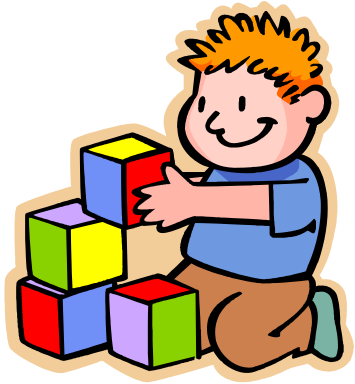
Topics for this Lecture

- Build systems
- Static analysis



Topic 1: Build Systems

- In a simple world, compiling and running a computer program is simple:
  - `> gcc –o myexe myprogram.c`
  - `> ./myexe`

- The world is not that simple most of the time, as you may notice if you've tried compiling any open source programs

Topic 1: Build Systems

- The steps of compilation
  1. Pre-processing
  2. Compilation
  3. Assembly
  4. Linking

Topic 1: Build Systems

Vi print.c

```c
#include <stdio.h>
#define STRING "Hello World"
int main(void)
{
/* Using a macro to print 'Hello World'*/
printf(STRING);
return 0;
}
```

```
$ gcc -Wall print.c -o print
$ ./print
Hello World
```

Topic : Pre-processing

1. *Macro substitution*
2. *Comments are stripped off*
3. *Expansion of the included files*

```
#include <stdio.h>
#define STRING "Hello World"
int main(void)
{
/* Using a macro to print 'Hello
World'*/
printf(STRING);
return 0;
}

gcc -Wall -E print.c
```

```
# 846 "/usr/include/stdio.h" 3 4
extern FILE *popen (__const char *__command, __const
char *__modes) ;

# 886 "/usr/include/stdio.h" 3 4
extern void flockfile (FILE *__stream) __attribute__
((__nothrow__));

# 916 "/usr/include/stdio.h" 3 4
# 2 "print.c" 2

int main(void)
{
printf("Hello World");
return 0;
}
```

Topic : Compilation

1. *Take print.i as input.*
2. *Produce an intermediate compiled output.*
3. *The output file for this stage is ‘print.s’.*
4. *The output is assembly level instructions.*

```
# 846 "/usr/include/stdio.h" 3 4
extern FILE *popen (__const char
*__command, __const char *__modes)
;

# 886 "/usr/include/stdio.h" 3 4
extern void flockfile (FILE *__stream)
__attribute__ ((__nothrow__));

# 916 "/usr/include/stdio.h" 3 4
# 2 "print.c" 2

int main(void)
{
printf("Hello World");
return 0;
}
```

```
.file "print.c“
………..
main:
.LFB0:
.cfi_startproc
pushq %rbp
.cfi_def_cfa_offset 16
movq %rsp, %rbp
.cfi_offset 6, -16
.cfi_def_cfa_register 6
movl $.LC0, %eax
movq %rax, %rdi
movl $0, %eax
call printf
……………
```

Topic : Assembly

1. *Take print.s as input.*
2. *Produce an intermediate compiled output.*
3. *The output file for this stage is ‘print.o’ is the object file.*
4. *The output is machine level instructions.*

```
.file "print.c“
……….
main:
.LFB0:
.cfi_startproc
pushq %rbp
.cfi_def_cfa_offset 16
movq %rsp, %rbp
.cfi_offset 6, -16
.cfi_def_cfa_register 6
movl $.LC0, %eax
movq %rax, %rdi
movl $0, %eax
call printf
```

```
$ vi print.o
^?ELF^B^A^A^@^@^@^@^@^@^@^
@^@^A^@>^@^A^@^@^@^@^@^@
^@^@^@^@^@^@^@^@^@^@^@^
@^@0^
^@UH<89>å¸^@^@^@^@H<89>Ç¸Hell
o World^@^@GCC: (Ubuntu 4.4.3-
4ubuntu5) 4.4.3^@^
```

How to Compile a Program

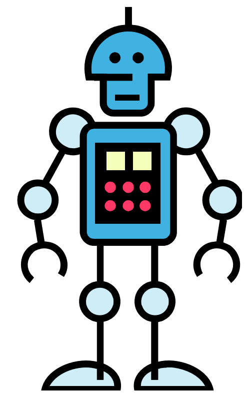
- Most larger programs require many complex commands with many arguments:
  - `> gcc -g -c lib1.c -DARCH_X86 -DLINUX -DDEBUG -D -ftest-coverage -fprofile-arcs -O0`
  - `> gcc -g -c lib2.c -DARCH_X86 -DLINUX -DDEBUG -D -ftest-coverage -fprofile-arcs`
  - `> gcc -o mainexec m.c lib1.o lib2.o  -DARCH_X86 -DLINUX -DDEBUG -D -ftest-coverage -fprofile-arcs –lm –DNO_X –O3`
- Compiling all the components of a modern software system may take *a long time*
  - Building the Linux kernel

Not Just a Shell Script

- Simply bundling all the compilation into a script doesn't solve the problem
  - If you only change one file, which other files have to be recompiled? Do you start over?
  - A script is a series of commands; if you want to take advantage of multiple machines, you have to design the parallelism yourself

Build Systems

- Again, automation comes to our rescue
- Build systems:
  - Given a description including at least:
    - Components of a system (files)
    - How they depend on each other
    - How to produce the ones that are not provided by humans
  - A build system:
    - Uses information on which files have been modified and which files don't exist yet to produce the final products – for example, executables – for a system



"Let the robot do the boring stuff!"

Build Systems

- Lots of different build systems
  - Some are very simple, don't do much beyond what I just described
  - Others are very complex, attempt to determine dependencies for you, automatically parallelize compilation, etc.
    - Sometimes integrated with figuring out local configuration (hardware, operating system, available tools)
    - Sometimes integrated with source control or testing

- In this class, we'll use a very simple system, *make*

Simple Structure of a Makefile

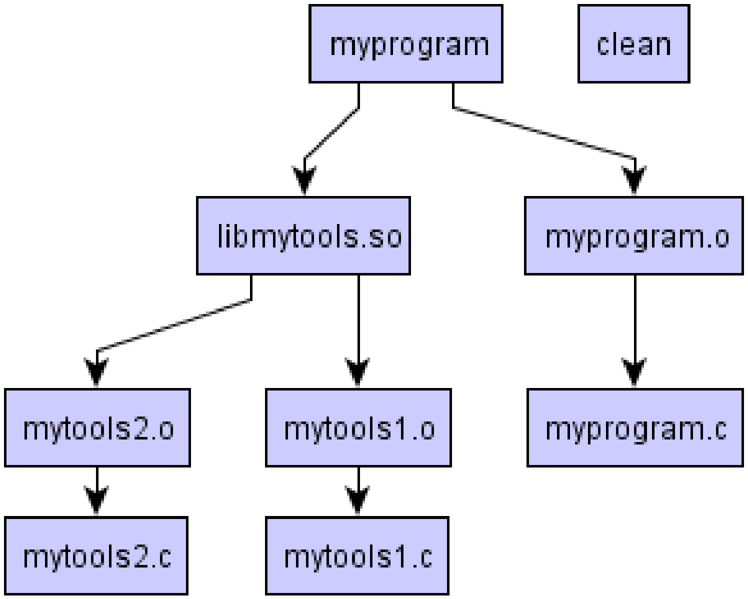
- See dominion/Makefile in the class repository
- Structure is like this:

```
<targetfile1>:  <dependfile1> <dependfile2>
   <command to create targetfile1>
   <command to create targetfile1>
<targetfile2>: <dependfile3> <dependfile4>
   <command to create targetfile2>…
```

Simple Structure of a Makefile

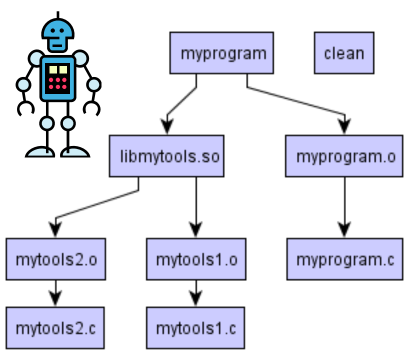
- Textual representation of a graph:

```
myprogram: libmytools.so myprogram.o
        …
libmytools.so: mytools2.o mytools1.o
        …
mytools1.o: mytools1.c
        …
mytools2.o: mytools2.c
        …
myprogram.o: myprogram.c
        …
Clean:
        rm -rf *.o *.so myprogram
```



Simple Structure of a Makefile

- Typing
  - `> make <targetfile>`
  - Tries to create <targetfile>
    - In particular, it first checks all the things <targetfile> depends on
      - If any don't exist, they are created
      - If any are older than things they depend on, they are re-generated



Topic 2: Static Analysis

- Before we get to testing (our first big main topic) want to discuss another method for finding bugs
  - Analyze the source code for bad “patterns”
  - Happens to some extent every time you build a program
    - Your compiler has to analyze the code to compile and optimize it
    - `gcc -Wall` will warn you about some problems that might show up in testing

What is Static Analysis?

- Called “static” analysis because it analyzes your program without running it
  - Analysis that runs the program is called “dynamic” analysis (testing is the most common dynamic analysis)
- Differs in a few key ways:
  - Static analysis can catch bugs without a test case – just by structure of code
  - Static analysis can give “false positives” – warn you about a problem that can’t actually show up when the program runs

Static Analysis: Not Just Compilers

- While the compiler does some limited "bug hunting" during compilation, that's not its main job
  - There are dedicated tools for analyzing source code for bugs
  - A few such tools include:
    - Uno (open source, available on the web)
    - Coverity (paid software, quite pricey but very powerful, used by NASA and others)
    - Klocwork
    - CodeSonar
- Won't say much about these in this class, because they are typically fairly easy to use, just run them on your code

Static Analysis: Not Just Compilers

- Testing, on the other hand, requires more work from the programmer/test engineer
- So why not prefer static analysis in general?
  - Static analysis is generally limited to simple properties – don’t reference null pointers, don’t go outside array bounds
  - Also good for some security properties
  - But very hard/impossible to check things like “this sort routine really sorts things”

Static Analysis: Not Just Compilers

- - Predicate abstraction
- - Shape analysis
- - Taint analysis
- - Program slicing
- - Alias analysis