Part 3

Programs

Assembly Basics

The beautiful language of the computer

High-Level Programming

- You are used to writing programs in high level programming languages
- Examples:
  - C#
  - Java
  - Python
  - Visual Basic

High-Level Programming

- These are *third-generation languages*
- They and are designed to isolate you from architecture of the machine
- This layer of abstraction makes programs "portable" between systems

Machine Language

- Instructions, that are *actually* executed on the processor, are a series of bytes
- Bytes contain *encoded* instructions
  - each instruction is in a compact binary form
  - easy for the processor to interpret and execute
  - some instructions are take more bytes than others – not all are equal

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"

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 a common machine-language mistake

1. Consistent Instructions

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

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

3. Labels & Addresses

- Assembly uses *labels* are used to store addresses
- These can be memory locations or parts of your running program
- They are automatically calculated when the assembler is creating machine code

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 use to the other notation)*

Battle of the Syntax

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

AT&T / GNU Example (not x86)

```
# Just a simple add

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


Intel Example (not x86)

```
; Just a simple add

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


Assembly Program Structure

How these little beasties are organized

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 processor instructions that will make up your program

Labels

- As the machine code is created, the assembler keeps track of the current address
- You can define *labels*
  - will be assigned an *address*
  - … of the program created up to that point
- Notation: end in a colon

Literals – the dollar sign

- Literals are denoted using a dollar sign $ prefix
- This is commonly used for constants and to get the actual value of a label (an address)
- A common mistake is to forget it

Registers – the percent sign

- Registers are using a percent sign % prefix
- If a percent sign is left off, the assembler will think you typed a label
- The explicit notation is actually useful – albeit odd looking

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

Hello World – Using csc35.o

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

.text
.global _start

_start:
    mov   $message, %rax
    call  PrintCString

    call  EndProgram
```


Hello World – Using csc35.o

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

.text
.global _start

_start:
    mov   $message, %rax
    call  PrintCString

    call  EndProgram
```

Data Section

Data Section

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

Start data section

Create a label called 'message'. It is an address.

Allocate the bytes required to store text

Hello World – x86, Linux

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

.text
.global _start

_start:
    mov   $message, %rax
    call  PrintCString

    call  EndProgram
```

Text / Code Section

Text / Code Section

```
.text
.global _start

_start:
    mov   $message, %rax
    call  PrintCString

    call  EndProgram
```

Start text section

Make visible to the linker. Header will call _start

Move the address of message into eax

Call the library subroutine

Compilers, Assemblers & Linkers

Programs, Coding, and Nerds… oh my!

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

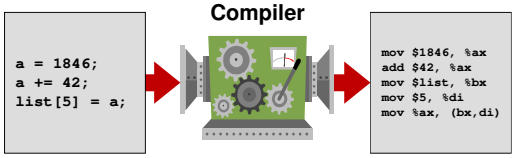
Development Process

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

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

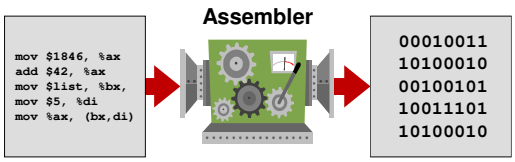
Compilers



Assembler

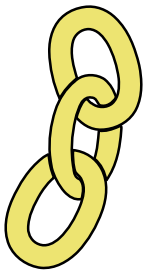
- Converts assembly into the binary representation used by the processor
- Often the result is an *object file*
  - usually not executable - yet
  - contains computer instructions and information on how to "link" into other executable units
  - file may include: relocation data, unresolved labels, debugging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

Assembler



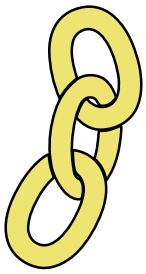
Linkers

- Often, parts of a program are created separately
- Happens *more often than you think* – almost always
- A *linker* joins multiple parts (usually object files) into a single file



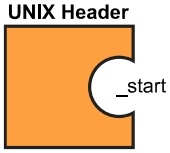
What a Linker Does

- Connects labels (identifiers) - used in one object - to the object to that defines it
- So, one object can call another object
- What you will see: label conflicts and missing labels



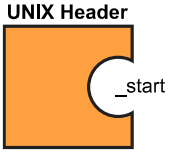
Linking your program

- 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



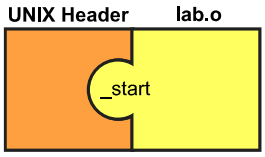
Linking your program

- It references a subroutine called _start and is used as the executable entry point
- But… it is not defined in the header



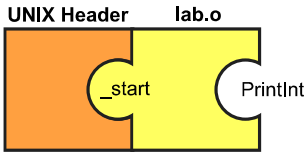
Linking your program

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



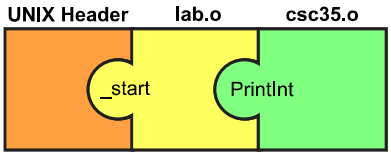
You will use my library

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



You will use my library

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

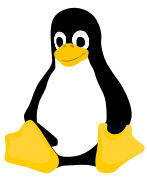


Basics of UNIX

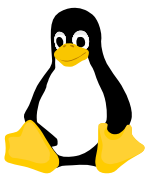
Feel the pow-wah of the dark side

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



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

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!
```

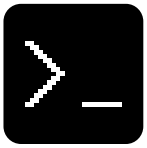

Command Line Interface

- 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

```
name «argument 1» «argument 2» …
```


`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

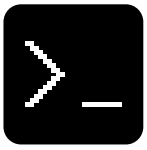


`ls` Command

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


`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

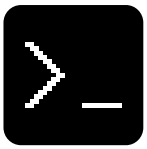


`ll` Command

```
> 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.s
```

`mkdir` Command

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

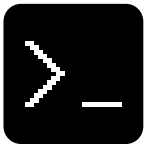


`mkdir` Command

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

> mkdir csc35

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


`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



`cd` Command

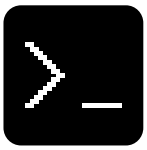
```
> cd csc35

> cd ..
```

Move into csc35 folder

Return to parent folder

`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

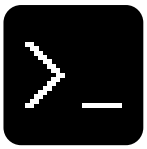


`rm` Command

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

> rm a.out

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

nano

- 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



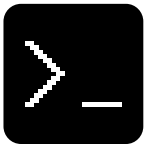
nano

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

```
nano filename
```


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)



as Command

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

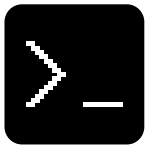
```
as -o lab.o lab.s
```


as Command

```
> ls
lab.s

> as -o lab.o lab.s

> ls
lab.s  lab.o
```


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



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
```


ld Command

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

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


ld Command

```
> ls
lab.o  csc35.o

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

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