Computer Systems Org - 009 Recitation

Arahant Ashok Kumar

Myself…

- Arahant (please refer to me by my 1st name :))
  - CS Grad student @ Courant.
  - Entrepreneurship, Quantum Computing, Biz analytics
  - Avid biker, love traveling, writing and a dog-person
- Email: aak700@nyu.edu
- Recitation - Mondays, 11 am to 12:15 pm
- Office hours - Tuesdays, 5pm to 6pm
  - Zoom link in NYU classes
- Seating arrangements
  - Choose one and **stick to it**
- Communication - Email, NYU classes (forums)
- Let's get started…

Basic Unix/Linux commands

- **man** - manual
- ls - list
- cd - change dir
- pwd - current dir
- mkdir - make an empty dir
- cp - copy
- mv - move
- rm - remove
- echo - write arguments to the standard output
- cat - output content of a file
- wc - word count
- grep - pattern matching
- touch - create a file

- Google/ man
- https://github.com/jlevy/the-art-of-command-line

Git

- Git config
  - https://stackoverflow.com/questions/35942754/how-to-save-username-and-password-in-git-gitextension
  - git config -- global credential.helper store
  - git pull
- Git commands
  - git clone <url>
  - git status
  - git add
  - git commit -m “<your message>”
  - git push origin <branch>
  - git pull origin <branch>

Linserv1 - Logging in

- You may program your labs on any system, but they **must** be tested on linserv1 servers, where it will be graded
- ssh <netid>@access.cims.nyu.edu
- ssh linserv1

Linserv1 - copy

- Secure copy
  - scp -r /full/path/to/folder netid@access.cims.nyu.edu:/home/<netid>
  - scp /full/path/to/file netid@access.cims.nyu.edu:/home/<netid>

- Git - copying from local machine to linserv1 (I'd recommend this)
  - local machine > Git add > Git commit > Git push > repo
  - repo > git clone > git pull origin <branch> (master) > cims

- Git - for each new assignment
  - git clone <url>
  - git add file1.c file2.c
  - git commit
  - git push

Linserv1 - testing

- The default gcc compiler on linserv1 is 4.8.5 (gcc —version)
- module avail gcc
- module load gcc-6.3.0 #loads gcc 6.3.0
- module unload gcc-6.3.0 #reverts back to default gcc

What is a compiler?

- C, which a high level language, is for people, not computers
  - In fact, high level languages are for **people**
  - Computer processors only understand **binary instructions**
- A compiler translates code between languages
  - In our cases, it translates from C (the source language) to machine code (the target language)

Source code - `main.c`

Machine code
(binary instructions)

How to use a compiler?

- Consider a simple C program:

main.c

```
#include<stdio.h>
int main( ) {
    printf("Hello World!\n");
    return 0;
}
```

- To run this program, we must first compile it
  - Can use `gcc: gcc main.c` will produce a file called `a.out`
  - We can run a.out by issuing `./a.out`
  - You can choose the name of executable with -o, as in `gcc main.c -o myprogram`

Potential challenge

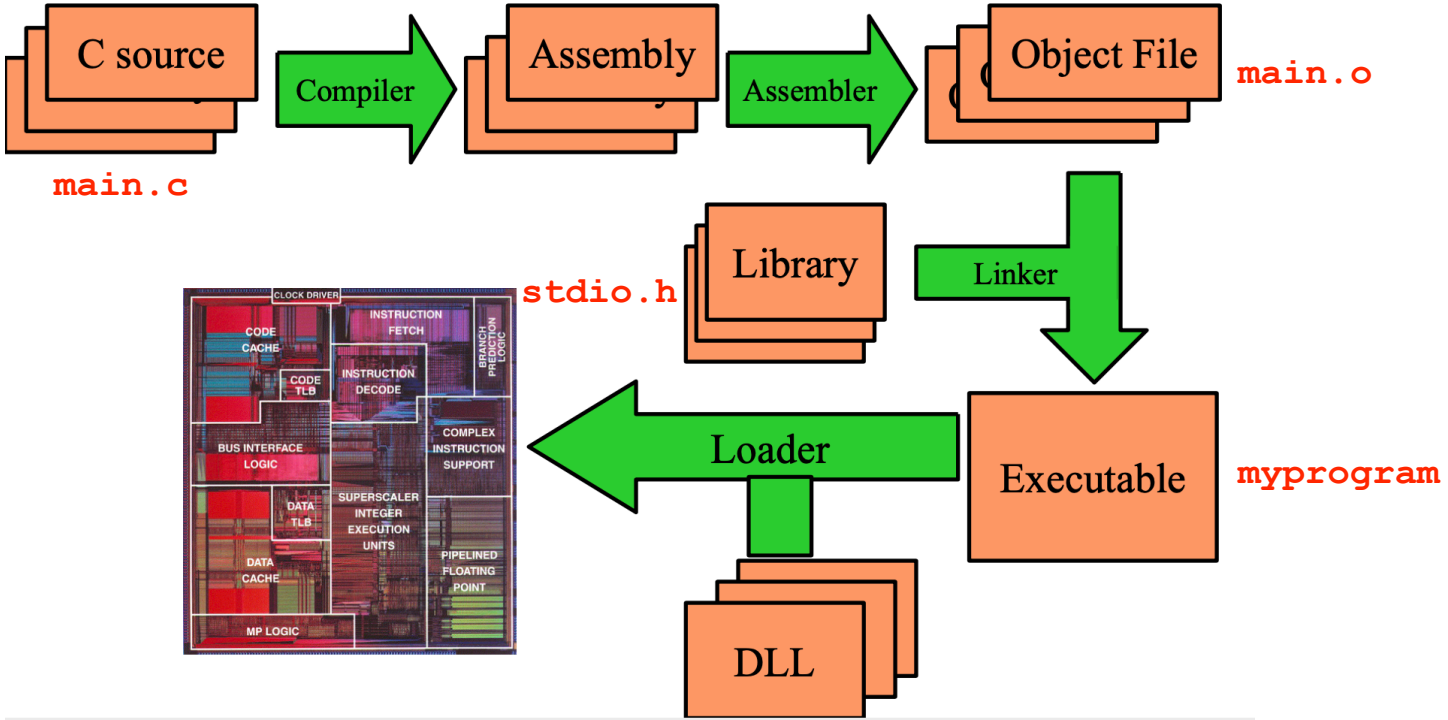
- For large projects, recompiling everything can be slow
- To avoid this, we can compile files separately, **without linking** using the -c flag
    1. Ordinarily, the compiler compiles each source file into object code
    2. Then links them and
    3. Deletes the intermediate object code
- `gcc -c main.c util.c`
    - Will create `main.o` and `util.o`
    - Then we can add to main.c and util.c and not have to recompile the other
    - We can later do link by running `gcc main.o util.o -o myprogram`

Compiling and Linking

- Compiling isn't quite the same as creating an executable file!
- Instead, creating an executable is a multistage process divided into two components: compilation and linking
- Compilation:
  - Refers to the processing of source code files and the creation of an 'object' file
  - Merely products the machine language instructions
- Linking:
  - Refers to the creation of a single executable file from multiple object files

- .c is the source code
- .o extension is for an object file - this is a binary file generated by compilation unit
- executable files should have no extension (by default a.out, which has the .out extension)

Source code to execution - stages



Makefile

- In the real world, things are so elaborate and complicated…
  - The Linux kernel has over 45,000 files of C code - it uses almost 2700 Makefiles
- Make will also know when we need to recompile different sources
- Make builds projects for us, keeping track of when it needs to recompile or not
  - When make recompiles, each modified C source file must be recompiled
  - If any source file has been recompiled, all the object files must be linked together to produce the new executable
- We tell make about the dependencies in our code in the Makefile
- Makefile consists of a number of ‘rules’:

Makefile

- Rules specify:
  - A target, which is usually the name of a file that is generated by a program
    - Targets include main.o or myprogram
  - Dependencies, which are files that are used as input to create the target
    - main.o needs main.c
    - myprogram needs main.o and util.o
  - Commands, which are actions that make carries out
    - gcc -c main.c -o main.o
- Rules look like this:

```
myprogram: main.o util.o
   gcc main.o util.o -o myprogram
```

- However, the rule that specifies commands for the target need not have dependencies, for example “clean”
  - make clean
- A clumsy Makefile…

```
myprogram: main.c util.c
   gcc main.c util.c -o myprogram
```

Makefile

A good Makefile:

```
myprogram: main.o util.o
  gcc main.o util.o -o myprogram
main.o: main.c
  gcc -c main.c -o main.o
util.o: util.c
  gcc -c util.c -o util.o
clean:
  rm -f main.o util.o myprogram
```

- Make supports pattern matching with the %
  - %.c means all .c files
- Make has “automatic variables”
  - The meaning of variables within a rule is contextual
  - $@ is the name of the rule
  - $^ is the list of dependencies
- Example:

```
%.o: %.c
  gcc -c $^ -o $@
```