Programming with Unix

Raymond Klefstad, Ph.D.

gcc - The GNU Compiler Collection <http://en.wikipedia.org/wiki/GNU_Compiler_Collection>

Make - a program for re-building modular systems

gdb - a symbolic debugger

script - capture an interactive session with Unix

Other tools

gcc - the GNU C compiler

. gcc translates C source code into executable programs

. has (at least) three passes:

. preprocessor

handles #include, #define, #ifdef

. compiler

translates C to machine code object module (.o file)

. linker

joins .o files and libraries into an executable program

resolves references to named entities, e.g., functions, methods, static variables

Ways to use gcc

% **gcc** foo.c

. generates an executable **a.out**

% gcc foo.c **-o myprog**

. generates an executable myprog, (only if no errors)

% gcc **-c** foo.c

. generates link module foo.o

% gcc foo**.o** bar**.o** -o myprog

links object modules foo.o and bar.o **with stdlib** then generates myprog

other flags to gcc

% gcc **-ggdb** foo.c

% gcc -g foo.c

. inserts symbol table information for use by gdb (GNU debugger)

% **strip** a.out

. removes symbol table information from a.out

% gcc **-pg** foo.c

. inserts profiling hooks for gprof (GNU profiler)

The GNU make Facility (<http://www.gnu.org/software/make/manual/html_node/index.html>)

. make is a tool for building systems composed of modules

. eg programs, papers, lectures

. `Makefile` contains dependency relations and commands to update

. need for update is determined by comparing time of last modification

. Makefile contains macros, dependency rules, and update commands

Dependency Rules and Update Commands

. form

targets: components ; command

command

…

. executing the command must update the target (or make gets confused)

. can use **‘touch’** to update the modification time

. Simple Example: a single file

date: date.c ; gcc date.c -o date

. Basic Example: a program composed of two modules: date.[ch] main.c

date: date.o main.o

gcc date.o main.o -o date

date.o: date.c date.h

gcc -c date.c

main.o: main.c date.h

gcc -c main.c

. More Typical Example: a program composed of multiple (4) modules: [ABC].[ch] main.c

main: main.o A.o B.o C.o

gcc main.o A.o B.o C.o -o main

A.o: A.c A.h B.h

gcc -c A.c

B.o: B.c B.h C.h

gcc -c B.c

C.o: C.c C.h A.h

gcc -c C.c

main.o: main.c A.h B.h C.h

gcc -c main.c

Updating Target Files

. make builds a dependency graph

. it does depth-first-search on the graph to determine what must be 'remade'

. if a target is older than any of its components, the update commands

are issued

. commands are printed as they are executed

@ before command prevents printing

Macros

. definition

VAR=value

. value is referenced with $(VAR) or ${VAR} or $V

. eg

OBJECTS = a.o b.o \

c.o d.o e.o

SOURCES = a.c b.c c.c

HEADERS = a.h b.h c.h

ALLC = $(SOURCES) $(HEADERS)

CCFLAGS = -ggdb

CC = gcc

Example

. More Typical Example: a program composed of multiple (4) modules: [ABC].[ch] main.c

OBJECTS = main.o A.o B.o C.o

main: $(OBJECTS)

gcc $(OBJECTS) -o main

A.o: A.c A.h B.h

gcc -c A.c

B.o: B.c B.h C.h

gcc -c B.c

C.o: C.c C.h A.h

gcc -c C.c

main.o: main.c A.h B.h C.h

gcc -c main.c

test: main

main < test\_input > test\_output

clean:

/bin/rm -f $(OBJECTS)

Make commands

. % make

. remakes the first target defined in the Makefile

. can add a ‘default’ as the first target and list several components

. % make main

. remakes target main

. % make clean

. removes all the intermediate files that can be reconstructed

. % make -k

. continue to make even after an error with an update command

. % make -n

. just print (don't execute) the update commands

ADVANCED make

Suffix Dependency and Implicit Rules

. you can specify general rules for updating components

. form

.ComponentSufix.TargetSuffix: ; command

. special variables

. $@ - full name of the target to be made

. $\* - common base name of component and target

. $< - full component name for implicit rule

. some of make's default definitions

.SUFFIXES: .out .o .c .h # already done, just showing what some are

CC=gcc # this is the default, so you can omit this

CFLAGS= -ggdb # we are changing the CFLAGS, so we must set it here

O = main.o a.o b.o c.o

.c.o:

$(CC) -c $(CFLAGS) $<

main: $O

main.o: A.h B.h C.h

A.o: A.h B.h

B.o: B.h C.h

C.o: A.h C.h

XXXDDebugging C programs with gdb (<http://sourceware.org/gdb/onlinedocs/gdb/>)

. gdb is a source-level debugger

. provides the following features

. source-level control tracing and single-stepping

. setting breakpoints and evaluating expressions at breakpoints

. displaying and editing of source files

. to use gdb, code must be compiled with -g or -ggdb option

. % gcc -ggdb my\_prog.c -o my\_prog

. run gdb supplying the name of your program

. % gdb my\_prog

. run your program

. (gdb) run args < input\_file > output\_file

. the program will stop upon breakpoint or interrupt (^C) or other signal (SEG FAULT)

. it will show you the exact instruction that caused the SEG FAULT - VERY USEFUL!

. Summary of gdb commands follows

Dynamic Tracing

. (most arguments can be line number or function name)

. trace line\_number

. show whenever the specified line is executed

. trace function\_name

. show each call to named function

. trace variable

. print new value whenever variable is set

. trace expression at line

. print value of expression when executing line

. trace if condition

. trace only if condition is true

. delete trace number

. removes tracepoint number (del tr)

Listing Source Code

. list

. show the next 10 lines of source code

. list start,end

. show lines from number start to number end

. list function

. show the source code for the named function

. whatis name

. displays the declared type structure of identifier name

Setting Breakpoints

. break line

. suspend execution when it reaches line

. break line if condition

. suspend execution at line when it reaches that point and condition is true

. print expression

. display the value of expression

. c

. continue execution from a breakpoint

. bt or backtrace

. displays the sequence of function calls on the stack

. VERY HELPFUL!!!!

Single Stepping

. step

. executes one source line

. perhaps into a function call

. next

. execute up to the next source line

. even across function calls

. quit

. terminate the program and leave the debugger

Script

. captures an interactive session in a file

. warning: do not run editors (like vim or emacs in scripts - ok go ahead and try it!)

. e.g.,

**openlab: $ script hw1.log**

Script started, file is hw1.log

**openlab: $ ls**

a.c b.c #foobar# **hw1.log** MyString.h whatsUp\*

a.h b.h forifscr\* ListNode.h outOfX\* whilescr\*

a.out\* casescr\* forscr\* main.c paramscr\*

args.c FinalQuizTest\* getpage\* Makefile scr\*

**openlab: $ exit**

exit

Script done, file is hw1.log

**openlab: $ cat hw1.log**

Script started on Tue 09 Apr 2013 01:55:47 PM PDT

**openlab**: $ ls

a.c b.c #foobar# hw1.log MyString.h whatsUp\*

a.h b.h forifscr\* ListNode.h outOfX\* whilescr\*

a.out\* casescr\* forscr\* main.c paramscr\*

args.c FinalQuizTest\* getpage\* Makefile scr\*

**openlab**: $ exit

exit

Script done on Tue 09 Apr 2013 01:55:52 PM PDT

**openlab: $**

Other tools

. gprof

profiles your program

must compile with -gp option

<http://sourceware.org/binutils/docs/gprof/>

. valgrind

profiles and helps locate memory leaks

<http://www.cprogramming.com/debugging/valgrind.html>