PROGA

PROGRAMMER J. J. JONES

DATE

PUNCHING INSTRUCTIONS GRAPHIC PUNCH

PAGE OF

CARD ELECTRO NUMBER

STATEMENT

Name 1 8 · Operation 10 14 · Operand 16 20 25 30 35 40 45 50 55 · Comments 60 65 71 · Identification-Sequence 73 80

```
         TITLE 'ILLUSTRATIVE PROGRAM'
PROGA    START 256
BEGIN    BALR  11,Ø
         USING *,11
         L     2,DATA              LOAD REGISTER 2
         A     2,CON               ADD 1Ø
         SLA   2,1                 THIS HAS EFFECT OF MULTIPLYING BY 2
         S     2,DATA+4            NOTE RELATIVE ADDRESSING
         ST    2,RESULT
         L     6,BIN1
         A     6,BIN2
         CVD   6,DEC               CONVERT TO DECIMAL
         EOJ                       END OF JOB
DATA     DC    F'25'
         DC    F'15'
CON      DC    F'1Ø'
RESULT   DS    F
BIN1     DC    F'12'
BIN2     DC    F'78'
DEC      DS    D
         END   BEGIN
```

Assembly Language

- Easier to remember and code, less error prone
- Locations identified by name
- Identifiers can be mapped to different memory addresses (this is one of the things a loader does
- **Problems**: Still tied to machine architecture – you can't take an assembly language program and run it on a different machine

Higher Level (compiled) Language

Instruction	Action
inventory = inventory - 12;	Subtract 12 from the value in the specified location
If (inventory <= 20) goto low_inventory;	In the value of inventory is less than or equal to 20, jump to the address “low_inventory”
• • •	
low_inventory:	

<u>Pros</u>
- Much easier and less error prone
- Not tied to a particular machine (Grace Hopper’s idea!)

<u>Cons</u>
- Early compilers generated inefficient code

Compiler Inefficiencies

Instruction	Action
inventory = inventory - 12;	LDA inventory SBA 12 STA inventory
If (inventory <= 20) goto low_inventory;	LDA inventory CMPA 20, low_inventory
• • •	
low_inventory:	

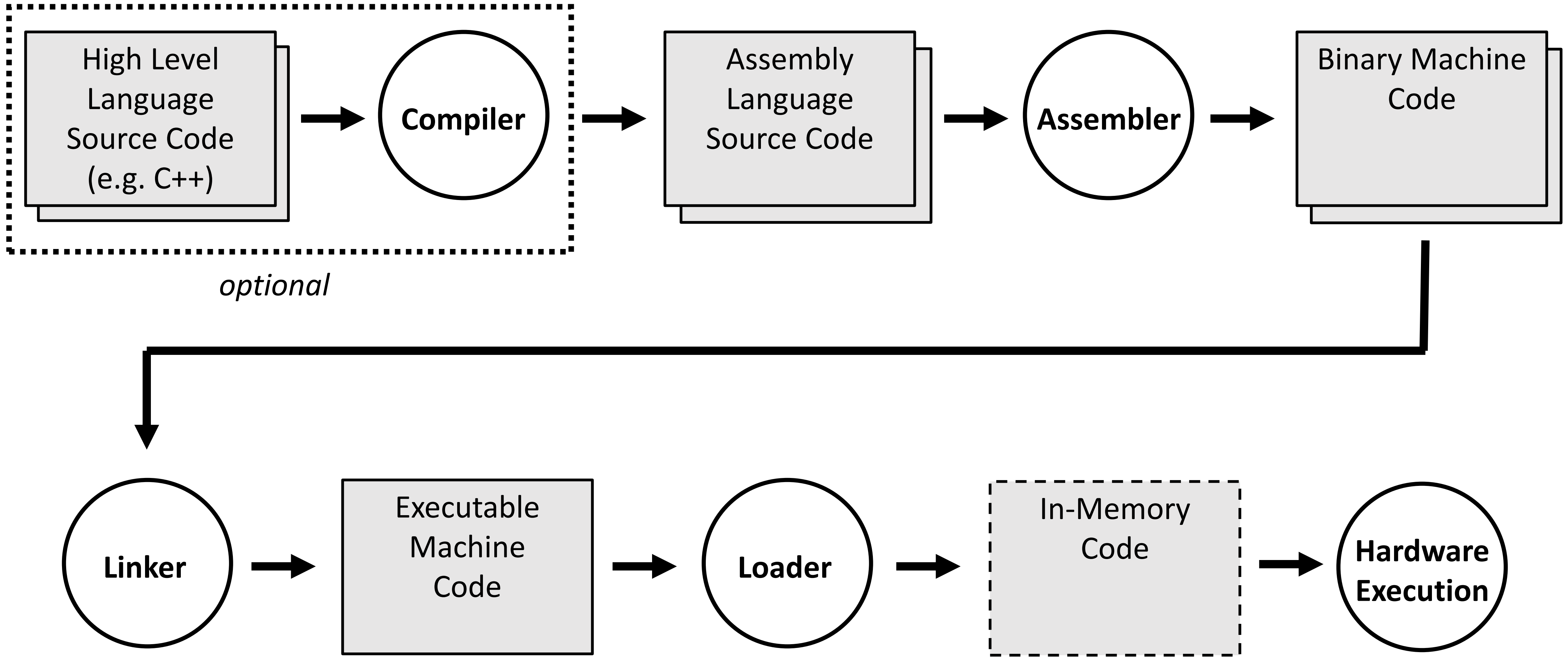
Inefficient Code

- Literal, line-by-line translation (smart, optimizing compilers weren't developed until years later)
- In this example, we generate *five* instructions instead of *four* – a 25% increase in memory requirements

Linkers and Loaders

- Linkers combine machine code modules into a single program, allowing programs to be written in pieces. Linkers also allow the development of shared *libraries*.
- Loaders load machine code program files into memory, adjust addresses, and initialize registers.

Building Software



Interpreters

- Interpreters implement programming languages.
- Like a compiler, an interpreter parses the language.
- Rather than emitting assembly language, the interpreter immediately executes that language statements.
- Interpreters eliminate the need for assemblers, linkers, and loaders, but the programs run much slower than compiled programs that have been converted to machine code.

Think About It

- Our company makes the LunaVac Model 1 computer. We've spent years developing assemblers, linkers, loaders, compilers, and a *lot* of application software… all written in C++.
- The hardware guys have just completed the new LunaVac Model 2 computer… but the architecture and instruction set are VERY different!
- How do we get our software to run on the Model 2?

Step One: Write Model 2 Assembler

- Devise an assembly language instruction set for the Model 2
- Using C++ (or another high level language), write an assembler for the Model 2 instruction set that generates Model 2 machine code
- We can now write and assemble programs on a Model 1, and emit machine code for the Model 2

Step 2: Write a Compiler for the Model 2

- Write a compiler in C++ that compiles C++ into Model 2 assembly language
- We now have a path to write C++ code, which we can then compile to emit Model 2 assembly code, which we can then assemble to generate Model 2 binary instructions

  This is called “cross compiling”

Step 3: Write (or port) Code for the Model 2

- Write machine-specific code like system software using C++
- Some code – things like editors, application programs, and utility programs – may not even need to be re-written. If they were originally written in C++, they can simply be “cross compiled” to move them to the new machine.

Course Goals and Requirements

- One of the important goals of this class is to understand how system software components fit together and the problems that they solve, so that you can use them to address real problems.
- This is not a programming-intensive class. Programming exercises will be very short. But if you don’t have a good grasp of programming, you will have a very hard time following the *concepts,* and keeping up with the assignments.

Shell scripting

Shell Scripting

- A “shell” is a command-line interface to a computer system. The shell implements a high-level interpreted programming language.
- On POSIX systems, the shell is directly accessible from the console or any authorized terminal device.
- On MacOS, the shell is accessed through the *Terminal* program.
- On Windows, the shell is accessed through the *Command Prompt*.
- Shell commands can be entered directly, or loaded from a file.
- The most commonly used shell is called *bash* (the Born Again SHell), and that’s the one I’ll use for my examples.

History

- The first Unix shell, called *sh*, was developed by Ken Thompson at Bell Labs in 1971. Thompson, along with Dennis Ritchie, invented and popularized the Unix operating system. (Ritchie was also the inventor of the *C* programming language.)
- Sh was completely rewritten by Stephen Bourne in 1979, who created the Bourne Shell. A number of spinoff and replacement shells were subsequently developed, including ksh, csh, tcsh, and bash.

Ken Thompson

Bash Basics

- Bash has a number of built-in commands that can be executed from the command line or in a script file.
- If the command line input does not correspond to a built-in shell command, the shell will look for an *executable program* or *shell script file* of that name.
- The shell has a built-in variable called *path*. Path is an ordered list of directories (separated by “;”) that bash will search for executable programs.
- Arguments can be passed to shell commands, or to programs launched from the shell, by simply appending them on the command line (with spaces as separators):
  command arg1 arg2 arg3

Bash Basics (cont)

- Many shell commands accept options, indicated by a leading “-” character
  ls –al directory1 directory2 directory3
- Multiple commands or programs can be executed from a single line, using the “;” character as a separator:
  command; program1 arg1 arg2; program 2
- The “*” character may be used as a wildcard when matching filenames in commands:
  *ls *.jpg*
- A “\” character is used to *escape* special characters:
  *ls –al file*name*

The rc file

- When an interactive shell session is launched, it automatically looks for and executes a shell startup script. In the case of *bash*, this file is called *.bashrc*
- This shell startup script may be used to initialize variables, print a header message, perform cleanup activities, etc.
- One of the most common things you will want to do in your startup script is set the *path* variable, which tells the shell where to look for programs.

File References

- *“name”* refers to an unqualified program name. The shell will search for the program in the directories in the *path* variable:
  name
- *“/”* refers to the root of the file system. Subsequent / characters are used to specify subfolders or files, e.g.:
  /usr/bin/name
- *“.”* refers to the *current working directory*. To execute a local program rather than search the path, you would enter:
  ./name
- “..” refers to the parent of the current working directory:
  ../siblingdirectory/name

POSIX File Permissions

- Permissions are granted to three classes of *user*:
  - The file’s Owner
  - The file’s Group
  - World – which means everyone
- The possible permissions for each class are *read*, *write*, and *execute*.
- When applied to a directory, execute means that the directory can be searched or traversed.
- Programs inherit the permissions of the user who launched the program, unless the program has a SUID (Set User ID) flag set, in which case it inherits the permissions of the owner of the executable file.

POSIX File Permissions (cont)

- Permissions are displayed by commands such as *ls* as a nine character string, showing read/write/execute permissions for owner, group, and world. A "-" character indicates non-permission:
  rwxrw-r--
  Owner: read, write execute
  Group: read, write
  World: read
- Permissions are also represented as a three digit octal string; each bit represents one of the permissions:
  0764

Useful Shell Commands

- **pwd** – print the current working directory
- **cd** – change the current working directory:
  cd ../../documents
- **ls** – list directory contents. By default, the command will list the contents of the current working directory. Optionally, the command accepts a directory name:
  ls ../documents
  - ls accepts many options, prefaced by a "-" character. Useful options include:
    -a – list *all* files (by default, filenames starting with a "." are not listed
    -l – list files in *long* format (show ownership, permissions, and date)
    -r – list files recursively

Useful Shell Commands (cont)

- **chmod** – change of permissions on files
  chmod 750 file1 file2 file3
- **cat** – concatenate the specified files, and send the result to STDOUT (the standard output)
  cat file1 file2 file3
- **grep** – search the designated file and output lines matching the specified regular expression
  grep expression file
- **history** – list the history of commands executed in the current shell session

Useful Shell Commands (cont)

- **!!** – repeat the previous command
- **!*(chars)*** – repeat the most recent command starting with the designated characters:
  !ls
- ***!(number)*** – repeat the specified command number. Bash keeps a numerically indexed list of all commands executed in the current session, starting with 1.
  !173
- ^(pattern)^(replacement) – repeat the previous command, replacing instances of pattern with the replacement string:
  ^Tuesday^Wednesday

Standard File Streams

- When POSIX systems launch a program, the system automatically opens three file streams:
  - STDIN – standard input
  - STDOUT – standard output
  - STDERR – standard error reporting stream
- Bash allows you to assign these streams when programs are launched, using the "<" and ">" characters.
- Execute a program and assign the contents of a file to STDIN
  program <file
- Execute a program and direct STDOUT to a file (replacing the contents)
  program >file

Standard File Streams (cont)

- Execute a program and direct STDOUT to a file (appending to the contents):
  program >> file
- Execute a program and direct STDERR to a file:
  program 2> file
- File direction can be combined with arguments:
  program arg1 arg2 <myinput >myoutput

Pipelines (aka Pipes)

- Multiple programs can be executed from a single line. The STDOUT stream of each program can be attached to the STDIN stream on the next using the "|" character:
  program1 argument | program2 | program3

Variables

- Bash allows the setting and retrieval of variables.
- To set a variable, simply append an equal sign after its name (no spaces):
  today=Wednesday
- To set a value including spaces, quote the string:
  today="Wednesday, January 8, 2020"
- To reference a variable, prepend a dollar sign to its name:
  echo $today
- To append to a variable, reference it and follow the reference with the additional contents:
  path=$path;/usr/bin/local

Conditionals

- By convention, programs return a 0 for success and a 1 for failure.
- **command1 && command2** – execute command2 if – and only if – command 1 succeeds
- **command1 || command2** – execute command2 if – and only if – command 1 fails
- **[[condition]]** – The bracket characters are used to evaluate the condition and return success or failure:
  [[$today==Wednesday]]

Conditional Expressions

Expression	True if:
[[string1 == string2]]	string1 is equal to string2
[[string1 != string2]]	string1 is not equal to string2
[[num1 –eq num2]] -ne, -lt, -le, -gt, -ge	num1 is equal to num2 Not equal, less than, less than or equal, greater than, greater than or equal
[[-e file]]	File exists
[[-r file]]	File is readable
[[-w file]]	File is writeable
[[-x file]]	File is executable
[[file1 -nt file2]]	File1 is more recent than file2

Conditional Blocks: if

Block statements can be used in scripts, but not on the command line.

- **if *condition* then** - Execute statements if the condition is true

```
if condition then
        statements
else
        statements
fi
```

```
if [[ $today == “Wednesday” ]]; then
        echo “It\’s Wednesday”
else
        echo “It\’s not Wednesday”
fi
```

Loop Blocks: for

Block statements can be used in scripts, but not on the command line.

- **for *variable* in *values* do**

  statements

  done – iterates through values

  ```
  for day in Mon Tue Wed Thu Fri do
          echo $day
  done
  ```
- Values may be specified as a range:

  ```
  for value in {1..100} do
          echo $value
  done
  ```

Loop Blocks: while

Block statements can be used in scripts, but not on the command line.

- **while [*condition*] do**

  statements

  done – iterates while condition is true

```
while [ $i –lt 100] do
        i = $[$i+1]
        echo $i
done
```

make

What is Make?

- *Make* is an application to organize and automate the process of compiling programs.
- Make builds programs based on instructions in a file, called a *makefile*.
- A makefile contains:
- A list of dependencies (e.g. a hierarchical list files that go into building a program)
- Instructions for building the program and its components

History

- Make was created by Stuart Feldman, at Bell Labs, in 1976
  - Feldman received the *2003 ACM Software System Award* for the authoring of this widespread tool
- Ken Thompson and Dennis Ritchie were not the only people working on Unix!
- Prior to make, most programmers created shell scripts to build complex programs
- Make is now part of the POSIX standard, so makefiles are reasonably portable

Dependencies: Example

- A program – prog.exe – is dependent on several object files
  prog.exe depends on module1.o, module2.o, and module3.o
- Each object file depends on a source file and an include file.
  module1.o depends on module1.c and module1.h
  module2.o depends on module2.c and module2.h
  module3.o depends on module3.c and module3.h

Instructions: Example

- There are instructions (command lines) for:
  - Building each of the object files by compiling the source and include files
  - Building the program file by linking the object files

Using Make

- When make is run it checks the modification dates of all dependencies, and executes the necessary instructions to build the program
- For example, if only module2.c has changed, then:
  module2.o will be rebuilt by compiling module2.c
  the program will be rebuilt by linking module1.o. module2.o, and module3.o
- Note that module1.o and module3.o will *not* be rebuilt, because the underlying source files on which they depend have not changed

Makefile: Example of a Make Rule

- target: dependencies

```
target: dependencies
	command 1
	command 2
	command 3
hellomake: hellomake.c hellofunc.c
	gcc -o hellomake hellomake.c hellofunc.c -I.
```

- The first line lists the dependencies for the program *hellomake*
- The second line lists the command line for building *hellomake* (the target)
- Command lines *must* begin with a tab character (not spaces)
- You can have multiple rules – separated by blank lines - in a makefile

Invoking Make

- To build the program, type the make command on the command line:
  make
  - Make will look for a file called “makefile” in the current working directory
  - The target *on the first line* will be built
  - That first rule may include other *targets* as *dependencies*, thus triggering those rules
- You can include several independent programs in a makefile, in which case you would need to specify which program to build:
  make hellomake

Make Macros

- Make supports macros, which are similar to variables

```
CC=gcc
CFLAGS=-I.
hellomake: hellomake.o hellofunc.o
	$(CC) $(CFLAGS) -o hellomake hellomake.o hellofunc.o
```

- Note that we don't list hellomake.c as a dependency for hellomake.o.
- We also don't provide a command line for building hellomake.o.
- These rules are *built in* to make. You may not be comfortable depending on built-in rules; feel from to include explicit rules.

Predefined Macros

Macro	Definition	Default	Macro	Definition	Default
AS	Assembler	as	ASFLAGS	Flags for assembler	(none)
CC	C compiler	cc	CFLAGS	Flags for C compiler	(none)
CXX	C++ Compiler	c++	CXXFLAGS	Flags for C++ compiler	(none)
CPP	C pre-processor	$(CC) -E	CPPFLAGS	Flags for C pre-processor	(none)
FC	Fortran 77 compiler	f77	FFFLAGS	Flags for Fortran compiler	(none)
GET	Extract a file from SCCS	get	GFLAGS	Flags for SCCS	(none)
LINT	Run Lint on source code	lint	LINTFLAGS	Flags for lint	(none)
PC	Pascal compiler	pc	PFLAGS	Flags for Pascal compiler	(none)
RM	Remove a file	rm -f	LDFLAGS	Flags for C/C++ loader	(none)

Generic Rules

- If you don’t want to rely on built-in make rules, you can create your own generic rules:

```
CC=gcc
CFLAGS=-I.
DEPS = hellomake.h

hellomake: hellomake.o hellofunc.o
	$(CC) -o hellomake hellomake.o hellofunc.o

%.o: %.c $(DEPS)
	$(CC) -c -o $@ $< $(CFLAGS)
```

- **$@** - automatic variable that contains the target name
- **$<** - automatic variable that contains the dependencies name

Rules You Should Include

- These are *conventions* that experienced users expect in every makefile
- **all:** useful if your makefile includes several top-level program
  all: program1 program2 program3
- **clean:** delete all intermediate files (such as .o files, libraries, etc) and then make the top level target.
  - This is useful when something just isn't working right and you suspect you may have a corrupted object file, incorrect modification dates, etc.
- **install:** install the targets in appropriate directories; set permissions; etc

Make for Everything

- Make is not limited to building programs!
- For example, you could use make to generate a report. The commands might include doing several database queries, and concatenating the output of each into a report file.

For Next Week

- Recommended: Read chapter 1 of the text
- Log in to Canvas and:
  - Submit a copy of your transcript, with the prerequisites highlighted
  - Download the Academic Integrity Pledge, sign it, and submit it
- Note that the Transcript and the Academic Integrity Pledge are required by the department; if you don’t submit them, you will be dropped from the class.