Reviewing gcc, make, gdb, and Linux Editors[1]

Colin Gordon
csgordon@cs.washington.edu

University of Washington

CSE333 Section 1, 3/31/11

[1] Lots of material borrowed from 351/303 slides

Today's Topics

- gcc: compilation, linking
- make: simple makefiles
- gdb: breakpoints, inspecting state
- Linux Editors: brief overview

Ask questions any time!

Basic GCC

The simplest way to use `gcc` is:

```
gcc -o program file1.c file2.c ...
```

- This creates the executable `program` in the current directory.
- Omitting the `-o` option generates `a.out`.

You'll probably also want debugging information and to be warned about dangerous things you might have done:

```
gcc -g -Wall -o program file1.c file2.c ...
```

More Flexible GCC

Sometimes we only want to build part of the program:

```
gcc -g -Wall -c module1.c
```

generates `module1.o` (an *object file*). An object file is compiled code with *unresolved references*. For example, if `module1` uses `printf`, there will be an unresolved reference to `printf` in the object file.

Linking

There is a tool called the *linker* that puts object files together. This includes resolving references:

- Linking an object file with an unresolved reference to `foo()` to an object file with the code for `foo()` resolves the reference.

The linker program is called `ld`, but you'll almost always just have `gcc` invoke the linker for you.

```
gcc -g -Wall -c module1.c
gcc -g -Wall -c module2.c
gcc -g -Wall -c main.c
gcc -o program main.o module1.o module2.o
```

`gcc` implicitly links against the standard C libraries.

Great, but *why did we bother doing this?*

Why Object Files and Linkers?

Linking object files has two big advantages:

- Can ship code around without source or building an executable
  - Usually done as a *library*, which is a slighly souped-up object file
- Can (re)compile part of your program without compiling *all* of it
  - Recompile only what has changed, avoid wasting time recompiling unaffected code
  - Compiling just the core parts of Linux or Solaris takes 8+ hours; compiling all of Windows™ takes *weeks*

But there are disadvantages, too:

- It is tedious to re-type these commands as needed
- It is easy to make mistakes about which parts of your program need to be recompiled.

Introducing `make`

`make` is a tool that does all of this tedious work for you. All you do is:

- Write a *makefile*
  - Specify what depends on what. For example:
    - An object file depends on its C source and the headers that source uses
    - The program depends on all the object files
  - Specify how to generate a file given what it depends on
    - e.g. how to generate an object file from C files and headers
- Run the `make` command and tell it what final result you would like.

Something to generate is called a *target*. Something processed to produce a target is called a *source*.

Format of a Makefile

```
target: source1 source2 ...
	command1
	command2
	...
queue.o: queue.c queue.h
	gcc -g -Wall -c queue.c
```

Command lines **must** start with a TAB character, not spaces. You can split a long command across multiple lines by putting \ at the end of the first line.

Running `make`

On the command line:

```
make target
```

- Looks for `Makefile` in the current directory, can be overridden with `-f file`
- If no target specified, uses first target in file

Make decides what to do based on the dependency graph and file modification times.

Standard `make` Targets

- `all`: make everything

  ```
  all: client server
  ```

- `clean`: remove generated files, start over with just source

  ```
  clean:
      rm -f *.o client server
  ```

- These are called *phony* targets, because they never exist on disk.

Advanced Makefiles

How many times do you want to write all the arguments to `gcc`? How many places do you want to update arguments to `gcc`?

Variables / Macros

`make` allows variables to hold common expressions. For example:

```
CC = gcc
CFLAGS = -g -Wall
queue.o: queue.c queue.h
	$(CC) $(CFLAGS) -c queue.c
```

Built-in Macros and Patterns

Built-in Macros

- `$@` current target
- `$^` all sources
- `$<` leftmost source

Patterns

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

Automatically Generating Dependencies

We now have a tool to exploit dependencies for us, but it's still a hassle to write them down correctly.

- `gcc -MM [src files]`
  - Useful variants like `-M` and `-MG` (`man gcc`)
  - Automatically create makefile rules to generate object files
  - Often run via a phony target:

    ```
    depend: $(SRC)
        $(CC) -M $^ > .depend
    ```

  - Then `include` the result in your makefile:

    ```
    include .depend
    ```

- Also a tool called `makedepend`
- Read more if you're interested

GCC Flags

`-o` *file*	Writes result to *file*
`-c`	Stops compilation with an object file; no linking
`-g`	Outputs debugging information
`-O`*n*	Uses optimization level *n*, for $0 \leq n \leq 3$
`-I` *dir*	Looks for header files in *dir* - an *include directory*
`-L` *dir*	Looks for libraries to link against in *dir*
`-l` *lib*	Link against the library *lib*
`-Wall`	Warn about anything questionable
`-Werror`	Treat all warnings as compilation errors

GDB

- `gdb` is the debugger accompanying `gcc`
- A text-mode debugger as an interactive shell, though GUI frontends exist
- Provides standard debugger functionality:
  - Breakpoints
  - Stepping over lines of code, into/out of functions
  - Stack traces
  - Print variables, heap structures
  - Listing code
- Also has more advanced functionality, like data breakpoints, disassembling code...
- Pretty good built-in help system (e.g. `help backtrace`)
- Not very useful without `-g` flag (emit debugging info) to `gcc`

Finding a SEGFAULT

```
#include<stdio.h>
#include<stdlib.h>
int main(int argc, char* argv[]) {
    int i;
    int total = 0;
    for (i = 0; i <= argc; i++) {
        total += atoi(argv[i]);
    }
    printf("The total is %d\n", total);
    return 0;
}
```

This program *should* print the sum of all the numbers in its arguments. Instead:

```
[csgordon@monarch:~/cse333]$ gcc -g sum.c -o sum
[csgordon@monarch:~/cse333]$ ./sum 3 4 5
Segmentation fault
[csgordon@monarch:~/cse333]$
```

Debugging a Segmentation Fault

First approach

Stare at the code. Really hard.

- Works sometimes.
- Often doesn't work at all.

Better approach

Run `gdb`

Debugging with GDB I

First, start gdb:

```
[csgordon@monarch:~/cse333]$ gdb sum
GNU gdb (GDB) Fedora (7.1-34.fc13)
...
Reading symbols from /homes/gws/csgordon/cse333/sum...done.
(gdb)
```

Now run the program with arguments:

```
(gdb) run 3 4 5
Starting program: /homes/gws/csgordon/cse333/sum 3 4 5

Program received signal SIGSEGV, Segmentation fault.
0x4debfc0c in ____strtol_l_internal () from /lib/libc.so.6
Missing separate debuginfos, use: debuginfo-install glibc-2.12.2-1.i686
(gdb)
```

Now we've reproduced the bug in the debugger...

Debugging with GDB II

Now we need to find out where we are

```
(gdb) backtrace
#0  0x4debfc0c in ____strtol_l_internal () from /lib/libc.so.6
#1  0x4debf970 in strtol () from /lib/libc.so.6
#2  0x4debc2c1 in atoi () from /lib/libc.so.6
#3  0x08048423 in main (argc=4, argv=0xbffff7e4) at sum.c:7
(gdb)
```

The library probably shouldn't be dereferencing a bad pointer unless we're providing bad input. It looks like we're calling the library at line 7, let's look at that:

```
(gdb) up 3
#3  0x08048423 in main (argc=4, argv=0xbffff7e4) at sum.c:7
7           total += atoi(argv[i]);
(gdb)
```

- Can also do `up` with no argument for moving by one frame
- There is also `down` to go in the other direction

Debugging with GDB III

What's wrong with our call?

```
...
#3  0x08048423 in main (argc=4, argv=0xbffff7e4) at sum.c:7
7         total += atoi(argv[i]);
(gdb) print i
$1 = 4
(gdb) p argv[i]
$2 = 0x0
(gdb) quit
```

Breakpoints

```
[csgordon@monarch:~/cse333]$ gdb sum
...
(gdb) break sum.c:7
Breakpoint 1 at 0x804840f: file sum.c, line 7.
(gdb) r 3 4 5
Starting program: /homes/gws/csgordon/cse333/sum 3 4 5

Breakpoint 1, main (argc=4, argv=0xbffff7f4) at sum.c:7
7          total += atoi(argv[i]);
(gdb) p i
$1 = 0
(gdb) p argv[i]
$2 = 0xbffff991 "/homes/gws/csgordon/cse333/sum"
(gdb) c
Continuing.

Breakpoint 1, main (argc=4, argv=0xbffff7f4) at sum.c:7
7          total += atoi(argv[i]);
(gdb) p i
$3 = 1
(gdb) p argv[i]
$4 = 0xbffff9b0 "3"
(gdb) q
[csgordon@monarch:~/cse333]$
```

GDB Cheat Sheet

Abbr.	Command	Result
r	run *args*	Runs program from the start with *args*
b	break *file*:*n*	Sets a breakpoint on line *n* of *file*
b	break *fn*	Sets a breakpoint at start of *fn*
b	break *file*:*fn*	Sets a breakpoint at start of *fn* in *file*
d	delete *breakpoint*	Delete breakpoint *breakpoint*, which can be a file and line number or a breakpoint number
	info breakpoints	List current breakpoints
	info locals	List local variables
	info variables	List local & global variables
c	continue	Continues execution to next breakpoint
n	next	Execute one statement and stop
s	step	Step inside function
l	list	Lists code: defaults to current code, takes optional location
bt	backtrace	Show stack trace, with arguments
w	where	Show stack trace, with arguments
h	help *topic*	Get help on *topic*

Text Editors

Linux has many great text editors. We don't care which you use, but here are a few options.

- gedit - like Notepad or TextEdit, but with syntax highlighting
- Eclipse - has a C/C++ mode
- emacs - probably seen it before, a very powerful text editor, which can integrate with build systems, version control, debuggers...
- vim - another powerful text editor with a more unusual interface

All of these have syntax highlighting, which might make it easier to read code. We will post links to tutorials for the more complicated editors online.