Section 0: Tools

CS162

August 28-30, 2019

Contents

Tools are important for every programmer. If you spend time learning to use your tools, you will save even more time when you are writing and debugging code. This section will introduce the most important tools for this course.

1 Make

GNU Make is program that is commonly used to build other programs. When you run `make`, GNU Make looks in your current directory for a file named `Makefile` and executes the commands inside, according to the makefile language.

```
my_first_makefile_rule:
        echo "Hello world"
```

The building block of GNU Make is a **rule**. We just created a rule, whose **target** is `my_first_makefile_rule` and **recipe** is `echo "Hello world"`. When we run `make my_first_makefile_rule`, GNU Make will execute the steps in the recipe and print "Hello world".

Rules can also contain a list of **dependencies**, which are other targets that must be executed before the rule. In this example, the `task_two` rule has a single dependency: `task_one`. If we run "`make task_two`", then GNU Make will run `task_one` and then `task_two`.

```
task_one:
        echo 1
task_two: task_one
        echo 2
```

1.1 More details about Make

- If you just run `make` with no specified target, then GNU Make will build the first target.
- By convention, target names are also file names. If a rule's file exists and the file is **newer** than all of its dependencies, then GNU Make will skip the recipe. If a rule's file does not exist, then the timestamp of the target would be "the beginning of time". Otherwise, the timestamp of the target is the **Modification Time** of the corresponding file.
- When you run "`make clean`", the "clean" recipe is executed every time, because a corresponding file named "clean" is never actually created. (You can also use the `.PHONY` feature of the makefile language to make this more robust.)
- Makefile recipes must be indented with tabs, not spaces.
- You can run recipes in parallel with "`make -j 4`" (specify the number of parallel tasks).
- GNU Make creates automatic rules if you don't specify them. For example, if you create a file named `my_program.c`, GNU Make will know how to compile it if you run "`make my_program`".
- There are many features of the makefile language. Special variables like `$@` and `$<` are commonly used in Makefiles. Look up the documentation online for more!

Pintos, the educational operating system that you will use for projects, has a complex build system written with Makefiles. Understanding GNU Make will help you navigate the Pintos build system.

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 hosting service, which offers all of the distributed revision control and SCM functionality of Git as well as adding many useful and unique features.

In this course, we will use Git and GitHub to manage all of our source code. It's important that you learn Git, but NOT just by reading about it. If you're unfamiliar with Git, go to https://try.github.io/ and try it out yourself.

2.1 Commands to know

- **git init**
  Create a repository in the current directory
- **git clone <url>**
  Clone a repository into a new directory
- **git status**
  Show the working tree status
- **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 or delete branches
- **git checkout**
  Checkout a branch or paths to the working tree
- **git merge**
  Join two or more development histories together
- **git rebase**
  Reapply commits on top of another base
- **git diff [--staged]**
  Show a line-by-line comparison between the current directory and the index (or between the index and HEAD, if you specify --staged).
- **git show [--format=raw] <tree-ish>**
  Show the details of anything (a commit, a branch, a tag).
- **git reset [--hard] <tree-ish>**
  Reset the current state of the repository
- **git log**
  Show commits on the current branch

- **git reflog**
  Show recent changes to the local repository

When you begin working on group projects for this course, an advanced knowledge of Git will help you manage your team's code. Git is especially useful when multiple developers want to work on the same codebase simultaneously.

3 GDB: The GNU Debugger

GDB is a debugger that supports C, C++, and other languages. You will not be able to debug your projects effectively without advanced knowledge of GDB, so make sure to familiarize yourself with GDB as soon as possible.

3.1 Commands to know

- **run, r**
  Start program execution from the beginning of the program. Also allows argument passing and basic I/O redirection.
- **quit, q**
  Exit GDB
- **kill**
  Stop program execution.
- **break, break x if condition, clear**
  Suspend program at specified function (e.g. "`break strcpy`") or line number (e.g. "`break file.c:80`"). The "clear" command will remove the current breakpoint.
- **step, s**
  If the current line of code contains a function call, GDB will step into the body of the called function. Otherwise, GDB will execute the current line of code and stop at the next line.
- **next, n**
  Execute the current line of code and stop at the next line.
- **continue, c**
  Continue execution (until the next breakpoint).
- **finish**
  Continue to end of the current function.
- **print, p**
  Print value stored in variable.
- **call**
  Execute arbitrary code and print the result.
- **watch, rwatch, awatch**
  Suspend program when condition is met. i.e. `x > 5`.
- **backtrace, bt, bt full**
  Show stack trace of the current state of the program.

- **disassemble**
  Show an assembly language representation of the current function.
- **set follow-fork-mode <mode>** (Mac OS does not support this)
  GDB can only debug 1 process at a time. When a process forks itself (creates a clone of itself), we can follow the parent (original) or the child (clone). The mode argument can be either `parent` or `child`.

The **print** and **call** commands can be used to execute arbitrary lines of code while your program is running! You can assign values or call functions. For example, "`call close(0)`" or "`print i = 4`". (You can actually use **print** and **call** interchangeably most of the time.) This is one of the most powerful features of gdb.

4 Debugging programs on Linux

Debuggers and print statements are not the only ways to debug programs. Your student VM comes with a bunch of tools that provide information about running programs. Here are some of them that you will find useful for completing homework in this course.

4.1 strace and ltrace

All programs use standard library functions like `printf`, and most library functions involve **system calls** that interact with the kernel. This is true whether your program is written in C, Python, Java, or any other language. strace and ltrace are two programs that print out system calls (strace) and standard library function calls (ltrace) used by a program.

```
# Run 'ls files/', but print out system calls
strace ls files/

# Run 'ls files/', but print out library calls
ltrace ls files/

# Print out system calls of a running process with process ID = 1234
strace -p 1234

# Print out library calls of a running process with process ID = 1234
ltrace -p 1234
```

You can use strace and ltrace to debug a hung process, debug performance issues, or peek into the internals of somebody else's program if you're curious. You can add the `-f` flag to make `strace` and `ltrace` follow the descendants of the original process.

4.2 catchsegv

`catchsegv` will give you helpful information when your program encounters a segmentation fault.

```
catchsegv ./my_buggy_program
```

There are 3 parts to catchserv output. The 1st part prints out the values of all the CPU registers when the program crashed. The 2nd part prints out a stack trace of the crash, but only if you compiled with debug flags (-g). The 3rd part prints out the memory map of the crash—we will discuss shared memory in this course. `catchsegv` also handles aborts, traps, and other abnormal program crashes.

4.3 top

`top` is a text user interface for viewing the most active processes on your system. Press `q` to quit. Press `c` to show full commands. Press `<` or `>` to change the sorted column. Here are some of the most useful columns of top:

- PID – The process ID
- PR, NR – The priority value and nice value
- VIRT – The amount of virtual memory used by the process (KiB)
- RES – The amount of resident memory (actually located in RAM) used by the process (KiB)
- SHR – The amount of memory that is sharable (shared libraries, mapped files, etc) (KiB)
- S – The state of the process (S for sleeping, R for running, T for stopped, Z for zombie)
- %CPU, %MEM – The percentage of CPU and RAM used by the process
- TIME+ – The amount of "CPU time" used by the process. The unit for this is seconds. If a process is idle, it is using 0 CPU time.
- COMMAND – The name of the process

4.4 ps

`ps` prints out a list of running processes. By default, `ps` only prints the processes that are part of your own console session, which probably only includes **bash** and **ps** itself. You can use "`ps -eLf`" or "`ps auxww`" to get way more detail. (There's a long history about why one of these commands requires a dash.)

The `ps` command contains mostly the same information as top, but it is easier to filter and search, because it only appears once and shows all the processes.

The **PPID** column of "`ps -eLf`" tells you the **parent process ID**, which is the process ID of a process's parent. The **NLWP** column shows the number of light-weight processes (also known as threads) that belong to a particular process.

4.5 pidof and pgrep

`pidof <process name>` gets you the process ID of a single running process. For example, `pidof vim` will get me the process ID of my vim process. If there are multiple matches, then all the process IDs are printed. It's useful as a argument to another program. For example, I could attach gdb to my `httpserver` process by running "`gdb -p $(pidof httpserver)`".

You can also use `pgrep <process name>`, which doesn't require the full process name, but only a part of it. For example, "`pgrep a`" would give me the PID of all processes that contain 'a' in their name.

4.6 proc

The `/proc` directory on Linux is a gold mine of information. You can run `ls /proc/1` to get information about the process with PID = 1. Many of the tools mentioned earlier are implemented by reading the `/proc` directory, so if you need more detail about a process, look there! The `/proc/self` directory is a shortcut to the current process. (If you run `ls /proc/self`, then you will get information about the `ls` process!)

Some useful parts of proc:

- `cmdline` – The command used to start the process
- `cwd` – The current working directory of the process
- `environ` – The environment variables of the process
- `exe` – The program that is running in the process
- `fd/` – A list of the file descriptors of the process
- `fdinfo/` – More information about the file descriptors
- `maps` – The memory map of the process
- `net` – Details about network configuration
- `status` – Status of the process

4.7 lsof

`lsof` gets you information about the files, processes, and network connections on your system. Each line of output from `lsof` corresponds to an "open file", which can correspond to a process's current directory, an actual file that is in use by a process, a shared library, a network connection, or a chunk of shared memory. Here are a few useful ways to use it:

- `lsof /lib/x86_64-linux-gnu/libc.so.6` – List instances of programs using the C standard library
- `lsof -p 1234` – List open files for the process with process ID = 1234
- `lsof -i tcp:80` – List uses of TCP port 80 (http)

4.8 netstat and ip

`netstat` and `ip` are two commands that help you debug the network connections on your system. Here are a few useful ways to use them:

- You can run "`ip addr`" or "`ifconfig`" to get the IP addresses and network interfaces of the system.
- You can run "`ip route`" or "`netstat -nr`" to get the routing table.
- You can run "`netstat -nt`" to get the list of current TCP connections on the system.
- You can run "`netstat -ntl`" to get the TCP ports that are listening for new connections.

Several homework assignments in this course will require the use of network connections in your programs. We will introduce more tools for testing network connections when those homework assignments are released.

4.9 kill, pkill, and killall

To kill a process, you can use "`kill <PID>`". You can also attach a signal to the command, like "`kill -TERM <PID>`" or "`kill -QUIT <PID>`".

`pkill` will kill all processes that match a pattern. For example, "`pkill v`" will kill all processes with 'v' in the name. `killall` does the same thing as `pkill`, but requires an exact match.

5 Debugging Example

Take a moment to read through the code for `asuna.c`. It takes in 0 or 1 arguments. If an argument is provided, asuna uses quicksort to sort all the chars in the argument. If no argument is provided, then asuna uses a default string to sort.

```
int partition(char* a, int l, int r){
  int pivot, i, j, t;
  pivot = a[l];
  i = l; j = r+1;

  while(1){
    do
      ++i;
    while( a[i] <= pivot && i <= r );
    do
      --j;
    while( a[j] > pivot );
    if( i >= j )
      break;
    t = a[i];
    a[i] = a[j];
    a[j] = t;
  }
  t = a[l];
  a[l] = a[j];
  a[j] = t;
  return j;
}
```

```
void sort(char a[], int l, int r){
  int j;

  if(l < r){
    j = partition(a, l, r);
    sort(a, l, j-1);
    sort(a, j+1, r);
  }
}
```

```
void main(int argc, char** argv){
  char* a = NULL;
  if(argc > 1)
    a = argv[1];
  else
    a = "Asuna is the best char!";
  printf("Unsorted: \"%s\"\n", a);
  sort(a, 0, strlen(a) - 1);
  printf("Sorted:   \"%s\"\n", a);
}
```

When asuna is run, we get the following output:

```
$ ./asuna "Kirito is the best char!"
Unsorted: "Kirito is the best char!"
Sorted  : "    !Kabceehhiiiorrssttt"

$ ./asuna
Unsorted: "Asuna is the best char!"
Segmentation fault (core dumped)
```

Use the debugging tools to find why `asuna.c` crashes when no arguments are provided.

Hint: `catchsegv` is a great place to start for debugging seg faults.