NWEN 241 Crash Course: Basic Linux, Compiling, and Debugging

NWEN Tutors

March 8, 2019

School of Engineering and Computer Science

Basic Linux shell commands

Navigation

ls	List the contents of the current directory
cd <dir>	Change to directory `dir`
cd ..	Change to parent directory
mkdir <dir>	Create new directory named `dir`
rm <file>	Delete `file`
rm -r <dir>	Delete `dir`
cp <f1> <f2>	Make a copy of `f1` called `f2`
mv <f1> <f2>	Rename `f1` to `f2`
<up/down>	Navigate previous shell commands
history	Print shell command history

Common notation

```
$ this is a command you run
This is the output

# this is a command you run as admin (su/sudo)
This is the output

$ this is a command you run # this is a comment
This is the output
```

Compiling with GCC and G++

GCC

The *GNU Compiler Collection* (GCC) contains compilers for several languages:

- `gcc` (C)
- `g++` (C++)
- `gcj` (Java)
- `gccgo` (Go)
- and many more

Part of the *GNU Toolchain*, which also includes *GNU Make* (automation), *GNU Binutils* (linker & assembler), and the *GNU Debugger* (`gdb`).

Getting started

- Let's write a C program called `hello.c`
- Compile it:

```
$ gcc hello.c
```

- This produces a file called `a.out`. Run it:

```
$ ./a.out
Hello, World!
```

Named binaries

- You can give your compiled binary a name:

```
$ gcc hello.c -o hello
$ ./hello
Hello, World!
```

Named binaries

- You can give your compiled binary a name:

```
$ gcc hello.c -o hello
$ ./hello
Hello, World!
```

Tip

It is good practice to always name your binaries.

Compiling C++

- Let's write a C++ program called `hello.cc`
- Compile it:

```
$ g++ hello.cc -o hello
$ ./hello
Hello, World!
```

Compiling C++

- Let's write a C++ program called `hello.cc`
- Compile it:

```
$ g++ hello.cc -o hello
$ ./hello
Hello, World!
```

Note

You can use G++ to compile C code, but not the other way round.

More options

- Enable all warnings:

```
$ gcc hello.c -o hello -Wall
```

- Compile and link separately:

```
$ gcc -c hello.c        # produces hello.o
$ gcc hello.o -o hello # produces hello binary
$ ./hello
Hello, World!
```

More options

- Enable all warnings:

```
$ gcc hello.c -o hello -Wall
```

- Compile and link separately:

```
$ gcc -c hello.c        # produces hello.o
$ gcc hello.o -o hello # produces hello binary
$ ./hello
Hello, World!
```

Tip

You can use this to link together multiple `*.o` files compiled from multiple `*.c` files.

Debugging with GDB

Debugging with GDB

- `gdb` is the *GNU Debugger*
- Used to print stack traces, stop program at predefined breakpoints, or step through line by line
- Allows us to see values of each variable at each line

Using GDB

- Compile the program with debug flags using `-g`:

```
$ gcc hello.c -g -o hello
```

- Load the program into `gdb`:

```
$ gdb hello
```

- Run the program within GDB:

```
(gdb) run
Starting program:
/~/.../hello
Hello, world!
[Inferior 1 (process 4372) exited normally]
```

Debugging Example

- Let's write a buggy program `crash1.cc`
- Compile and run it:

```
$ g++ -g crash1.cc -o crash1
$ ./crash1
"./crash1" terminated by signal SIGFPE
    (Floating point exception)
```

- Let's start debugging:

```
(gdb) r
Starting program:
/~/.../crash1
Program received signal SIGFPE, Arithmetic
exception. 0x0005555555551cc in divint
(a=3, b=0) at crash1.cc:18
18          return a / b;
```

Learning more

- Use `list` to show more context for the crash:

```
(gdb) list
13          return 0;
14      }
15
16      int divint(int a, int b)
17      {
18          return a / b;
19      }
```

Backtrace

- Do a backtrace:

```
(gdb) where
#0  0x0000555555555551cc in divint (a=3, b=0)
    at crash1.cc:18
#1  0x0000555555555551a9 in main ()
    at crash1.cc:11
```

- Traces back where the call came from
- Can also type `backtrace` or `bt`

Exploring the stack

- Let's move up the stack trace and get some more context:

```
(gdb) up
#1  0x0005555555551a9 in main ()
    at crash1.cc:11
11          cout << divint(x, y);
(gdb) list
6       {
7           int x = 5, y = 2;
8           cout << divint(x, y);
9
10          x =3; y = 0;
11          cout << divint(x, y);
12
13          return 0;
14      }
15
(gdb) print x
$1 = 3
(gdb) print y
$2 = 0
```

A more complex example

- Let's try another buggy program `crash2.c`:

```
$ ./crash2
Enter a number: 1
"./crash2" terminated by signal SIGSEGV
    (Address boundary error)
```

- Start debugging:

```
$ gdb crash2
(gdb) run
Starting program: /~/.../crash2
Enter a number: 1

Program received signal SIGSEGV,
    Segmentation fault.
0x00007ffff7e5342a in _IO_str_overflow ()
    from /usr/lib/libc.so.6
```

Backtrace

```
(gdb) where
#0  0x00007ffff7e5342a in _IO_str_overflow ()
    from /usr/lib/libc.so.6
#1  0x00007ffff7e51d11 in _IO_default_xsputn ()
    from /usr/lib/libc.so.6
#2  0x00007ffff7e23876 in vfprintf ()
    from /usr/lib/libc.so.6
#3  0x00007ffff7e464a7 in vsprintf ()
    from /usr/lib/libc.so.6
#4  0x00007ffff7e2c5c8 in sprintf ()
    from /usr/lib/libc.so.6
#5  0x000055555555524a in print_sum ()
    at crash2.c:29
#6  0x000055555555527e in main ()
    at crash2.c:35
```

Setting breakpoints

- Set a breakpoint at the offending line:

```
(gdb) break 29
Breakpoint 1 at 0x55555555521a: file crash2.c, line 29.
```

- Run it again:

```
(gdb) run
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /~/.../crash2
Enter a number: 1

Breakpoint 1, print_sum () at crash2.c:29
29          sprintf(buf, "sum=%d", sum_to_n(atoi(line)));
```

Printing buffers

- What are we trying to print?

```
(gdb) print line
$1 = "1\000\000RUUUU\000"
```

- Where are we trying to print to?

```
(gdb) print buf
$2 = 0x0
```

Conditional breakpoints

- Break at line 11 **if** num is 50:

```
(gdb) break crash2.c:11
Breakpoint 1 at 0x11a7: file crash2.c, line 11.
(gdb) condition 1 num==50
(gdb) run
Starting program: /~/.../crash2
Enter a number: 50

Breakpoint 1, sum_to_n (num=50) at crash2.c:11
11          for(i = 1; i <= num; i++) {
```

Stepping through a program

```
(gdb) break crash2.c:26
Breakpoint 1 at 0x1207: file crash2.c, line 26.
(gdb) run
Starting program: /~/.../crash2
Enter a number: 50

Breakpoint 1, print_sum () at crash2.c:26
26              strtok(line, "\n");
```

- `next` or `n`: Next command, stepping *over* function calls

```
(gdb) next
29          sprintf(buf, "sum=%d", sum_to_n(atoi(line)));
```

- `step` or `s`: Next command, stepping *into* function calls

```
(gdb) step
sum_to_n (num=50) at crash2.c:9
9           int i, sum=0;
```

Other useful commands

until	Run until end of current loop
finish	Run until end of current function
watch <var>	Pause whenever `var` is modified
info break	List all breakpoints
delete <n>	Delete breakpoint `n`
delete	Delete all breakpoints
clear <fun>	Delete breakpoint set at `fun`
x <addr>	Print content at `addr`:

```
(gdb) print &num
$1 = (int *) 0xbffff580
(gdb) x 0xbffff580
0xbffff580: 0x00000064
```

Working remotely and transferring files using SSH and SCP

Remote login using SSH

- **SSH** (*Secure Shell*): A secure network protocol for operating network services remotely over an unsecured network
- Installed by default on Linux and macOS; Windows 10 uses **OpenSSH** (Alternative: **PuTTY**)

Using SSH

- `ssh <user>@<host>`
  - `user` is your ECS username (same as when logging into physical machines)
  - `host` can be IP address or domain name (e.g. `logan-brown.ecs.vuw.ac.nz`)
  - There is no visual feedback while you type your password!
  - Other available machines:
    - `barretts.ecs.vuw.ac.nz`
    - `embassy.ecs.vuw.ac.nz`
    - `greta-pt.ecs.vuw.ac.nz`
    - `regent.ecs.vuw.ac.nz`

Transferring files using SCP

- **SCP** (*Secure Copy Protocol*): An SSH-based protocol to securely transfer files between two hosts on a network
- Available by default on Linux and macOS; Windows: **WinSCP**

Using SCP

- Copying from *local* to *remote*:
  - `scp <file> <user>@<host>:[<path>]`
    - `path` is *optional*; if left out, file will be copied to your home directory on the remote host
    - However, the `:` is *required*! Otherwise you will copy to a local file with the name `host`
- Copying from *remote* to *local*:
  - `scp <user>@<host>:<path> <local_path>`
    - `local_path` is *required*, but you can use `.` to copy to current directory

Example remote workflow

1. Edit code locally
2. Upload using `scp`
3. Remote login using `ssh`
4. Compile remotely using `gcc`/`g++`
5. Debug remotely using `gdb`

Alternatively, some editors allow local editing of remote files using SCP as a base protocol (i.e., the file is automatically uploaded whenever you save) — this depends on the editor though, and may require setting up an SSH key.

Questions?