Basic Makefiles for Fun & Profit

Ivan Sergeev

```
git clone https://github.com/vsergeev/linux-talks.git
```

May 2018

Table of Contents

1. Introduction
2. Example Problem
3. Basic Concepts
4. Intermediate Concepts
5. Conclusion

What is make?

A general-purpose tool to build outputs from inputs, following the rules specified in a Makefile.

- Created for Unix by Stuart Feldman at Bell Labs, 1976

What is make?

A general-purpose tool to build outputs from inputs, following the rules specified in a Makefile.

- Created for Unix by Stuart Feldman at Bell Labs, 1976
- Often used for compilation, but is agnostic to objectives

What is make?

A general-purpose tool to build outputs from inputs, following the rules specified in a Makefile.

- Created for Unix by Stuart Feldman at Bell Labs, 1976
- Often used for compilation, but is agnostic to objectives
- GNU Make implementation is ubiquitous, and standard on Linux and Mac OS X

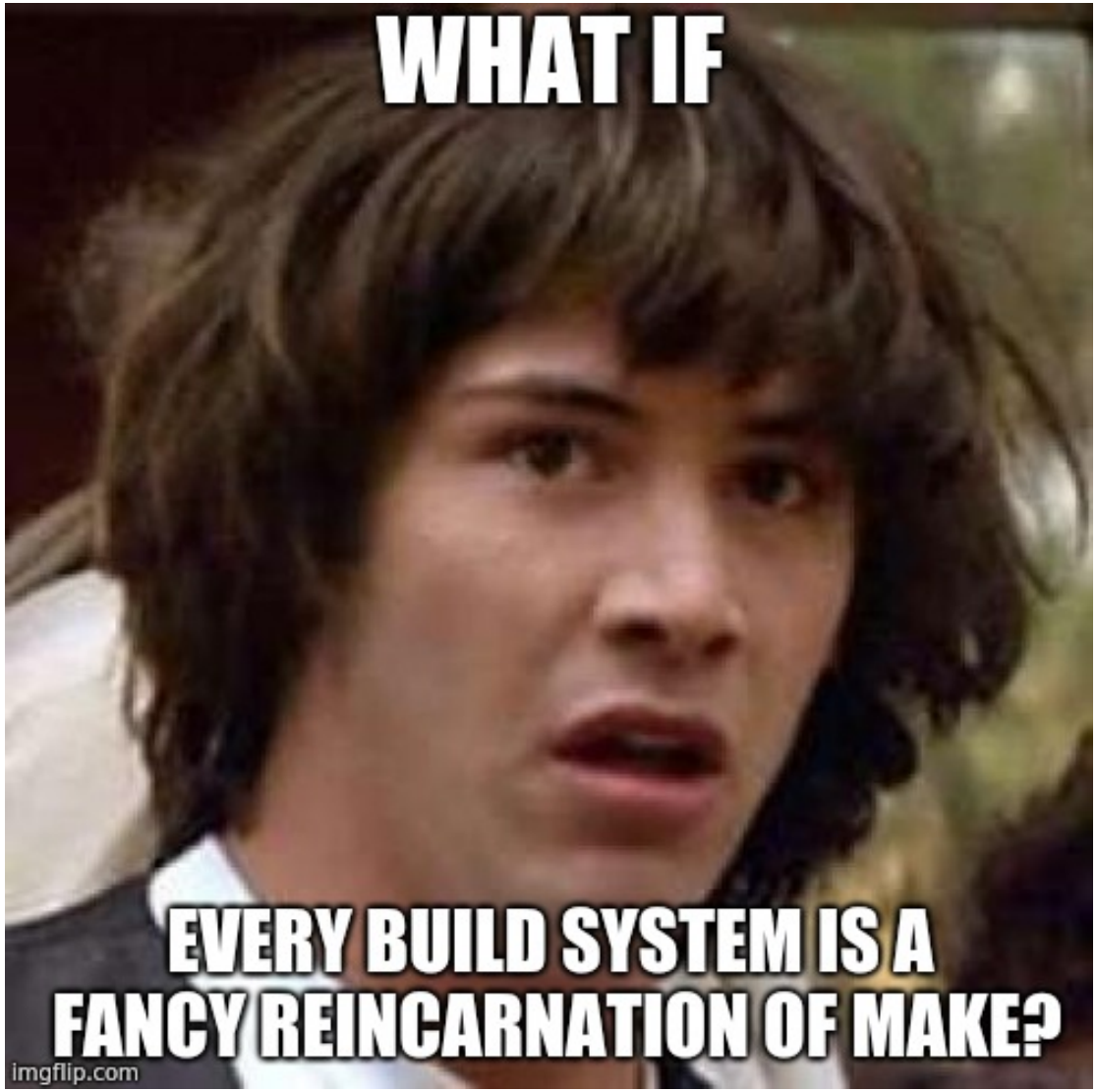
Example of Running make

Running make to build a small library project:

```
$ make
cc -std=gnu99 -pedantic   -c -o gpio.o gpio.c
cc -std=gnu99 -pedantic   -c -o spi.o spi.c
cc -std=gnu99 -pedantic   -c -o i2c.o i2c.c
cc -std=gnu99 -pedantic   -c -o mmio.o mmio.c
cc -std=gnu99 -pedantic   -c -o serial.o serial.c
cc -std=gnu99 -pedantic   -c -o version.o version.c
ar rcs periphery.a gpio.o spi.o i2c.o mmio.o serial.o version.o
$
```

Why learn `make`? Automation

Why learn `make`? Reincarnation



Example of a Makefile

```
LIB = periphery.a
SRCS = gpio.c spi.c i2c.c mmio.c serial.c version.c

CFLAGS += -std=gnu99 -pedantic
OBJECTS = $(patsubst %.c,%.o,$(SRCS))

.PHONY: all clean

all: $(LIB)

clean:
	-rm -f $(LIB) $(OBJECTS)

$(LIB): $(OBJECTS)
	ar rcs $(LIB) $(OBJECTS)

%.o: %.c
	$(CC) $(CFLAGS) $(LDFLAGS) -c $< -o $@
```

Table of Contents

1 Introduction

2 Example Problem

3 Basic Concepts

4 Intermediate Concepts

5 Conclusion

Wallpaper Gallery

Let's automate building a static site for wallpapers.

Wallpaper Gallery


beach.jpg


bridge.jpg


road.jpg


tree.jpg


mountain.jpg


flower.jpg

File structure

Project consists of wallpapers, a Python script, and a Mako template:

```
$ ls
beach.jpg   flower.jpg    gallery.py     road.jpg
bridge.jpg  gallery.mako  mountain.jpg   tree.jpg
$
```

Thumbnails

We create 320 px thumbnails with ImageMagick's `convert`:

```
$ convert -resize 320x beach.jpg beach.thumb.jpg
$ file beach.jpg beach.thumb.jpg
beach.jpg:       JPEG image data, JFIF standard 1.01,
                 resolution (DPI), density 72x72,
                 segment length 16, baseline,
                 precision 8, 5616x3744, frames 3
beach.thumb.jpg: JPEG image data, JFIF standard 1.01,
                 resolution (DPI), density 72x72,
                 segment length 16, baseline,
                 precision 8, 320x213, frames 3
$
```

Python Script

A Python script accepts the original images as arguments, and passes relevant information to a Mako template for rendering:

```
$ python3 gallery.py beach.jpg bridge.jpg mountain.jpg
                     tree.jpg flower.jpg road.jpg > gallery.html
$ file gallery.html
gallery.html: HTML document, ASCII text
$
```

Python Script

```
import sys
import collections
import mako.template

Image = collections.namedtuple('Image', ['filename', 'thumbnail'])

items = [Image(filename, filename.replace(".", ".thumb."))
         for filename in sys.argv[1:]]

template = mako.template.Template(filename="gallery.mako")
output = template.render(items=items, columns=3)
sys.stdout.write(output)
```

Gallery Template

```
<html>
<body>
<table>
<th colspan="${columns}"><h2>Wallpaper Gallery</h2></th>
% for row in range(len(items) // columns + 1):
<tr>
%   for image in items[row * columns:(row + 1) * columns]:
<td>
<a href="${image.filename}"><img src="${image.thumbnail}"></a>
<center><tt>${image.filename}</tt></center>
</td>
%   endfor
</tr>
% endfor
</table>
</body>
</html>
```

A Handcrafted, Artisanal Gallery

```
$ convert -resize 320x beach.jpg beach.thumb.jpg
$ convert -resize 320x bridge.jpg bridge.thumb.jpg
$ convert -resize 320x mountain.jpg mountain.thumb.jpg
$ convert -resize 320x tree.jpg tree.thumb.jpg
$ convert -resize 320x flower.jpg flower.thumb.jpg
$ convert -resize 320x road.jpg road.thumb.jpg
$ python3 gallery.py beach.jpg bridge.jpg mountain.jpg
                     tree.jpg flower.jpg road.jpg > gallery.html
$
```

Phew.

Shell Script

We could throw this all into a shell script, but we would still run into problems:

- Scalability

Shell Script

We could throw this all into a shell script, but we would still run into problems:

- Scalability
- Incremental builds

Shell Script

We could throw this all into a shell script, but we would still run into problems:

- Scalability
- Incremental builds
- Maintainability

Shell Script

We could throw this all into a shell script, but we would still run into problems:

- Scalability
- Incremental builds
- Maintainability

Shell Script

We could throw this all into a shell script, but we would still run into problems:

- Scalability
- Incremental builds
- Maintainability

So let's use `make` instead.

Table of Contents

1 Introduction

2 Example Problem

3 Basic Concepts

4 Intermediate Concepts

5 Conclusion

Golden Rule

Anatomy of a Makefile Rule

```
target: dependencies...
    commands
    ...
```

- `make` executes a rule's **commands** to transform its **dependencies** into a **target**

Golden Rule

Anatomy of a Makefile Rule

```
target: dependencies...
    commands
    ...
```

- `make` executes a rule's **commands** to transform its **dependencies** into a **target**
- If a dependency does not exist, `make` looks for and executes the rule responsible for creating it

Golden Rule

Anatomy of a Makefile Rule

```
target: dependencies...
    commands
    ...
```

- `make` executes a rule's **commands** to transform its **dependencies** into a **target**
- If a dependency does not exist, `make` looks for and executes the rule responsible for creating it
- `make` can be invoked to build specific target(s) with
  `make [targets...]`

Golden Rule

Anatomy of a Makefile Rule

```
target: dependencies...
    commands
    ...
```

- `make` executes a rule's **commands** to transform its **dependencies** into a **target**
- If a dependency does not exist, `make` looks for and executes the rule responsible for creating it
- `make` can be invoked to build specific target(s) with
  `make [targets...]`
- The first rule of a Makefile is the default target

Gotcha: Hard Tabs

Commands in a Makefile rule must be indented with a **hard tab**!

Not spaces!

```
$ cat Makefile
hello.txt:
	echo "Hello World" > hello.txt
$ make
Makefile:2: *** missing separator.  Stop.
$
```

```
$ cat Makefile
hello.txt:
        echo "Hello World" > hello.txt
$ make
echo "Hello World" > hello.txt
$
```

Everything is a File

```
beach.thumb.jpg: beach.jpg
    convert -resize 320x beach.jpg beach.thumb.jpg
```

In make, every target and dependency is simply a file.

Everything is a File

```
beach.thumb.jpg: beach.jpg
    convert -resize 320x beach.jpg beach.thumb.jpg
```

In make, every target and dependency is simply a file.

```
$ make beach.thumb.jpg
convert -resize 320x beach.jpg beach.thumb.jpg
$
$ make beach.thumb.jpg
make: Nothing to be done for 'beach.thumb.jpg'.
$
```

Once the target exists, there is no need to re-execute the rule. *

Starting small

Let's put together a Makefile to generate the static gallery with two images:

```
gallery.html: beach.jpg bridge.jpg \
              beach.thumb.jpg bridge.thumb.jpg \
              gallery.mako gallery.py
    python3 gallery.py beach.jpg bridge.jpg > gallery.html

beach.thumb.jpg: beach.jpg
    convert -resize 320x beach.jpg beach.thumb.jpg

bridge.thumb.jpg: bridge.jpg
    convert -resize 320x bridge.jpg bridge.thumb.jpg
```

Starting small

Let's put together a Makefile to generate the static gallery with two images:

```
gallery.html: beach.jpg bridge.jpg \
              beach.thumb.jpg bridge.thumb.jpg \
              gallery.mako gallery.py
    python3 gallery.py beach.jpg bridge.jpg > gallery.html

beach.thumb.jpg: beach.jpg
    convert -resize 320x beach.jpg beach.thumb.jpg

bridge.thumb.jpg: bridge.jpg
    convert -resize 320x bridge.jpg bridge.thumb.jpg
```

```
$ make
convert -resize 320x beach.jpg beach.thumb.jpg
convert -resize 320x bridge.jpg bridge.thumb.jpg
python3 gallery.py beach.jpg bridge.jpg > gallery.html
$
```

Breaking it down

```
gallery.html : beach.jpg bridge.jpg
               beach.thumb.jpg bridge.thumb.jpg \
               gallery.mako gallery.py
    python3 gallery.py beach.jpg bridge.jpg > gallery.html

beach.thumb.jpg: beach.jpg
    convert -resize 320x beach.jpg beach.thumb.jpg

bridge.thumb.jpg: bridge.jpg
    convert -resize 320x bridge.jpg bridge.thumb.jpg
```

```
$ make
```

Breaking it down

```
gallery.html : beach.jpg bridge.jpg
               beach.thumb.jpg bridge.thumb.jpg \
               gallery.mako gallery.py
    python3 gallery.py beach.jpg bridge.jpg > gallery.html

beach.thumb.jpg : beach.jpg
    convert -resize 320x beach.jpg beach.thumb.jpg

bridge.thumb.jpg: bridge.jpg
    convert -resize 320x bridge.jpg bridge.thumb.jpg
```

```
$ make
convert -resize 320x beach.jpg beach.thumb.jpg
```

Breaking it down

```
gallery.html : beach.jpg bridge.jpg
               beach.thumb.jpg bridge.thumb.jpg \
               gallery.mako gallery.py
    python3 gallery.py beach.jpg bridge.jpg > gallery.html

beach.thumb.jpg: beach.jpg
    convert -resize 320x beach.jpg beach.thumb.jpg

bridge.thumb.jpg: bridge.jpg
    convert -resize 320x bridge.jpg bridge.thumb.jpg
```

```
$ make
convert -resize 320x beach.jpg beach.thumb.jpg
```

Breaking it down

```
gallery.html: beach.jpg bridge.jpg
              beach.thumb.jpg bridge.thumb.jpg \
              gallery.mako gallery.py
    python3 gallery.py beach.jpg bridge.jpg > gallery.html

beach.thumb.jpg: beach.jpg
    convert -resize 320x beach.jpg beach.thumb.jpg

bridge.thumb.jpg: bridge.jpg
    convert -resize 320x bridge.jpg bridge.thumb.jpg
```

```
$ make
convert -resize 320x beach.jpg beach.thumb.jpg
convert -resize 320x bridge.jpg bridge.thumb.jpg
```

Breaking it down

```
gallery.html : beach.jpg bridge.jpg
               beach.thumb.jpg bridge.thumb.jpg \
               gallery.mako gallery.py
    python3 gallery.py beach.jpg bridge.jpg > gallery.html

beach.thumb.jpg: beach.jpg
    convert -resize 320x beach.jpg beach.thumb.jpg

bridge.thumb.jpg: bridge.jpg
    convert -resize 320x bridge.jpg bridge.thumb.jpg
```

```
$ make
convert -resize 320x beach.jpg beach.thumb.jpg
convert -resize 320x bridge.jpg bridge.thumb.jpg
python3 gallery.py beach.jpg bridge.jpg > gallery.html
$
```

Two image gallery, output

Wallpaper Gallery

beach.jpg

bridge.jpg

Simplifying Rules

```
beach.thumb.jpg: beach.jpg
    convert -resize 320x beach.jpg beach.thumb.jpg

bridge.thumb.jpg: bridge.jpg
    convert -resize 320x bridge.jpg bridge.thumb.jpg
```

Simplifying Rules

```
beach.thumb.jpg: beach.jpg
    convert -resize 320x beach.jpg beach.thumb.jpg

bridge.thumb.jpg: bridge.jpg
    convert -resize 320x bridge.jpg bridge.thumb.jpg
```

make exposes useful **automatic variables** to the commands of rules:

- $@ - the target
- $< - the first dependency
- $^ - all of the dependencies

Simplifying Rules

```
beach.thumb.jpg: beach.jpg
    convert -resize 320x beach.jpg beach.thumb.jpg

bridge.thumb.jpg: bridge.jpg
    convert -resize 320x bridge.jpg bridge.thumb.jpg
```

make exposes useful **automatic variables** to the commands of rules:

- $@ - the target
- $< - the first dependency
- $^ - all of the dependencies

We can avoid repetition by using these instead of explicit names:

```
beach.thumb.jpg: beach.jpg
    convert -resize 320x $< $@

bridge.thumb.jpg: bridge.jpg
    convert -resize 320x $< $@
```

Two image gallery, Makefile, rewritten

```
gallery.html: beach.jpg bridge.jpg \
              beach.thumb.jpg bridge.thumb.jpg \
              gallery.mako gallery.py
    python3 gallery.py beach.jpg bridge.jpg > $@

beach.thumb.jpg: beach.jpg
    convert -resize 320x $< $@

bridge.thumb.jpg: bridge.jpg
    convert -resize 320x $< $@
```

Rule Sprawl

```
beach.thumb.jpg: beach.jpg
    convert -resize 320x $< $@

bridge.thumb.jpg: bridge.jpg
    convert -resize 320x $< $@
```

At this point, we have fairly generic rule bodies, but a lot of duplication for each target.

Pattern Rules

Fortunately, `make` supports a simple form of pattern matching with **pattern rules**:

- % is the pattern match placeholder
- % matches one or more character
- % can be used in the target and the dependencies

Pattern Rules

Fortunately, `make` supports a simple form of pattern matching with **pattern rules**:

- % is the pattern match placeholder
- % matches one or more character
- % can be used in the target and the dependencies

We can rewrite our rules for thumbnail generation to be completely generic:

```
%.thumb.jpg: %.jpg
    convert -resize 320x $< $@
```

Two image gallery, Makefile, rewritten

```
gallery.html: beach.jpg bridge.jpg \
              beach.thumb.jpg bridge.thumb.jpg \
              gallery.mako gallery.py
    python3 gallery.py beach.jpg bridge.jpg > $@

%.thumb.jpg: %.jpg
    convert -resize 320x $< $@
```

```
$ make
convert -resize 320x beach.jpg beach.thumb.jpg
convert -resize 320x bridge.jpg bridge.thumb.jpg
python3 gallery.py beach.jpg bridge.jpg > gallery.html
$
```

Incremental Builds

By the way, what happens if we change a deeply buried dependency?

```
$ cp ~/better-beach.jpg beach.jpg
```

Incremental Builds

By the way, what happens if we change a deeply buried dependency?

```
$ cp ~/better-beach.jpg beach.jpg
$ make
convert -resize 320x beach.jpg beach.thumb.jpg
python3 gallery.py beach.jpg bridge.jpg > gallery.html
$
```

`make` only rebuilds the targets that were dependent on `beach.jpg`, and not the targets associated with `bridge.jpg`.

How does it know?

Incremental Builds

`make` will re-execute a rule when a dependency's **modified time** is newer than that of the target.

We can check the **mtime** of files with the `stat` command:

```
$ stat bridge.jpg
  File: bridge.jpg
  Size: 1594290       Blocks: 3120       IO Block: 4096   regular file
  ...
Modify: 2018-05-24 01:29:19.459575114 -0500
  ...
  File: bridge.thumb.jpg
  Size: 14485         Blocks: 32         IO Block: 4096   regular file
  ...
Modify: 2018-05-24 04:45:10.815519021 -0500
  ...
$
```

Incremental Builds

We can fool `make` into rebuilding targets associated with `bridge.jpg` by updated its mtime with `touch`:

```
$ make
make: 'gallery.html' is up to date.
$
```

Incremental Builds

We can fool `make` into rebuilding targets associated with `bridge.jpg` by updated its mtime with `touch`:

```
$ make
make: 'gallery.html' is up to date.
$ touch -m bridge.jpg
$
```

Incremental Builds

We can fool `make` into rebuilding targets associated with `bridge.jpg` by updated its mtime with `touch`:

```
$ make
make: 'gallery.html' is up to date.
$ touch -m bridge.jpg
$ stat bridge.jpg
  File: bridge.jpg
  Size: 1594290       Blocks: 3120       IO Block: 4096   regular file
      ...
Modify: 2018-05-24 05:06:33.815877712 -0500
 Birth: -
  File: bridge.thumb.jpg
  Size: 14485         Blocks: 32         IO Block: 4096   regular file
      ...
Modify: 2018-05-24 04:45:10.815519021 -0500
    ...
$
```

Incremental Builds

We can fool `make` into rebuilding targets associated with `bridge.jpg` by updated its mtime with `touch`:

```
$ make
make: 'gallery.html' is up to date.
$ touch -m bridge.jpg
$ stat bridge.jpg
  File: bridge.jpg
  Size: 1594290       Blocks: 3120       IO Block: 4096   regular file
      ...
Modify: 2018-05-24 05:06:33.815877712 -0500
 Birth: -
  File: bridge.thumb.jpg
  Size: 14485         Blocks: 32         IO Block: 4096   regular file
      ...
Modify: 2018-05-24 04:45:10.815519021 -0500
    ...
$ make
convert -resize 320x bridge.jpg bridge.thumb.jpg
python3 gallery.py beach.jpg bridge.jpg > gallery.html
$
```

Using Variables

`make` supports simple string variables.

- Can be scalars or space-delimited lists
- Defined with `FOO = text`
- Expanded with `$(FOO)`

Using Variables

`make` supports simple string variables.

- Can be scalars or space-delimited lists
- Defined with `FOO = text`
- Expanded with `$(FOO)`

We can define variables for our inputs and parameters:

```
IMAGE_FILES = beach.jpg bridge.jpg
THUMB_FILES = beach.thumb.jpg bridge.thumb.jpg
THUMB_WIDTH = 320
OUTPUT = gallery.html
```

and expand them with `$(IMAGE_FILES)`, `$(THUMB_FILES)`, `$(THUMB_WIDTH)`, `$(OUTPUT)`.

Two image gallery, Makefile, rewritten

```
IMAGE_FILES = beach.jpg bridge.jpg
THUMB_FILES = beach.thumb.jpg bridge.thumb.jpg
THUMB_WIDTH = 320
OUTPUT = gallery.html

$(OUTPUT): $(IMAGE_FILES) $(THUMB_FILES) \
           gallery.mako gallery.py
    python3 gallery.py $(IMAGE_FILES) > $@

%.thumb.jpg: %.jpg
    convert -resize $(THUMB_WIDTH)x $< $@
```

Clean up on aisle $PWD

Our working directory has become quite a mess:

```
$ ls
beach.jpg          bridge.thumb.jpg  gallery.mako  mountain.jpg
beach.thumb.jpg    flower.jpg        gallery.py    road.jpg
bridge.jpg         gallery.html      Makefile      tree.jpg
$
```

Clean up on aisle $PWD

Our working directory has become quite a mess:

```
$ ls
beach.jpg          bridge.thumb.jpg  gallery.mako  mountain.jpg
beach.thumb.jpg    flower.jpg        gallery.py    road.jpg
bridge.jpg         gallery.html      Makefile      tree.jpg
$
```

It's a common convention to include abstract targets, called **goals**, like `all`, `clean`, or `install`, to accomplish certain tasks.

```
clean:
    rm $(THUMB_FILES) $(OUTPUT)
```

```
$ make clean
rm beach.thumb.jpg bridge.thumb.jpg gallery.html
$
```

Two image gallery, Makefile, rewritten

```
IMAGE_FILES = beach.jpg bridge.jpg
THUMB_FILES = beach.thumb.jpg bridge.thumb.jpg
THUMB_WIDTH = 320
OUTPUT = gallery.html

$(OUTPUT): $(IMAGE_FILES) $(THUMB_FILES) \
           gallery.mako gallery.py
    python3 gallery.py $(IMAGE_FILES) > $@

clean:
    rm $(THUMB_FILES) $(OUTPUT)

%.thumb.jpg: %.jpg
    convert -resize $(THUMB_WIDTH)x $< $@
```

Gotcha: Exit Codes

However, one thing to keep in mind is that `make` uses **exit codes** to detect failure when executing a rule.

```
$ make clean
rm beach.thumb.jpg bridge.thumb.jpg gallery.html
$ make clean
rm beach.thumb.jpg bridge.thumb.jpg gallery.html
rm: cannot remove 'beach.thumb.jpg': No such file or directory
rm: cannot remove 'bridge.thumb.jpg': No such file or directory
rm: cannot remove 'gallery.html': No such file or directory
make: *** [Makefile.6:11: clean] Error 1
$
```

Gotcha: Exit Codes

However, one thing to keep in mind is that `make` uses **exit codes** to detect failure when executing a rule.

```
$ make clean
rm beach.thumb.jpg bridge.thumb.jpg gallery.html
$ make clean
rm beach.thumb.jpg bridge.thumb.jpg gallery.html
rm: cannot remove 'beach.thumb.jpg': No such file or directory
rm: cannot remove 'bridge.thumb.jpg': No such file or directory
rm: cannot remove 'gallery.html': No such file or directory
make: *** [Makefile.6:11: clean] Error 1
$
```

We can ask `make` to ignore the failure of a command by prefixing it with a dash `-` character, so it will continue executing the next command or target.

In this case, it's common to use `rm -f` to surpress errors altogether.

Two image gallery, Makefile, rewritten

```
IMAGE_FILES = beach.jpg bridge.jpg
THUMB_FILES = beach.thumb.jpg bridge.thumb.jpg
THUMB_WIDTH = 320
OUTPUT = gallery.html

$(OUTPUT): $(IMAGE_FILES) $(THUMB_FILES) \
           gallery.mako gallery.py
    python3 gallery.py $(IMAGE_FILES) > $@

clean:
    rm -f $(THUMB_FILES) $(OUTPUT)

%.thumb.jpg: %.jpg
    convert -resize $(THUMB_WIDTH)x $< $@
```

Those .PHONY targets

What happened to everything is a file? Don't worry, everything still is.

```
$ echo "hello world" > clean
$ cat clean
hello world
$ make clean
make: Nothing to be done for 'clean'.
$
```

Those .PHONY targets

What happened to everything is a file? Don't worry, everything still is.

```
$ echo "hello world" > clean
$ cat clean
hello world
$ make clean
make: Nothing to be done for 'clean'.
$
```

In order for rules with goals or other file-less targets to always execute, we need to mark them .PHONY:

```
.PHONY: clean
clean:
    rm $(THUMB_FILES) $(OUTPUT)
```

```
$ cat clean
hello world
$ make clean
rm beach.thumb.jpg bridge.thumb.jpg gallery.html
$
```

Two image gallery, Makefile, rewritten

```
IMAGE_FILES = beach.jpg bridge.jpg
THUMB_FILES = beach.thumb.jpg bridge.thumb.jpg
THUMB_WIDTH = 320
OUTPUT = gallery.html

$(OUTPUT): $(IMAGE_FILES) $(THUMB_FILES) \
           gallery.mako gallery.py
    python3 gallery.py $(IMAGE_FILES) > $@

.PHONY: clean
clean:
    rm -f $(THUMB_FILES) $(OUTPUT)

%.thumb.jpg: %.jpg
    convert -resize $(THUMB_WIDTH)x $< $@
```

Using $(patsubst ...)

Our $(THUMB_FILES) variable is essentially a permutation of the input $(IMAGE_FILES) variable.

```
IMAGE_FILES = beach.jpg bridge.jpg
THUMB_FILES = beach.thumb.jpg bridge.thumb.jpg
```

Using $(patsubst ...)

Our $(THUMB_FILES) variable is essentially a permutation of the input $(IMAGE_FILES) variable.

```
IMAGE_FILES = beach.jpg bridge.jpg
THUMB_FILES = beach.thumb.jpg bridge.thumb.jpg
```

We can use the powerful $(patsubst *pattern*,*replacement*,*text*) function to generate it.

```
IMAGE_FILES = beach.jpg bridge.jpg
THUMB_FILES = $(patsubst %.jpg,%.thumb.jpg,$(IMAGE_FILES))
```

The % is used as a wildcard in the pattern and replacement, much like in pattern rules.

Two image gallery, Makefile, rewritten

```
IMAGE_FILES = beach.jpg bridge.jpg
THUMB_WIDTH = 320
OUTPUT = gallery.html

THUMB_FILES = $(patsubst %.jpg,%.thumb.jpg,$(IMAGE_FILES))

$(OUTPUT): $(IMAGE_FILES) $(THUMB_FILES) \
           gallery.mako gallery.py
    python3 gallery.py $(IMAGE_FILES) > $@

.PHONY: clean
clean:
    rm -f $(THUMB_FILES) $(OUTPUT)

%.thumb.jpg: %.jpg
    convert -resize $(THUMB_WIDTH)x $< $@
```

Adding a `dist/` folder

Often, we want to consolidate build products to a folder, e.g. website deployment.

We can model the `dist` folder like any other target or dependency:

```
OUTPUT_DIR = dist

$(OUTPUT_DIR):
    mkdir $@
```

And create an additional rule to copy the images to the `dist` folder:

```
$(OUTPUT_DIR)/%.jpg: %.jpg $(OUTPUT_DIR)
    cp $< $@
```

Two image gallery, Makefile, rewritten 1/2

```
IMAGES = beach.jpg bridge.jpg
THUMB_WIDTH = 320
WEBPAGE = gallery.html
OUTPUT_DIR = dist

IMAGE_FILES = $(patsubst %.jpg,$(OUTPUT_DIR)/%.jpg,$(IMAGES))
THUMB_FILES = $(patsubst %.jpg,$(OUTPUT_DIR)/%.thumb.jpg,$(IMAGES)

$(OUTPUT_DIR)/$(WEBPAGE): $(IMAGE_FILES) $(THUMB_FILES) \
                          gallery.mako gallery.py
    python3 gallery.py $(IMAGES) > $@
```

Two image gallery, Makefile, rewritten 2/2

```
.PHONY: clean
clean:
    rm -rf $(OUTPUT_DIR)

$(OUTPUT_DIR)/%.jpg: %.jpg $(OUTPUT_DIR)
    cp $< $@

$(OUTPUT_DIR)/%.thumb.jpg: %.jpg $(OUTPUT_DIR)
    convert -resize $(THUMB_WIDTH)x $< $@

$(OUTPUT_DIR):
    mkdir $(OUTPUT_DIR)
```

A slight problem...

Everything seems to work on the surface. But, if we run `make` twice:

```
$ make
mkdir dist
cp beach.jpg dist/beach.jpg
cp bridge.jpg dist/bridge.jpg
convert -resize 320x beach.jpg dist/beach.thumb.jpg
convert -resize 320x bridge.jpg dist/bridge.thumb.jpg
python3 gallery.py beach.jpg bridge.jpg > dist/gallery.html
$ make
cp beach.jpg dist/beach.jpg
cp bridge.jpg dist/bridge.jpg
convert -resize 320x beach.jpg dist/beach.thumb.jpg
convert -resize 320x bridge.jpg dist/bridge.thumb.jpg
python3 gallery.py beach.jpg bridge.jpg > dist/gallery.html
$
```

`make` is always rebuilding all the targets that have the folder as a dependency.

A slight problem...

Everything seems to work on the surface. But, if we run `make` twice:

```
$ make
mkdir dist
cp beach.jpg dist/beach.jpg
cp bridge.jpg dist/bridge.jpg
convert -resize 320x beach.jpg dist/beach.thumb.jpg
convert -resize 320x bridge.jpg dist/bridge.thumb.jpg
python3 gallery.py beach.jpg bridge.jpg > dist/gallery.html
$ make
cp beach.jpg dist/beach.jpg
cp bridge.jpg dist/bridge.jpg
convert -resize 320x beach.jpg dist/beach.thumb.jpg
convert -resize 320x bridge.jpg dist/bridge.thumb.jpg
python3 gallery.py beach.jpg bridge.jpg > dist/gallery.html
$
```

`make` is always rebuilding all the targets that have the folder as a dependency.

This is because the `dist` folder itself has a modified time, which will be updated any time a file within it changes.

"Order-only" Dependencies

'make' has a special syntax to separate dependencies that should not be time tracked, called "order-only" dependencies:

Anatomy of a Makefile Rule

```
target: dependencies... | order-only dependencies...
    commands
    ...
```

We can annotate the rules that depend on the `dist` folder with the pipe `|` symbol separator.

Two image gallery, Makefile, rewritten 1/2

```
IMAGES = beach.jpg bridge.jpg
THUMB_WIDTH = 320
WEBPAGE = gallery.html
OUTPUT_DIR = dist

IMAGE_FILES = $(patsubst %.jpg,$(OUTPUT_DIR)/%.jpg,$(IMAGES))
THUMB_FILES = $(patsubst %.jpg,$(OUTPUT_DIR)/%.thumb.jpg,$(IMAGES)

$(OUTPUT_DIR)/$(WEBPAGE): $(IMAGE_FILES) $(THUMB_FILES) \
                          gallery.mako gallery.py
    python3 gallery.py $(IMAGES) > $@
```

Two image gallery, Makefile, rewritten 2/2

```
.PHONY: clean
clean:
    rm -rf $(OUTPUT_DIR)

$(OUTPUT_DIR)/%.jpg: %.jpg | $(OUTPUT_DIR)
    cp $< $@

$(OUTPUT_DIR)/%.thumb.jpg: %.jpg | $(OUTPUT_DIR)
    convert -resize $(THUMB_WIDTH)x $< $@

$(OUTPUT_DIR):
    mkdir $(OUTPUT_DIR)
```

Scaling up with `$(wildcard ...)`

With the help of the function `$(wildcard ...)`, we can extend our Makefile to automatically include all images.

```
IMAGES := $(wildcard *.jpg)
```

- `:=` indicates immediate expansion, so the wildcard isn't evaluated multiple times in the Makefile.
- `make` features countless other functions useful for building target lists: `$(shell ...)`, `$(basename ...)`, `$(dir ...)`, etc.

Full image gallery, Makefile, rewritten 1/2

```
IMAGES = $(wildcard *.jpg)
WEBPAGE = gallery.html
THUMB_WIDTH = 320
OUTPUT_DIR = dist

IMAGE_FILES = $(patsubst %.jpg,$(OUTPUT_DIR)/%.jpg,$(IMAGES))
THUMB_FILES = $(patsubst %.jpg,$(OUTPUT_DIR)/%.thumb.jpg,$(IMAGES)

.PHONY: all
all: $(OUTPUT_DIR)/$(WEBPAGE)

.PHONY: clean
clean:
    rm -rf $(OUTPUT_DIR)
```

Full image gallery, Makefile, rewritten 2/2

```
$(OUTPUT_DIR)/$(WEBPAGE): $(IMAGE_FILES) $(THUMB_FILES) \
                          gallery.mako gallery.py
    python3 gallery.py $(IMAGES) > $@

$(OUTPUT_DIR)/%.jpg: %.jpg | $(OUTPUT_DIR)
    cp $< $@

$(OUTPUT_DIR)/%.thumb.jpg: %.jpg | $(OUTPUT_DIR)
    convert -resize $(THUMB_WIDTH)x $< $@

$(OUTPUT_DIR):
    mkdir $(OUTPUT_DIR)
```

When everything has to be a file... create them?

- Sometimes, a build step mutates existing files (like a patch, append, etc.) and doesn't create a new one.

When everything has to be a file... create them?

- Sometimes, a build step mutates existing files (like a patch, append, etc.) and doesn't create a new one.
- `make` can't track these steps, as it only understands a target is met with a file.

When everything has to be a file... create them?

- Sometimes, a build step mutates existing files (like a patch, append, etc.) and doesn't create a new one.
- `make` can't track these steps, as it only understands a target is met with a file.
- Idiomatic to create a dummy a "stamp" file marking the completion of that step.

When everything has to be a file... create them?

- Sometimes, a build step mutates existing files (like a patch, append, etc.) and doesn't create a new one.
- `make` can't track these steps, as it only understands a target is met with a file.
- Idiomatic to create a dummy a "stamp" file marking the completion of that step.

When everything has to be a file... create them?

- Sometimes, a build step mutates existing files (like a patch, append, etc.) and doesn't create a new one.
- `make` can't track these steps, as it only understands a target is met with a file.
- Idiomatic to create a dummy a "stamp" file marking the completion of that step.

```
secret.txt: data.txt
    rm -f $@
    cat $< | tr 'A-Za-z' 'N-ZA-Mn-za-m' > $@

permissions-stamp: secret.txt
    chmod 600 $<
    touch $@
```

Final Makefile 1/2

```
IMAGES = $(wildcard *.jpg)
WEBPAGE = gallery.html
THUMB_WIDTH = 320
OUTPUT_DIR = dist

IMAGE_FILES = $(patsubst %.jpg,$(OUTPUT_DIR)/%.jpg,$(IMAGES))
THUMB_FILES = $(patsubst %.jpg,$(OUTPUT_DIR)/%.thumb.jpg,$(IMAGES)

.PHONY: all
all: $(OUTPUT_DIR)/$(WEBPAGE)

.PHONY: clean
clean:
    rm -rf $(OUTPUT_DIR)
```

Final Makefile 2/2

```
$(OUTPUT_DIR)/$(WEBPAGE): $(IMAGE_FILES) $(THUMB_FILES) \
                          gallery.mako gallery.py
    python3 gallery.py $(IMAGES) > $@

$(OUTPUT_DIR)/%.jpg: %.jpg | $(OUTPUT_DIR)
    cp $< $@

$(OUTPUT_DIR)/%.thumb.jpg: %.jpg | $(OUTPUT_DIR)
    convert -resize $(THUMB_WIDTH)x $< $@

$(OUTPUT_DIR):
    mkdir $(OUTPUT_DIR)
```

Final Build

```
$ make
mkdir dist
cp mountain.jpg dist/mountain.jpg
cp beach.jpg dist/beach.jpg
cp bridge.jpg dist/bridge.jpg
cp flower.jpg dist/flower.jpg
cp road.jpg dist/road.jpg
cp tree.jpg dist/tree.jpg
convert -resize 320x mountain.jpg dist/mountain.thumb.jpg
convert -resize 320x beach.jpg dist/beach.thumb.jpg
convert -resize 320x bridge.jpg dist/bridge.thumb.jpg
convert -resize 320x flower.jpg dist/flower.thumb.jpg
convert -resize 320x road.jpg dist/road.thumb.jpg
convert -resize 320x tree.jpg dist/tree.thumb.jpg
python3 gallery.py beach.jpg bridge.jpg road.jpg
                   tree.jpg mountain.jpg flower.jpg
                                        > dist/gallery.html
$
```

Result

Wallpaper Gallery


beach.jpg


bridge.jpg


road.jpg


tree.jpg


mountain.jpg


flower.jpg

Limitations of Make

- Manual dependency grap

Limitations of Make

- Manual dependency grap
- Everything is a file

Limitations of Make

- Manual dependency grap
- Everything is a file
- Shell commands

Should you use make?

- Maybe

Should you use make?

- Maybe
  - Slightly more complicated than a shell script? Yes

Should you use make?

- Maybe
  - Slightly more complicated than a shell script? Yes
  - Building C/C++? Probably

Should you use make?

- Maybe
  - Slightly more complicated than a shell script? Yes
  - Building C/C++? Probably
  - Some hipster language with a built-in build system? No

Should you use make?

- Maybe
  - Slightly more complicated than a shell script? Yes
  - Building C/C++? Probably
  - Some hipster language with a built-in build system? No
- Inevitably

Should you use make?

- Maybe
  - Slightly more complicated than a shell script? Yes
  - Building C/C++? Probably
  - Some hipster language with a built-in build system? No
- Inevitably
  - You may find yourself automating other build tools...

Questions

Questions?