The Make language (cont'd)

Implicit rules

Make already knows how to compile C, C++, Fortran, and many other languages:

- Implemented as pattern rules for common file extensions (e.g. `.c`, `.cpp`, `.o`)
- These "implicit rules" are automatically present any time you run `make`
  - Even when you don't have a `Makefile` at all!
- They support limited customization
  - Recipe can be altered through predefined variables like `CC`, `CFLAGS`, and `LDFLAGS`
  - Dependencies can be altered by writing a rule definition with no recipe (see next slide)

`.c` to `.o` implicit rule:

```
%.o: %.c
    $(CC) $(CPPFLAGS) $(CFLAGS) -c
```

Using implicit rules

Not the target of any rule! Uses the implicit `%.o: %.c` pattern rule.

```
hello: hello.o world.o foo.o bar.o
```

No recipe; uses the one from the implicit `%: %.o` pattern rule.

Note: The program has to have the same name as one of the `.o` files or else the rule won't match.

Using implicit rules (cont'd)

```
$ cat Makefile
CFLAGS := -g
hello: hello.o world.o

$ make hello
cc -g -c -o hello.o hello.c
cc -g -c -o world.o world.c
cc hello.o world.o -o hello

rm -f hello.o
rm -f world.o
$
```

From implicit rule for compiling C programs

From implicit rule for linking C programs

By default, Make doesn't keep intermediate files between two pattern rules

Pitfalls of implicit rules

Implicit rules are convenient, and you should use them if you want, but be aware of the following:

- Header files not automatic prerequisites (but you can add them yourself)

```
hello.o: hello.c hello.h
```

- Not explicit, which some people and projects value

Silencing commands with @

- When you run a command, Make prints the command (not its output—the command itself) before running it
- Can be useful for debugging, but sometimes the command's only purpose is to print something else (e.g. `echo`)
- To prevent the command from being printed, preface its line in your recipe with `@`.

```
$ cat Makefile
.PHONY: mytarget
mytarget:
  echo "hello"

$ make mytarget
echo "hello"
hello
$

<...>

$ cat Makefile
.PHONY: mytarget
mytarget:
  @echo "hello"

$ make mytarget
hello
$
```

More on variables: ?= and += assignments

?= assignment:

- Only sets the variable if it hasn't been defined yet
- Doesn't override variables that have been defined, even if they're empty
- Always recursively-expanded

+= assignment:

- Appends the given text to the end of an existing variable
- If not yet defined, recursively-expanded; otherwise, expanded as specified by the previous definition

More on variables: overrides and the environment

Variable overrides (e.g. `make CFLAGS=-g hello`):

- *Not* an environment variable: value passed as an argument to `make` directly
- Sets the given Make variable to the given value
- Causes all assignments in Makefile (=, `:=`, and +=) to be ignored

Environment variables (e.g. `export CFLAGS=-g ; make hello`):

- Tells the shell to run `make` with `CFLAGS` set in the environment
- Make imports all environment variables as Make variables
- However, assignments in the Makefile override environment variables: = and `:=` replace them entirely, while += appends to them

More on variables: implicit variables

Some variables used by implicit rules, like `CC` (the C compiler) have default values (`CC` defaults to "`cc`"):

- These default values have lower precedence than environment variables
- Can be overridden by the environment or an assignment in your Makefile

.DEFAULT_GOAL

By default, make with no arguments builds the first target in the Makefile. Specify otherwise by setting the special variable `.DEFAULT_GOAL:`

```
.DEFAULT_GOAL := sometargetname
```

Compilation and linking internals

What is compilation?

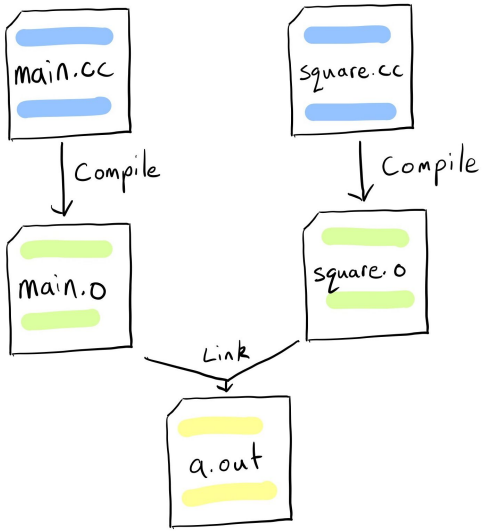
- Broadly, translating one language to another, sometimes less expressive, language
- Here, translating C source to machine code (.c to .o)

```c
int factorial(int x) {
  if (x < 2) return 1;
  return x * factorial(x - 1);
}
```

```
0000000000000000 <factorial>:
   0: f3 0f 1e fa           endbr64
   4: 55                    push   %rbp
   5: 48 89 e5              mov    %rsp,%rbp
   8: 48 83 ec 10           sub    $0x10,%rsp
   c: 89 7d fc              mov    %edi,-0x4(%rbp)
   f: 83 7d fc 01           cmpl   $0x1,-0x4(%rbp)
  13: 7f 07                 jg     1c <factorial+0x1c>
  15: b8 01 00 00 00        mov    $0x1,%eax
  1a: eb 11                 jmp    2d <factorial+0x2d>
  1c: 8b 45 fc              mov    -0x4(%rbp),%eax
  1f: 83 e8 01              sub    $0x1,%eax
  22: 89 c7                 mov    %eax,%edi
  24: e8 00 00 00 00        callq  29 <factorial+0x29>
  29: 0f af 45 fc           imul   -0x4(%rbp),%eax
  2d: c9                    leaveq
  2e: c3                    retq
```

What is linking?

- Combining multiple .o files into an executable
- "Fill in" references to functions and variables from other `.o` files.



Declares to the compiler that `logMessage()` exists. It's *defined* in `log.c`, though, so `main.o` has to be linked with `log.o` before the program will work.

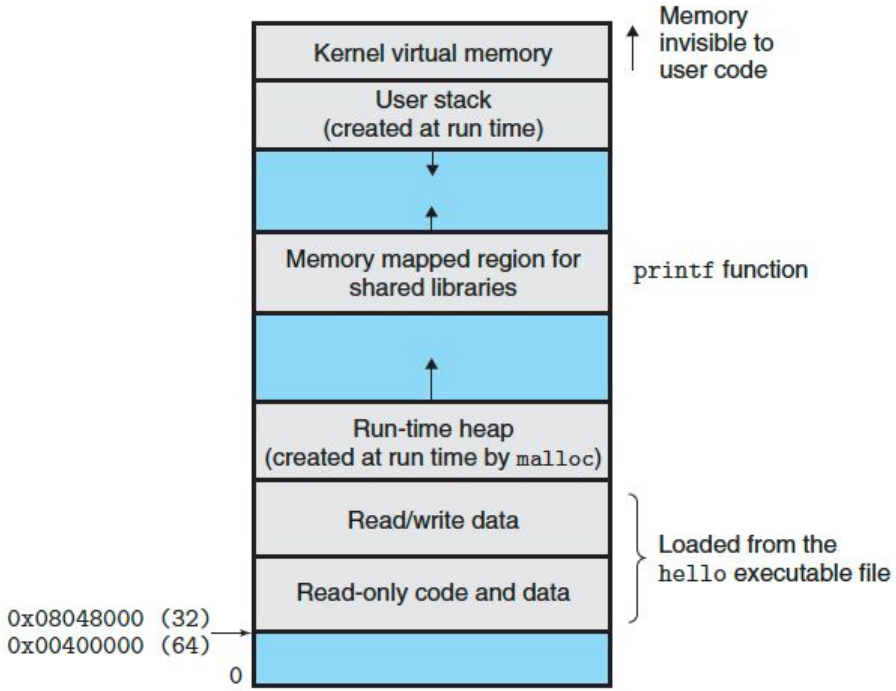
```
// main.c
void logMessage(const char *message);

int main() {
  logMessage("Hello, world!");
}
```

```
// log.c
void logMessage(const char *message) {
  fprintf(stderr, "%s\n", message);
}
```

What is loading?

- Reading on-disk machine code and data into memory

You'll get more into this in CS 40.



Single `gcc` invocations with multiple C files don’t scale

You have seen this, mentioned obliquely, earlier.

- Consider

```
main: main.c dep0.c dep1.c dep2.c
  gcc $< -o $@
```

Compile **and** link

- If you have 10 .c files and only change one, no need to rebuild them all
  - But you're doing it anyway. Why?

Separating compilation and linking into separate steps

```
main: main.o dep0.o dep1.o dep2.o
  gcc $^ -o $@
```

Just link

```
main.o: <...>
```

Just compile

```
%.o: %.c %.h
```

Just compile

Building large projects with Make

Goal: small, self-contained modules

- For projects with hundreds or thousands of files, those files will likely be organized into groups that make up individual components of the program.
- Ideally, each group should have its build rules in a single, small file.
  - Easier to find the rules you care about
  - Easier to copy between projects or split into own project
- **How can we achieve this, while still allowing the entire project to be built with a single command?**

First attempt: recursive Make

```
$ tree .
.
├── lib
│   ├── subsystem0
│   │   ├── Makefile
│   │   ├── system.c
│   │   └── system.o
│   └── subsystem1
│       ├── Makefile
│       ├── system.c
│       └── system.o
├── main
├── main.c
├── main.o
└── Makefile

3 directories, 10 files
$
```

First attempt: recursive Make

```
all: main

main: main.o subsystems
    @echo "Building main project..."
    gcc main.o lib/subsystem1/system.o lib/subsystem1/system.o -o main

subsystems:
    $(MAKE) -C lib/subsystem0
    $(MAKE) -C lib/subsystem1

clean:
    $(MAKE) -C lib/subsystem0 clean
    $(MAKE) -C lib/subsystem1 clean
    rm -f *.o main
```

Implicit variable containing the file used to invoke Make

The big problem with recursive Make

Impossible for files in one subdirectory to cleanly depend on files in another!

There are hacks to work around this, but they all introduce complexity:

1. Manually order all your submakes so that each directory is built after all the ones it depends on.
2. Run multiple submake passes, each time building the things the next one depends on.
3. Structure your code so that each subdirectory is totally self-contained.

"Recursive Make Considered Harmful":
https://accu.org/journals/overload/14/71/miller_2004/

Case study: the Linux kernel

From the "Linux Kernel Makefiles" documentation:

> "A Makefile is only responsible for building objects in its own directory. Files in subdirectories should be taken care of by Makefiles in these subdirs. The build system will automatically invoke make recursively in subdirectories, provided you let it know of them."

Recursive Make is a good fit for projects written entirely in C:

- Cross-module dependencies use header files, which don't need to be built
- Individual modules produce object files, which the higher-level Make can link together
- Corner cases still require hacks, but the volume is manageable

Second attempt: The `include` directive

```
all: main

main: main.o lib/subsystem0/system.o lib/subsystem1/system.o
    @echo "Building main project..."
    gcc main.o lib/subsystem1/system.o lib/subsystem1/system.o -o main

main.o: main.c
    gcc -c $< -o $@

include lib/subsystem0/Makefile
include lib/subsystem1/Makefile

clean: subsystem0-clean subsystem1-clean
    rm -f *.o main
```

Second attempt: The `include` directive

```
# Makefile for lib/subsystem0
lib/subsystem0/system.o: lib/subsystem0/system.c
    @echo "Building subsystem0..."
    gcc -c $< -o $@

subsystem0-clean:
    rm -f lib/subsystem0/*.o
```

Must use paths relative to the top-level directory, despite being in a subdirectory. This is due to the direct textual inclusion.

Case study: the Android Open Source Project