Introduction to Make

Daniel Lucio

Overview

- What is it?
- How it works?
- Special Characters
- Rules
- Macros
- Examples
- Limitations
- Building a program
- More information

What is it?

NAME

make - GNU make utility to maintain groups of programs

SYNOPSIS

make [-f makefile] [options] ... [targets] ...

DESCRIPTION

The purpose of the make utility is to determine automatically which pieces of a large program need to be recompiled, and issue the commands to recompile them. You can use it to describe any task where some files must be updated automatically from others whenever the others change.

How it works?

To prepare to use make, you must write a file called the `makefile` that describes the relationships among files in your program, and the states the commands for updating each file. In a program, typically the executable file is updated from object files, which are in turn made by compiling source files.

The description file, usually named Makefile commonly resides in the working directory for the project. The Makefile specifies a hierarchy of dependencies among individual files called components. At the top of the hierarchy there is a target.

How it works?

The `make` program uses the makefile data base and the last-modification times of the files to decide which of the files need to be updated. For each of those files, it issues the commands recorded in the data base.

'`make`' executes commands in the makefile to update one or more target names, where name is typically a program. If no `-f` option is present, make will look for the makefiles `GNUmakefile`, `makefile`, and `Makefile`, in that order.

Special Characters

#	Starts a comment. Comments are ignored.
/	The backslash is used to continue a line.
TAB	The character before a ‘command’ is a tab.

Rules

- A makefile consists of rules. Each rule begins with a textual dependency line which defines a target followed by a colon (:) and optionally an enumeration of components (files or other targets) on which the target depends.
- The dependency line is arranged so that the target (left hand of the colon) depends on components (right hand of the colon).
- It is common to refer to components as prerequisites of the target.

Rules

```
target [target ...]: [component ...]
<TAB>command 1]
                .
                .
                .
[<TAB>command n]
```

Each command line must begin with a TAB character to be recognized as a command.

Macros

- A makefile can contain definitions of macros.
- Macros are usually referred to as variables when they hold simple string definitions, like `"CC=gcc"`.
- Macros in makefiles may be overridden in the command-line arguments passed to the **Make** utility.
- Environment variables are also available as macros.
- A macro is used by expanding it. Traditionally this is done by enclosing its name inside $( ). An equivalent form uses curly braces rather than parenthesis, i.e. ${ }.

Macros

```
MACRO = definition

NEW_MACRO = $(MACRO)-$(MACRO2)

YYYYMMDD  = ` date `
```

Macros can be composed of shell commands by using the command substitution operator, denoted by backticks (`).

The generic syntax for overriding macros on the command line is:

```
make MACRO="value" [MACRO="value" ...] TARGET [TARGET ...]
```

Makefile example

```
CC      = gcc
CFLAGS = -g

all: helloworld

helloworld: helloworld.o
        # Commands start with TAB not spaces
        $(CC) $(CFLAGS) -o $@ $^

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

clean:
        rm -f helloworld helloworld.o
```

← Macros

← Target

← Rule

← Rule

← Rule

Makefile example

```
# ptoc: print a table of contents

ptoc: ptoc_main.o prn headings.o \
      get_head_info.o check_head.o

        cc -o ptoc ptoc_main.o \
                   prn_headings.o \
                   get+head_info.o \
                   check_head.o

ptoc_main.o prn_headings.o \
get_head_info.o  check_head.o: ptoc.h
```

← Comment

← Target

← Prerequisites

← Action

← Targets

← Prerequisite

Makefile example

Target

Prerequisites

```
manual: ch01.fmt ch02.fmt ch03.fmt
        lp ch0[1-3].fmt

ch01.fmt: ch01
        nroff -mm ch01 > ch01.fmt

ch02.fmt: ch02
        tbl ch02 | nroff -mm > ch02.fmt

ch03.fmt: ch03a ch03b ch03c
        nroff -mm ch03[abc] > ch03.fmt
```

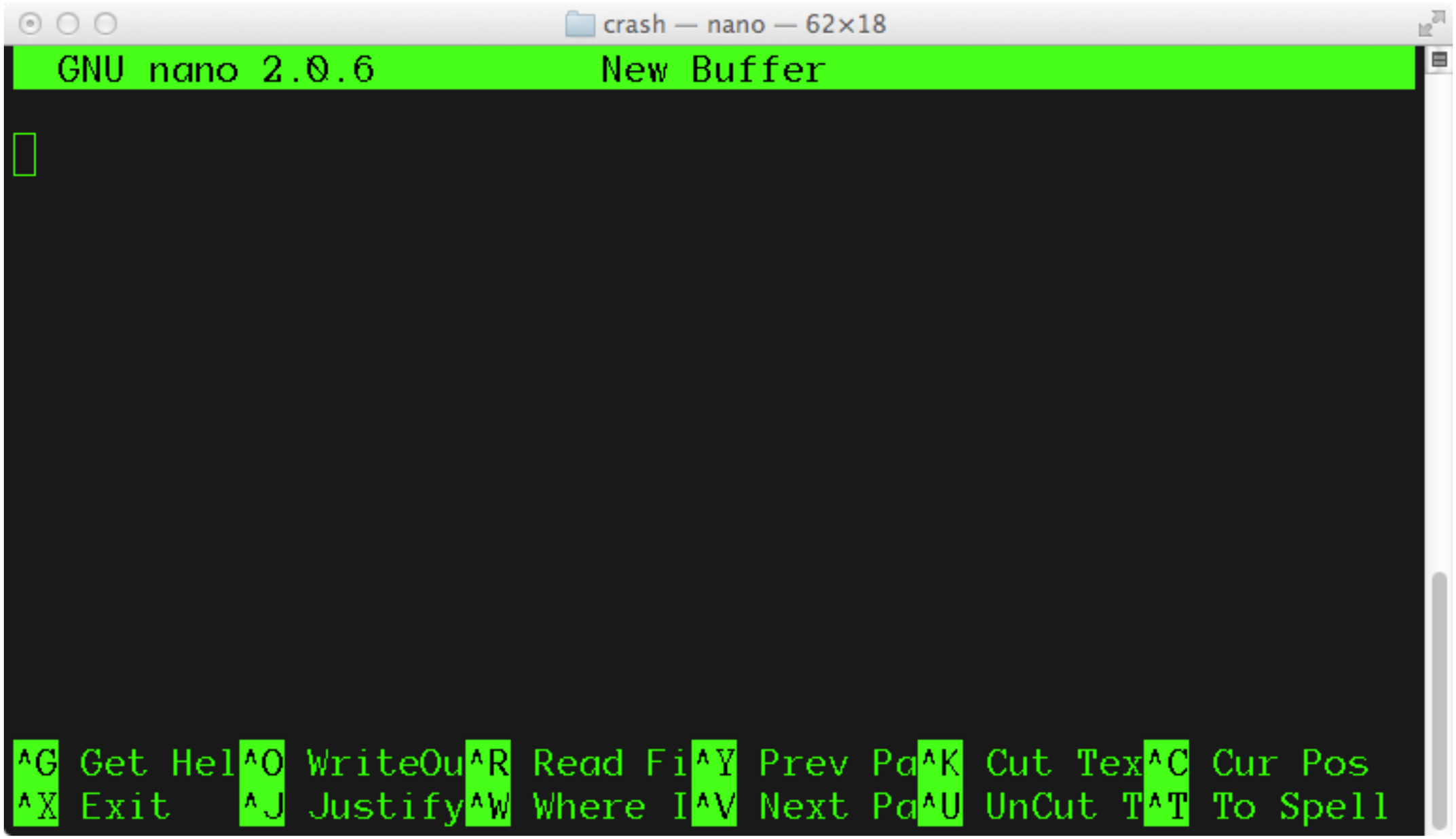
Action

Building a program

Example of how to build a typical program on Unix/Linux.

```
$ wget http://www.nano-editor.org/dist/v2.2/nano-2.2.6.tar.gz
$ tar -zxf nano-2.2.6.tar.gz
$ cd nano-2.2.6
$ ./configure --prefix=$HOME
$ make
$ make install
$ export PATH=$PATH:~/bin
$ nano
```

The ‘nano’ Text Editor



Limitations

- Tailoring build processes to a given platform are not well handled by **Make**. *For instance, the compiler used on one platform might not accept the same options as the one used on another.*
- There are other tools like **Autoconf** and **CMake** that do generate platform specific build instructions, which in turn are processed by **Make**.

More information

http://www.gnu.org/software/make/

```
$ man make
```