Build Tools & Program Management

Joshua Ervin

November 19, 2019

CSE391

tar

tar

`tar`: create or extract tar archives.

- To **create** a single file from multiple files:
  `$ tar -cf filename.tar stuff_to_archive`
  - `-c` **creates** an archive
  - `-f` read to/from a file
  - `stuff_to_archive` — can be a list of filenames or a directory
- To **extract** files from an archive:
  `$ tar -xf filename.tar`
  - `-x` **extracts** files from an archive.

Compressed Files

command	description
zip, unzip	create or extract /zip compressed archives
gzip, gunzip	GNU free compression programs (single-file)
bzip2, bunzip2	slower, optimized compression program (single-file)

- To **compress** a file

  `$ gzip filename` produces: `filename.gz`

- To **decompress** a file

  `$ gunzip filename.gz` produces: `filename`

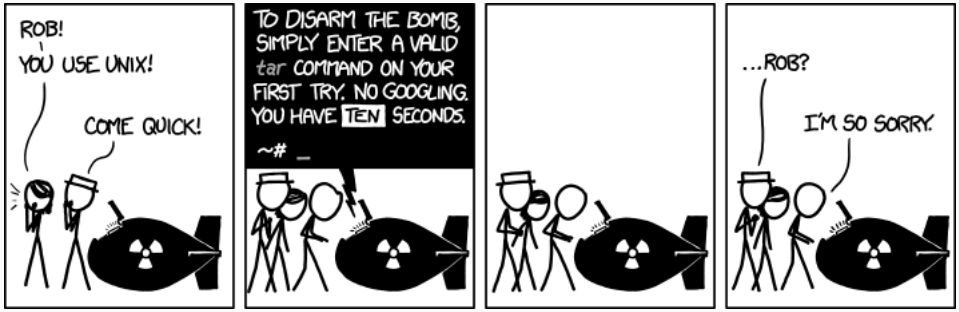
.tar.gz archives

- Many Linux programs are distributed as `.tar.gz` archives
- You could unpack this in two steps:
  1. `gzip foo.tar.gz` — produces: `foo.tar`
  2. `tar -xf foo.tar` — extracts individual files
- You can also use the `tar` command to create/extract compressed archive files all in one step: `tar -xzf filename.tar.gz`
  - `-x` extracts files from an archive
  - `-z` filter the archive through gzip (compress/decompress it)
  - `-f` read to/from a file

tar examples

You can combine options (-v, -z, etc) in various ways:

- Create a single `.tar` archive file from multiple files (without compression)
  `$ tar -cvf filename.tar stuff_to_archive`
  - `-c` creates an archive file called filename.tar
  - `-c` verbosley list the files processed
  - `-f` read to/from a file
  - `stuff_to_archive` can be a list of filenames or a directory
- Note: use the `-z` option and use filename.tar.gz to use compression
  `$tar -cvzf filename.tar.gz stuff_to_archive`


ROB!
YOU USE UNIX!
COME QUICK!
TO DISARM THE BOMB, SIMPLY ENTER A VALID tar COMMAND ON YOUR FIRST TRY. NO GOOGLING. YOU HAVE TEN SECONDS.
~# _
...ROB?
I'M SO SORRY.

The PATH Variable

How is PATH Used?

- When you run a command like `ls`, your system uses the following algorithm to find and execute the program:

```
procedure execute(command)
    for every directory in $PATH
        if directory contains the command
            execute the command
            exit
    end for

    print "command not found"
```

Suppose we have the following PATH and executable files with the following contents

```
PATH=/usr/home:/usr/bin:/usr/sbin
```

/usr/bin/hello.sh

```
echo "howdy partner"
```

/usr/sbin/hello.sh

```
echo "salutations"
```

What would be the output of the following?

```
$ hello.sh
```

Adding to your PATH

We usually add directories to our `PATH` in our `.bashrc` or `.bash_profile`

Prepending a directory to your `PATH`

```
PATH=/your/new/directory:$PATH
```

Appending a directory to your `PATH`

```
PATH=$PATH:/your/new/directory
```

With Great Power comes Great Responsibility!

What happens if we were to run the following command:

```
PATH=''
```

And then tried to open `vim` or `emacs`?

With Great Power comes Great Responsibility!

What happens if we were to run the following command:

```
PATH=''
```

And then tried to open `vim` or `emacs`?

We get the following error!

```
bash: vim: command not found
```

How Do We Fix This?

Instead of typing just individual commands, give the full path to an editor or command that you can use to fix your PATH

```
$ /usr/bin/vim /homes/iws/joshue/.bashrc
```

Package Managers

Installing Programs: Package Manager

Most UNIX-like distributions come with a package manger — a tool to install, update, and remove packages (i.e. applications) from the command line.

Installing `firefox` on windows:

1. Google search “firefox download”
2. Visit mozilla.org
3. Click on “Download Now”
4. Run the `firefox` installer

Installing `firefox` on Linux (centOS):

1. type `sudo yum install firefox` on the command line

What Exactly is a Package Manager?

Packages, or applications, are stored on a central repository managed by the organization who builds a Linux distribution.

Most package managers require a vetting process before an application can be added to the repository — this prevents malware and unstable applications from being added.

Your local package manager (`yum` on attu), pulls binaries from this repository and adds them to your PATH.

Language-Specific Package Managers

Many modern programming languages come with their own package managers to install, upgrade, and remove dependencies. Some examples include:

Programming Language	Package Manager
python	pip
rust	cargo
ruby	bundle
javascript	npm
haskell	cabal

make

Make: Motivation

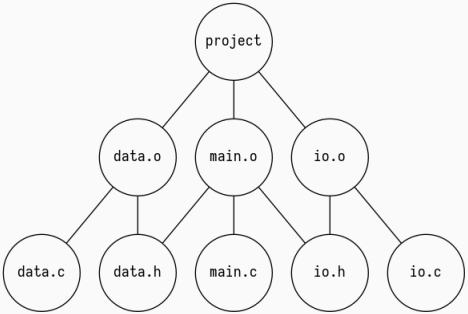
- Single-file programs do not work well when code-bases get large
  - Compilation can be slow
  - Difficult to collaborate with other developers
  - Cumbersome
- Large program are split into multiple files
  - Each file represents a partial program or *module*
  - Modules can be compiled separately or together.
  - A module can be shared between multiple programs

make

- `make`: A utility for automatically compiling (building) executables and libraries from source code.
  - A very basic compilation manager
  - Often used for `C` programs, but not language-specific
  - Primitive, but still widely used due to familiarity and simplicity
  - Similar programs: ant, maven, gradle
- `Makefile`: A script file that defines rules for what must be compiled and how to compile it.
  - Makefiles describe which files depend on which others, and how to create/compile/build/update each file in the system as needed.

Dependencies

- When a file relies on the contents of another
  - Can be displayed as a dependency graph
  - To build `main.o`, we need `data.h`, `main.c`, and `io.h`
  - If any of those files are updated, we must rebuild `main.o`
  - If `main.o` is updated, we must updated project



Makefile rule syntax

```
target: source1 source2 … sourceN
        command
        command
```

- `source1` through `sourceN` are *dependencies* for building `target`
- A `source` is a file that is used as input to create the `target` (Sources are sometimes called prerequisites)
- A `target` often depends on several `sources`
- `make` will execute the `commands` in the order they are listed.

NOTE: Makefiles must be indented using **tabs**. USING SPACES WILL NOT WORK!!!!

Using make

```
$ make target
```

- Uses the file named `Makefile` in the current directory
- Finds a `rule` in `Makefile` for building `target` and follows it
  - If the `target` file does not exist, or if it is older than any of its sources, its commands will be executed.

```
$ make
```

- builds the *first* target in the `Makefile` by default.

```
$ make -f makefilename
```

```
$ make -f makefilename target
```

- uses a makefile other than `Makefile`

Building a Simple Makefile

```
dress: pants shoes jacket
    @echo "All done. Let's go outside!"

jacket: shirt
    @echo "Putting on jacket"

shirt:
    @echo "Putting on shirt"

pants: underpants
    @echo putting on pants

... See attached files for full Makefile
```

Poll Everywhere!

Suppose we have the following Makefile. What files would be changed running `$ make`

Makefile

```
aprogram: foo.o bar.o
    gcc -o aprogram foo.o bar.o

foo.o: foo.c
    gcc -c foo.c

bar.o: bar.c
    gcc -c bar.c
```

And `ls -l` produces the following:

```
aprogram: Nov 19 12:17
bar.c     Nov 19 12:17
bar.o     Nov 19 12:17
foo.c     Nov 19 12:34
foo.o     Nov 19 12:34
```

make: Variables

```
NAME = value      (declaring a variable)
$(NAME)           (using a variable)
```

Example `Makefile`

Makefile

```
OBJFILES = file1.o file2.o file3.o
PROGRAM = myprog

$(PROGRAM) : $(OBJFILES)
    gcc -o $(PROGRAM) $(OBJFILES)

clean:
    rm $(OBJFILES) $(PROGRAM)
```

```
$@    the current target file
$^    all sources listed for the current target
$<    the first (left-most) source for the current target
```

Example `Makefile`

Makefile

```
myprog: file1.o file2.o file3.o
    gcc $(CFLAGS) -o $@ $^

file1.o: file1.c file1.h file2.h
    gcc $(CFLAGS) -c $<
```

- Rather than specifying individually how to convert every `.c` file into its corresponding `.o` file, we can make use of the following pattern rules:

Makefile

```
CC = gcc
CLAGS = -Wall

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

- In English, this means *To create filename.o from filename.c, run gcc -c -Wall filename.c -o filename.o*