Section 1: Tools

August 21, 2015

Contents

This section is to give an introduction to tools that we will use in homework 0 and CS 162.

1 Make

In software development, Make is a utility that automatically builds executable programs and libraries from source code by reading files called Makefiles which specify how to derive the target program.

1.1 Motivation

You are working on a project in C that involves four files, specified in Appendix A. All section codes are available at https://github.com/Berkeley-CS162/Sections. You want to compile and link them to an executable file. First a compiler compiles each C file to object files. Then a linker links all object files together to form a executable. Here shows the commands you need to type.

```
$ gcc -c main.c
$ gcc -c hello.c
$ gcc -c factorial.c
$ gcc hello.o factorial.o main.o -o hello
```

However this approach doesn't scale out. A large project could have thousands of source code files. It's not feasible to type all these commands every time you need to compile or link them. Make automates this process by specifying them in a Makefile. With a Makefile, you can just type

```
$ make
```

and the executable comes out.

1.2 How

A Makefile consists of a series 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.

```
target: dependencies
[tab] system commands
```

When Make is called on a target, the target's dependencies are checked first. If any of the are themselves targets, they will be called recursively if changes in the source code have happened since the last compialtion. Once all the targets are available and ready to use, the system commands for the original target call are executed. The Makefile for the example project can be found in Appendix A.5.

2 Git

Git is a distributed revision control and source code management (SCM) system with an emphasis on speed, data integrity, and support for distributed, non-linear workflows. GitHub is a Git repository web-based hosting service, which offers all of the distributed revision control and SCM functionality of Git as well as adding many useful and unique features.

2.1 Learn by Doing

A good way to become familiar with a tool is to use it. Create a public repository on your GitHub account, and give it a name of your choosing, e.g. foobar. Now execute the next few lines in your terminal, replacing strings within [] with your own strings. Following the example above, [projectname] will become foobar.

```
$ git clone https://github.com/[username]/[projectname].git
$ cd [projectname]/
$ touch [file name]
$ echo "some words" > [file name]
$ git status
$ git add *
$ git commit -m "first commit"
$ git log --graph --all --decorate --color
$ git push --all
```

Make some changes to README on github.

```
$ git pull
$ git log --graph --all --decorate
$ git checkout [first commit hash code]
$ git branch parallel-world
$ git branch
$ git checkout parallel-world
```

Make some changes to [file name].

```
$ git add *
$ git commit -m "new branch"
$ git push -all
$ git log --graph --all --decorate --color
```

2.2 Commands to Know

- **git clone <url>**
  Clone a repository into a new directory
- **git pull [repo] [branch]**
  Fetch from and integrate with another repository or a local branch
- **git push [repo] [branch]**
  Update remote refs along with associated objects
- **git add <file(s)>**
  Add file contents to the index
- **git commit -m "commit message"**
  Record changes to the repository with the provided commit message
- **git branch**
  List, create, or delete branches
- **git checkout**
  Checkout a branch or paths to the working tree

- **git merge**
  Join two or more development histories together
- **git log**
  Show commit logs
- **git status**
  Show the working tree status

3 GDB: The GNU Debugger

GDB is the standard debugger for the GNU operating system.

3.1 Learn by doing

The codes we are using are in Appendix B.

```
$ make
$ gdb main
(gdb) break main
(gdb) run
(gdb) kill

(gdb) break main
(gdb) r
(gdb) s
(gdb) n 1
(gdb) n
(gdb) p ptr
(gdb) p *ptr
(gdb) c

(gbd) r
(gdb) clear
(gdb) kill

(gdb) break main
(gdb) r
(gdb) watch ptr
(gdb) c
(gdb) c
(gdb) q
```

uncomment `//print_ptr_val (ptr);`

```
$ make clean
$ make
$ gdb main
(gdb) r
(gdb) bt
(gdb) bt full
(gdb) q
```

(Note: Mac OS does not support follow-fork-mode)
comment out `print_ptr_val (ptr);`
uncomment `//fork_print_process ();`

```
$ make clean
$ make
$ gdb main
(gdb) set follow-fork-mode child
(gdb) break fork_print_process
(gdb) r
(gdb) n
(gdb) n
(gdb) n
(gdb) kill

(gdb) set follow-fork-mode parent
(gdb) break fork_print_process
(gdb) r
(gdb) n
(gdb) n
(gdb) n
(gdb) q
```

3.2 Commands to know

- **run, r**
  Start program execution from the beginning of the program. The command break main will get you started. Also allows basic I/O redirection.
- **quit, q**
  Exit GDB debugger.
- **kill**
  Stop program execution.
- **break, break x if condition, clear**
  Suspend program at specified function of line number.
- **step, s**
  Step to next line of code. Will step into a function.
- **next, n**
  Execute next line of code. Will not enter functions.
- **continue, c**
  Continue execution to next breakpoint.
- **finish**
  Continue to end of function.
- **print, p**
  Print value stored in variable.
- **watch, rwatch, awatch**
  Suspend processing when condition is met. i.e. `x > 5`.

- **backtrace, bt, bt full**
  Show trace of where you are currently. Which functions you are in. Prints stack backtrace.
- **set follow-fork-mode mode** (Mac OS does not support this)
  Set the debugger response to a program call of fork or vfork. A call to fork or vfork creates a new process. The mode argument can be: parent or child.

4 tmux

Tmux is a software application that can be used to multiplex several virtual consoles, allowing a user to access multiple separate terminal sessions inside a single terminal window or remote terminal session. It is useful for dealing with multiple programs from a command-line interface, and for separating programs from the Unix shell that started the program. It also allows you to maintain persistent working states on remote servers.

4.1 Motivation

You are working on your server. It is compiling a project that takes 3 hours, but meanwhile you want to write documentations for your project, or you have to leave the cafeteria and disconnect your laptop from the Internet. Or maybe you want to edit your code in Vim and test the code simultaneously, so you'd rather have both Vim and the terminal on the screen, instead of switching back and forth between them.

4.2 Concepts

A *session* is a single collection of pseudo terminals under the management of tmux. Each session has one or more *windows* linked to it. A *window* occupies the entire screen and may be split into rectangular *panes*, each of which is a separate pseudo terminal. Any number of tmux instances may connect to the same *session*, and any number of *windows* may be present in the same *session*. Once all sessions are killed, tmux exits.

4.3 Commands to know

Note: All commands below use `C-b` as the prefix, but because that isn't as easy to type, on your virtual machines we remapped it to `C-a`.

- **Sessions**
  - `C-b s`
    Select a new session for the attached client interactively.
  - `C-b d`
    Detach the current client.
- **Windows**
  - `C-b c`
    Create a new window.
  - `C-b [0-9]`
    Select windows 0 to 9.

  - `C-b &`
    Kill the current window.

- **Panes**

  - `C-b %`
    Split the current pane into two, left and right.
  - `C-b "`
    Split the current pane into two, top and bottom.
  - `C-b C-o`
    Rotate the panes in the current window forwards.
  - `C-b x`
    Kill the current pane.

A Source Codes for Make

A.1 main.c

```c
#include <stdio.h>
#include "functions.h"

int main() {
    print_hello ();
    printf ("The 10th Fibonacci number is %d.\n", fibonacci (10));
    return 0;
}
```

A.2 hello.c

```c
#include <stdio.h>
#include "functions.h"

void print_hello () {
    printf ("Hello Luke! I am your father!\n");
}
```

A.3 fibonacci.c

```c
#include "functions.h"

int fibonacci (int n) {
  if (n == 0) return 0;
  if (n == 1) return 1;
  int current = 1, previous = 0, temp;
  while (n >= 2) {
    temp = current;
    current = current + previous;
    previous = temp;
    n--;
  }
  return current;
}
```

A.4 functions.h

```c
void print_hello ();
int fibonacci (int n);
```

A.5 Makefile

```
all: hello
hello: main.o fibonacci.o hello.o
  gcc main.o fibonacci.o hello.o -o hello
main.o: main.c
  gcc -c main.c
fibonacci.o: fibonacci.c
  gcc -c fibonacci.c
hello.o: hello.c
  gcc -c hello.c
clean:
  rm -rf *o hello
```

B Source Codes for GDB

B.1 main.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>

void print_ptr_val (int *ptr)
{
    printf ("int value %d\n", *ptr);
}

void fork_print_process ()
{
    pid_t pid = fork();

    if (pid == 0)
        printf ("I'm child process\n");
    else
        printf ("I'm parent process\n");
}

int main () {
    int *ptr = NULL;
    ptr = (int *) malloc(sizeof(int));
    *ptr = 42;
    print_ptr_val (ptr);
    free(ptr);
    ptr = NULL;
    //print_ptr_val (ptr);
    //fork_print_process ();
    return 0;
}
```

B.2 Makefile

```
all: main
main: main.o
  gcc -ggdb3 main.o -o main
main.o: main.c
  gcc -ggdb3 -c main.c
clean:
  rm -rf *o main
```