Programs

Part 4

1

Compilers, Assemblers & Linkers

Programs, Coding, and Nerds… oh my!

2

Compilers & Assemblers

- When you hit "compile" or "run" (e.g. in your Java IDE), many actions take place *"behind the scenes"*
- You are usually only aware of the work that the parser does

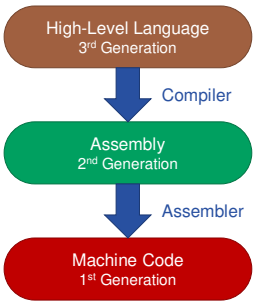
3

Development Process

1. Write program in high-level language
2. Compile program into assembly
3. Assemble program into objects
4. Link multiple objects programs into one executable
5. Load executable into memory
6. Execute it

4

From Abstract to Machine



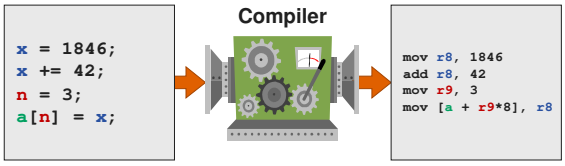
5

Compiler

- Convert programs from high-level languages (such as C or C++) into assembly language
- Some create machine-code directly…
- *Interpreters*, however…
  - never compile code
  - Instead, they run parts of their own program

6

Compilers: 3rd → 2nd Generation



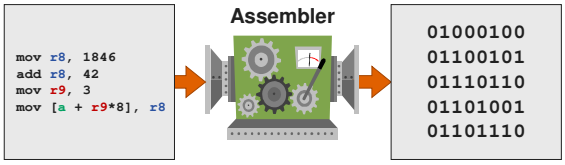
7

Assembler

- Converts assembly into the binary representation used by the processor
- Often the result is an *object file*
  - usually not executable - yet
  - contains computer instructions and information on how to "link" into other executable units
  - file may include: relocation data, unresolved labels, debugging data

8

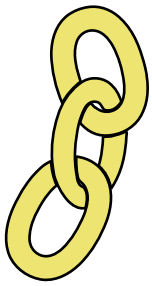
Assembler: 2nd → 1st Generation



9

Linkers

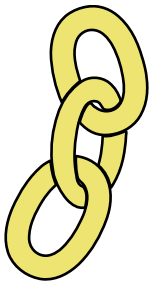
- Often, parts of a program are created separately
- Happens *more often than you think* – almost always
- Different parts of a program are called *objects*
- A *linker* joins them into a single file



10

What a Linker Does

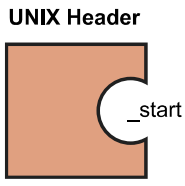
- Connects labels (identifiers) -used in one object - to the object that defines it
- So, one object can call another object
- A linker will show an error if there are label conflicts or missing labels



11

Linking your program

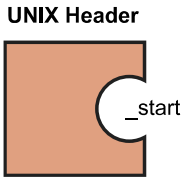
- UNIX file header defined by crt1.o and crti.o
- They are supplied behind the scenes, *so you don't need to worry about them*



12

Linking your program

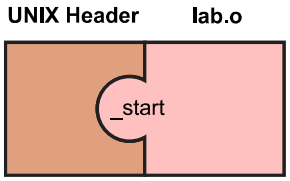
- It references a subroutine called _start
- But… it is not defined in the header
- It is used to start your program (main in Java)



13

Linking your program

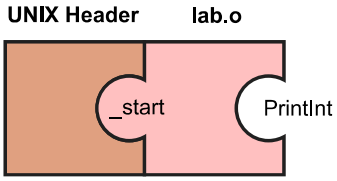
- Your program supplies this subroutine
- The linker connects the two, so the header calls your subroutine



14

You will use my library

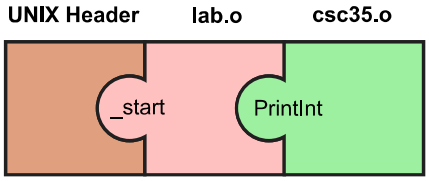
- To make labs easier, you will use my library
- Your program will reference its subroutines



15

You will use my library

- Once the object file "csc35.o" is linked, the program is complete



16

Assembly Basics

The beautiful language of the computer

17

Assembly Language

- *Assembly* allows you to write machine language programs using easy-to-read text
- Assembly programs is based on a specific processor architecture
- So, it won't "port"

18

Assembly Benefits

1. Consistent way of writing instructions
2. Automatically counts bytes and allocates buffers
3. *Labels* are used to keep track of **addresses** which prevents common machine-language mistakes

19

1. Consistent Instructions

- Assembly combines related machine instructions into a single notation *(and name)* called a *mnemonic*
- For example, the following machine-language actions are different, but related:
  - register → memory
  - register → register
  - constant → register

20

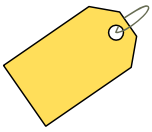
2. Count and Allocate Buffers

- Assembly automatically counts bytes and allocates buffers
- Miscounts (when done by hand) can be very problematic – and can lead to hard to find errors

21

3. Labels & Addresses

- Assembly uses *labels* to store **addresses**
- Used to keep track of locations in your programs
  - data
  - subroutines (functions)
  - …and much more



22

Battle of the Syntax

- The basic concept of assembly's notation and syntax hasn't changed
- However, there are two major competing notations
- They are *just* different enough to make it confusing for students and programmers *(who are used to the other notation)*

23

Battle of the Syntax

- AT&T Syntax
  - dominate on UNIX / Linux systems
  - registers prefixed by %, values with $
  - receiving register is last
- Intel Syntax
  - *actually created by Microsoft*
  - dominate on DOS / Windows systems
  - neither registers or values have a prefix
  - receiving register is first

24

AT&T Example (not x86)

```
# Just a simple add

mov $42, %b       #b = 42
mov value, %a     #a = value
add %b, %a        #a += b
```


25

Intel Example (not x86)

```
# Just a simple add

mov b, 42         #b = 42
mov a, value      #a = value
add a, b          #a += b
```


26

Assembly Program Structure

How these little beasties are organized

27

Assembly Programs

- Assembly programs are divided into two sections
- *data section* allocate the bytes to store your constants, variables, etc…
- *text/code section* contains the instructions that will make up your program

28

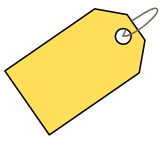
Directives

- A *directive* is a special command for the assembler
- Notation: starts with a period
- What they do:
  - allocate space
  - define constants
  - start the text or data section
  - define the "start" address

29

Labels

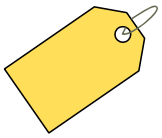
- You can define *labels* by following an identifier with a colon
- As the assembler is reading your program, it is generating machine code instructions and storage



30

Labels

- Anytime the assembler sees a label declaration, it will save the current address (at that point) into a table
- This allows you to use the label anywhere in your assembly, and it will be replaced by that address



31

Hello World – Using csc35.o

```
.intel_syntax noprefix
.data
message:
   .ascii "Hello World!\n\0"

.text
.global _start

_start:
    lea   rbx, message
    call  PrintCString

    call  EndProgram
```

32

Hello World – Using csc35.o

```
.intel_syntax noprefix
.data
message:
   .ascii "Hello World!\n\0"
```

Data Section

```
.text
.global _start

_start:
    lea   rbx, message
    call  PrintCString

    call  EndProgram
```


33

Data Section

Use Intel format

No prefix characters

```
.intel_syntax noprefix
.data
message:
    .ascii "Hello World!\n\0"
```

Start data section

34

Data Section

```
.intel_syntax noprefix
.data
message:
    .ascii "Hello World!\n\0"
```

Create a label called 'message'. It will store an address.

Allocate the bytes required to store text

35

Hello World – x86, Linux

```
.intel_syntax noprefix
.data
message:
   .ascii "Hello World!\n\0"

.text
.global _start

_start:
    lea   rbx, message
    call  PrintCString

    call  EndProgram
```

Text / Code Section

36

Text / Code Section

```
.text
.global _start

_start:
    lea   rbx, message
    call  PrintCString

    call  EndProgram
```

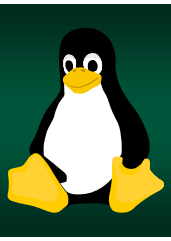
Start text section

Make label visible to the linker. Header will call _start

Loads the Effective Address 'message' into rbx

Call the library subroutine (it needs an address in rbx)

37



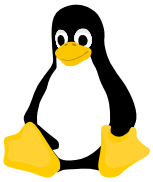
Basics of UNIX

Feel the pow-wah of the dark side

38

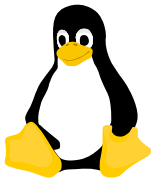
Basics UNIX

- UNIX was developed at AT&T's Bell Labs in 1969
- Design goals:
  - operating system for mainframes
  - stable and powerful
  - but not exactly easy to use – GUI hadn't been invented yet



39

Basics UNIX



- There are versions of UNIX with a nice graphical user interface
- A good example is all the various versions of Linux
- However, all you need is a command line interface

40

Command Line Interface

- Command line interface is text-only
- But, you can perform all the same functions you can with a graphical user interface
- This is how computer scientists have traditionally used computers

```
>gcc hello.c
>ls
a.out hello.c

>a.out
Hello world!
```


41

Command Line Interface

- Each command starts with a name followed by zero or more arguments
- Using these, you have the same abilities that you do in Windows/Mac

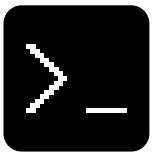
Spaces separate name & arguments

name ***argument1 argument2*** **…**

42

`ls` Command

- Lists all the files in the current directory (folder)
- It has arguments that control how the list will look
- Folder names will have a slash suffix
- Programs have an asterisk suffix



43

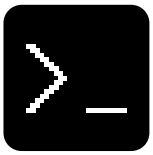
`ls` Command

```
> ls
a.out*  csc35/  html/  mail/
test.asm
```


44

`ll` Command

- This command is a shortcut notation for `ls -l`
- It displays all the files in "long" format
- Besides the filename, its size, access rights, etc… are displayed



45

`ll` Command

```
> ll

-rwx------  1 cookd othcsc  4650 Sep 10 17:44 a.out*
drwx------  2 cookd othcsc  4096 Sep  5 17:49 csc35/
drwxrwxrwx 10 cookd othcsc  4096 Sep  6 11:04 html/
drwxrwxrwx  2 cookd othcsc  4096 Jun 20 17:58 mail/
-rw-------  1 cookd othcsc    74 Sep 10 17:44 test.asm
```


46

Please Wait

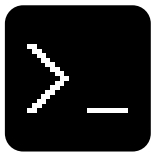
CSC 35 will begin in a moment

Please open the chat Window

47

`mkdir` Command

- This command will "make a directory"
- You will want to create one to store your CSc 35 work



48

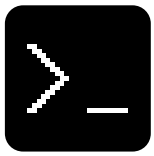
mkdir Command

```
> ls
a.out*  html/  mail/  test.asm

> mkdir csc35

> ls
a.out*  csc35/  html/  mail/  test.asm
```


49

cd Command

- To change your current folder, you will use the "change directory" command
- If you specify a folder name, you will move into it
- If you use .. (two dots), you will got to the parent folder



50

cd Command

```
> cd csc35

> cd ..
```

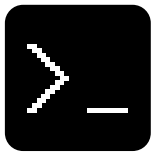
Move into csc35 folder

Return to parent folder

51

rm Command

- If you want to delete a file, you can use the "remove" command
- It's good to cleanup your folders from time to time
- It can also delete multiple files using patterns



52

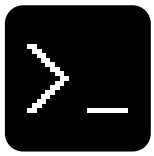
rm Command

```
> ls
a.out*  html/  mail/  test.asm

> rm a.out

> ls
html/  mail/  test.asm
```


53

nano Application

- Nano is the UNIX text editor (well, the best one – that is)
- It is very similar to Windows Notepad – but can be used on a terminal
- You will use this to write your programs



54

nano Application

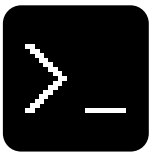
- Nano can create new file (use a new name)
- It can also open an existing file to edit

```
nano filename
```


55

as Command

- This is the GNU assembler
- It will take an assembly program and convert it into an object
- You will be alerted of any syntax errors or unrecognized mnemonics (typos)



56

as Command

- The **-o** specifies the next name listed is the output file
- So, the second is the output file (object)
- The third is your input (assembly)

```
as -o lab.o lab.asm
```


57

as Command

- **Be very careful** – if you list your input file first, it will be destroyed
- There is no "undo" in UNIX!
- Check the two extensions for "o" then "asm"

```
as -o lab.o lab.asm
```


58

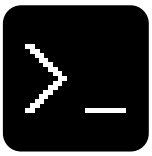
as Command

```
> ls
lab.asm

> as -o lab.o lab.asm

> ls
lab.asm  lab.o
```


59

ld Command

- This is the GNU linker
- It will take one (or more) objects and link them into an executable
- You will be alerted of any unresolved labels



60

`ld` Command

- The `-o` specifies the next name is the output
- The second is the output file (executable)
- The third is your input objects (1 or more)

```
ld -o a.out csc35.o lab.o
```


61

`ld` Command

- Be very careful – if you list your input file (an object) first, it will be destroyed
- I will provide the "csc35.o" file

```
ld -o a.out csc35.o lab.o
```


62

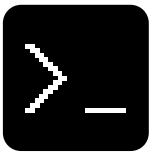
`ld` Command

```
> ls
lab.o  csc35.o

> ld -o a.out lab.o csc35.o

> ls
lab.o  csc35.o  a.out*
```


63

`alpine` Application

- Alpine is an e-mail application
- Has an easy-to-use interface similar to Nano
- You will use this software to submit your work



64

`alpine` Application

- To run Alpine, just type its name at the command line
- There are no arguments
- You will have to login (again)

```
alpine
```


65