Makefiles

CS 0449: Introduction to System Software

CS0449 Teaching Assistants

https://pitt.edu/~shk148/

University of Pittsburgh | School of Computing and Information

Meta-Notes

- These slides were adapted heavily from recitation slides created by *Martha Dixon* who was a teaching assistant (TA) for this course in Fall of 2020. They contain materials which were obtained from various sources, including, but not limited to, the following:

[1] J. Misurda, CS 0449: Introduction to Systems Software, 3rd ed. Pittsburgh, PA: University of Pittsburgh, 2017.

[2] S. J. Matthews, T. Newhall, and K. C. Webb, Dive into Systems: A Gentle Introduction to Computer Systems. San Francisco, CA: No Starch Press, 2022.

[3] R. Bryant, D. R. O'Hallaron, and M. S., Computer Systems: A Programmer's Perspective. Princeton, NJ: Pearson, 2016.

[4] L. Oliveira, V. Petrucci, and J. Misurda, in Introduction to Systems Software, 2022

Makefiles

Automating and Optimizing Builds

Why and goal

- Multiple files can be compiled independently and then merged together in a process called **linking**.
  - Generally, these two phases use different tools behind the scenes.
- Project 1.
  - Write a Makefile to compile multiple files.

A Brief Overview Of Makefiles

- What is make?
  - The make utility is a software tool for managing and maintaining computer programs consisting many component files. The make utility automatically determines which pieces of a large program need to be recompiled, and issues commands to recompile them
  - Make reads its instruction from Makefile (called the descriptor file) by default.
  - Makefile is a way of automating software building procedure and other complex tasks with dependencies.
  - Makefile contains: dependency rules, macros and suffix(or implicit) rules.
- How does it work?
  - The relationships are described in a file named “Makefile” [by default]
    - You can name it differently, but it’s not current practice!
    - https://www.gnu.org/software/make/manual/make.html#Makefile-Names
  - Make will look into that file, and follow the rules described
- Allows us to create custom settings and compile multiple files quickly with a single command (make)

Rules

- Rules specifying how to make files
  - How to make a file is specified by a **recipe**
  - **Target** is the file created using the recipe
  - Targets have **prerequisite files**
  - Prerequisites can be made by another rule

Example Rule:

```
target: prerequisites
    recipe
```

```
prog: main.o
    gcc -o prog main.o
main.o: main.c
    gcc -c main.c
```

Target main depends on `main.o` (that is created by another rule) and it's made by invoking `gcc -o prog main.o`

Because the Makefile has a rule to generate main.o…
But you still need main.c

Example: without makefile

```
int hellomake(){
    return 0;
}
```

- Compile this code:
  - **gcc -c hellomake.c**
  - **gcc -o hellomake hellomake.o**

The “-c” argument to gcc will create a **hellomake.o object file** instead of link an entire executable.

We can now **link** the object file with the C standard library and create an executable called **hellomake** using this line.

https://www.cs.colby.edu/maxwell/courses/tutorials/maketutor/

Example: without makefile

hellomake.c

```
#include <hellomake.h>

int main() {
  // call a function in another file
  myPrintHelloMake();

  return(0);
}
```

hellofunc.c

```
#include <stdio.h>
#include <hellomake.h>

void myPrintHelloMake(void) {

  printf("Hello makefiles!\n");

  return;
}
```

hellomake.h

```


/*
example include file
*/

void myPrintHelloMake(void);
```

```
gcc -c hellomake.c

gcc -c hellofunc.c

gcc -o hellomake hellomake.o hellofunc.o
```

https://www.cs.colby.edu/maxwell/courses/tutorials/maketutor/

Example: with makefile

hellomake.c

```
#include <hellomake.h>

int main() {
  // call a function in another file
  myPrintHelloMake();

  return(0);
}
```

hellofunc.c

```
#include <stdio.h>
#include <hellomake.h>

void myPrintHelloMake(void) {

  printf("Hello makefiles!\n");

  return;
}
```

hellomake.h

```
/*
example include file
*/

void myPrintHelloMake(void);
```

Makefile

```
hellomake: hellomake.o hellofunc.o

	gcc -o hellomake hellomake.o hellofunc.o

 hellomake.o: hellomake.c

	gcc -c hellomake.c

hellofunc.o: hellfunc.c

	gcc -c hellofunc.c
```

Hint: Indentation are strictly tabs.

https://www.cs.colby.edu/maxwell/courses/tutorials/maketutor/

More On MakeFiles

- We can do a lot with MakeFiles, using custom rules and commands, variables, functions, conditional expressions, and more…

- Read more about them on the course website
  - https://cs0449.gitlab.io/fa2023/resources/
- Official documentation
  - https://www.gnu.org/software/make/manual/make.html

Your Turn

https://cs0449.gitlab.io/fa2023/resources/worksheets/makefiles/makefiles.pdf