

CMPSC 311 Scribe Notes

GDB, SVN

Lecture 9 - 2/3/16

Waleed Malik, Joshua Marini, Michael Mankos

Overview

This lecture mostly dealt with using GDB from the shell to debug programs. Previous lectures had mentioned GDB, but had not gone into details. The next topic of the lecture was Revision Control Systems, particularly svn and how to use it. This was the day project 1 was officially assigned.

Project 1

The first project was officially assigned today, to be due in 2 weeks. The project is a simple wordcount application that finds the number of occurrences of unique words in a large text file, puts them in alphabetical order, and prints the words and the number of occurrences of the words as an output file. This will be done first in C, then in the shell. The full project description is on Angel.

GDB

The GNU Debugger, or GDB, is a debugger for many Unix-like systems and can be used for several different programming languages. In this class, we will be using it for C programs. The debugger lets us analyze the code of a program in a much more in-depth way than by simply trace printing.

The GDB allows you to:

- See the execution path of your program. You can follow your program as it executes each line of code and find where exactly there is a problem.
- Examine and change the values of all variables. By doing this, you can test invalid inputs.
- Setup breakpoints, which allow you to pause a program at a certain point to change or examine variables before continuing.

To run the GDB on a program from the shell, this is the code to use:

```
$gcc -g -o [program] [program].c
```

The “-g” is rather important because it supplies the GDB with useful information. If you don’t include it, you won’t be able to use GDB effectively.

Some important commands for the GDB:

- b <function>--create a breakpoint upon entering a function in the program. You can also use b <line number> to set a breakpoint at a specific line of code.
- r--run the program directly, stopping at any breakpoints.
- n--execute the next statement.
- s--execute one step (this allows you to step into function calls and run through their code).
- c--continue running the program as normal.

- `p <variable>`--print the value of a variable. This is usually done once the program has been paused by a breakpoint.
- `bt`--backtrace the stack. This prints the last stack frame of the code, and is useful for “zooming in” on particular pieces of the stack.
- `fr <num>`--makes stackframe <num> the current frame for printing variables.
- `up / down`--go up or down the stack.
- `q`--quit GDB.
- `help`--get more GDB help, including a list of GDB commands and their uses.
- `info f`--gives you more information on the current stack frame.

For more on GDB, such as command descriptions and general information, go to <http://betterexplained.com/articles/debugging-with-gdb/>.

GDB Basics

GDB (GNU debugger) works with executable files. To use gdb, first compile your c file in the terminal with the gcc:

```
$gcc <name of file>
```

This command will create a file named "a.out" that you can run inside the terminal with

```
$/a.out
```

You can alternatively compile your file with gcc and give your file a specific file rather than letting it defaultly be named "a.out":

```
$gcc -g <name of file> -o <enter the name for your executable>
```

Here, you can run your file with:

```
$/<what you named your executable>
```

Now that we reviewed how to compile files, let us talk about how to use gdb!

To enter gdb, enter the following command into your shell:

```
$gdb <name of executable>
```

This command enters the gdb with the executable file specified loaded into memory. You can also enter gdb with just:

```
$gdb
```

but you will have to specify which file you want to debug with:

```
(gdb)file example.c
```

When you enter gdb, the “\$” prompt you see in the terminal will be replaced with “(gdb)”.

where "example.c" is the name of our file. We will use the program in the lecture 9 slides on slide number 9 to step you through gdb.

[QUICK NOTE: To clear your screen of clutter in the gdb, use the hotkey "control-L".]

To run the program, simply enter "r" on the command line.

To set breakpoints, which are the bread and butter of any debugger, enter on the command line:

(gdb)b filename:<function name/line number>

Examples with the GDB

[NOTE: in each example shown below, we start our debugging session from the beginning of the program.]

```
Terminal
(gdb) b main
Breakpoint 1 at 0x4004f5: file example.c, line 6.
(gdb) r
Starting program: /home/michael/Documents/CMPSC 311/example.exe

Breakpoint 1, main () at example.c:6
6          int loc = 2;
(gdb) n
7          foo ();
(gdb) s
foo () at example.c:12
12         bar();
(gdb) s
bar () at example.c:17
17         gee();
(gdb) s
gee () at example.c:22
22         return;
(gdb) n
23     }
(gdb) n
bar () at example.c:18
18         return;
(gdb) n
19     }
(gdb) n
foo () at example.c:13
13         return;
(gdb) n
14     }
(gdb) n
main () at example.c:8
8         return 0;
(gdb) n
9     }
(gdb) n
0x00007ffff7a35ec5 in __libc_start_main ()
    from /lib/x86_64-linux-gnu/libc.so.6
(gdb) n
Single stepping until exit from function __libc_start_main,
which has no line number information.
[Inferior 1 (process 4497) exited normally]
(gdb)
```

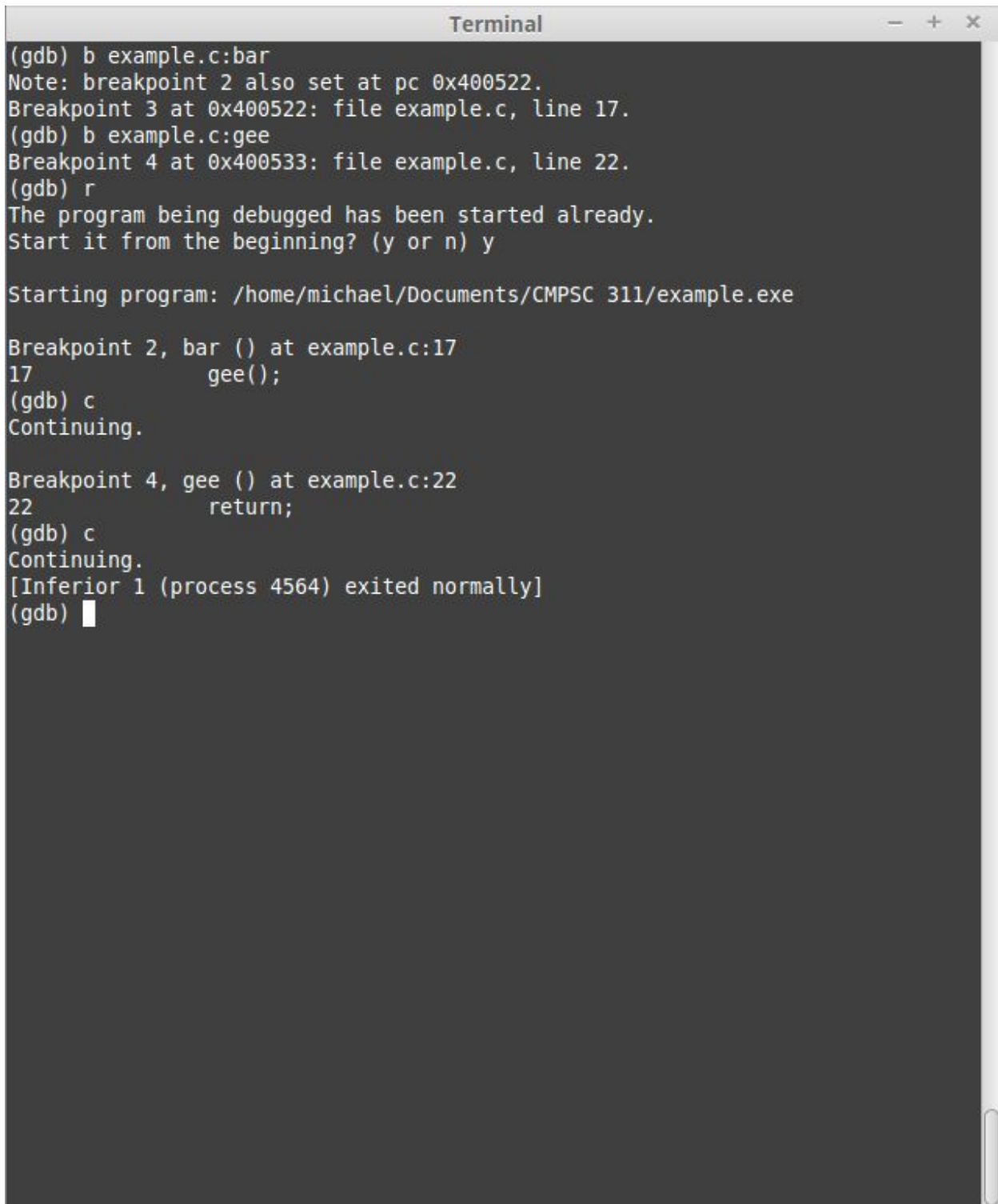
Figure 1a

We initially set a breakpoint at the first line in “int main()” with:

```
(gdb)b main
```

After setting the initial breakpoint, we run our program with “r”. (If we didn’t set a breakpoint and ran the program then we would have just ran through the whole program.) From there, we use

to “s” to step into a function. Note that at line 7, we step INTO foo(). We use “n” to step over to the next command. This steps to the next statement in your current function.

A terminal window titled "Terminal" with standard window controls (minimize, maximize, close) in the top right corner. The window contains a GDB session transcript. The user sets breakpoints at 'example.c:bar' and 'example.c:gee'. They then run the program with 'r'. The program starts at '/home/michael/Documents/CMPSC 311/example.exe'. The first breakpoint is hit at line 17, 'gee();', and the user continues with 'c'. The second breakpoint is hit at line 22, 'return;', and the user continues with 'c'. The program then exits normally, and the user is back at the GDB prompt.

```
(gdb) b example.c:bar
Note: breakpoint 2 also set at pc 0x400522.
Breakpoint 3 at 0x400522: file example.c, line 17.
(gdb) b example.c:gee
Breakpoint 4 at 0x400533: file example.c, line 22.
(gdb) r
The program being debugged has been started already.
Start it from the beginning? (y or n) y

Starting program: /home/michael/Documents/CMPSC 311/example.exe

Breakpoint 2, bar () at example.c:17
17         gee();
(gdb) c
Continuing.

Breakpoint 4, gee () at example.c:22
22         return;
(gdb) c
Continuing.
[Inferior 1 (process 4564) exited normally]
(gdb) █
```

Figure 1b

Before running the program here, we set a breakpoint at bar() and gee() with the commands:

```
(gdb)b example.c:bar()
```

```
(gdb)b example.c:gee()
```

This is an alternative to

```
(gdb)b example.c:<line number>
```

We specify the file we want to set breakpoints at. Notice how we use the source file and not the executable file to set breakpoints.

We used the “c” command to jump to the next breakpoint and skip over any intermediate commands.


```
Terminal
(gdb) r
The program being debugged has been started already.
Start it from the beginning? (y or n) y

Starting program: /home/michael/Documents/CMPSC 311/example.exe

Breakpoint 5, main () at example.c:6
6          int loc = 2;
(gdb) list
1      void foo ();
2      void bar ();
3      void gee ();
4
5      int main (){
6          int loc = 2;
7          foo ();
8          return 0;
9      }
10
(gdb) p loc
$2 = 0
(gdb) c
Continuing.

Breakpoint 7, main () at example.c:7
7          foo ();
(gdb) list
2      void bar ();
3      void gee ();
4
5      int main (){
6          int loc = 2;
7          foo ();
8          return 0;
9      }
10
11     void foo(){
(gdb) p loc
$3 = 2
(gdb)
```

Figure 1c

Here we use the command “list” to print out the rest of the program starting from the line you are on. As we advance to line 2 and use the “list” command again, only the part of the program starting from line 2 is printed.

We can also use the “p” command to print out a specific variable. Here we print out the value of loc before and after it is assigned the value 2. You can also use

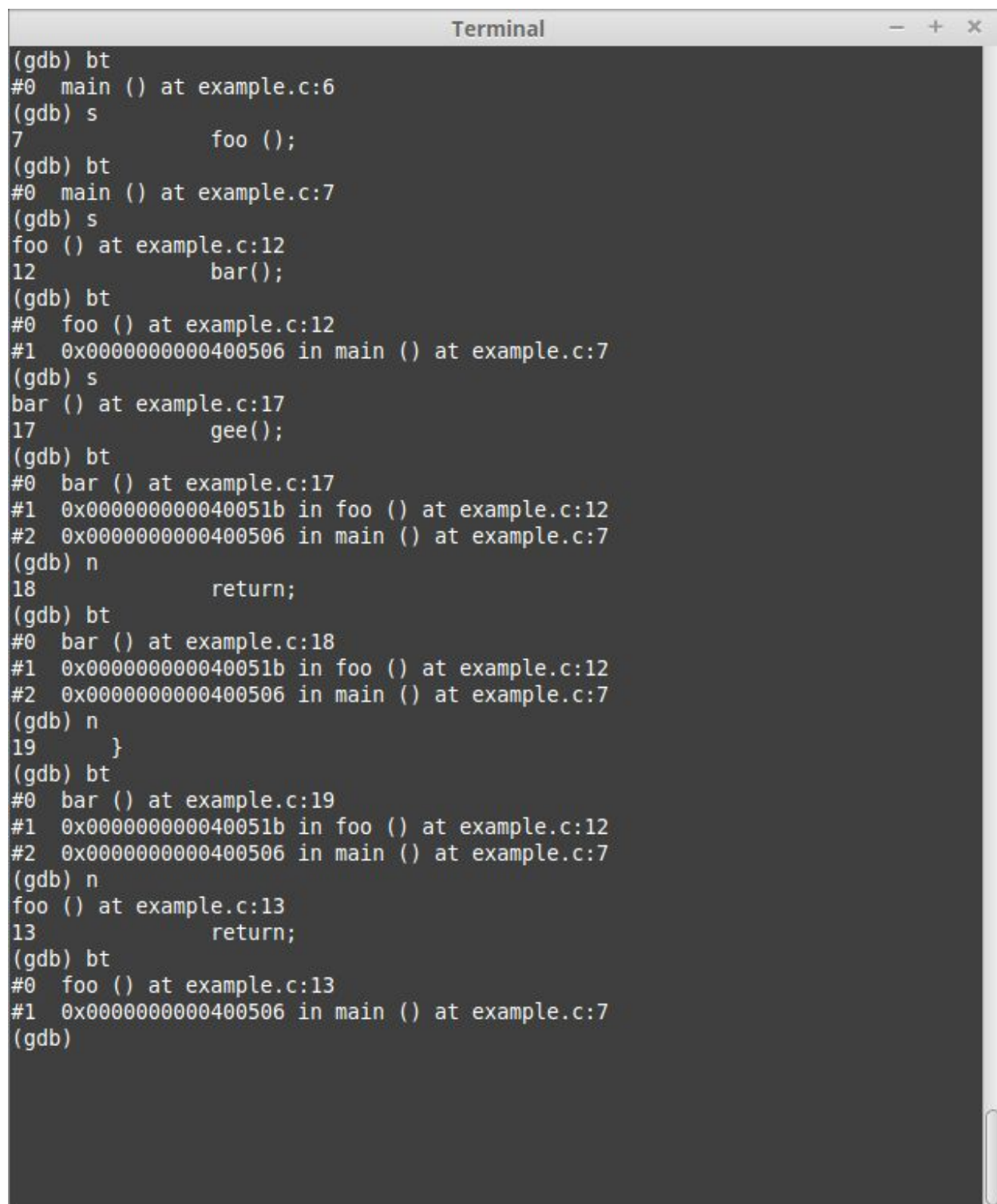
```
(gdb)info locals
```

to print out all the local variables and their current values.

Using

```
(gdb)whatis <variable_name>
```

will print the type of the variable you entered.



```
(gdb) bt
#0 main () at example.c:6
(gdb) s
7          foo ();
(gdb) bt
#0 main () at example.c:7
(gdb) s
foo () at example.c:12
12          bar();
(gdb) bt
#0 foo () at example.c:12
#1 0x000000000400506 in main () at example.c:7
(gdb) s
bar () at example.c:17
17          gee();
(gdb) bt
#0 bar () at example.c:17
#1 0x00000000040051b in foo () at example.c:12
#2 0x000000000400506 in main () at example.c:7
(gdb) n
18          return;
(gdb) bt
#0 bar () at example.c:18
#1 0x00000000040051b in foo () at example.c:12
#2 0x000000000400506 in main () at example.c:7
(gdb) n
19      }
(gdb) bt
#0 bar () at example.c:19
#1 0x00000000040051b in foo () at example.c:12
#2 0x000000000400506 in main () at example.c:7
(gdb) n
foo () at example.c:13
13          return;
(gdb) bt
#0 foo () at example.c:13
#1 0x000000000400506 in main () at example.c:7
(gdb)
```

Figure 1d

The “bt” command (short for backtrace) prints out the current stack at the current location in the program. This command is very useful for pinpointing program crashes as it shows you where in your program you were when it crashed. In this figure, every time we enter a new function, a

new item is added onto the stack, e.g. when we entered the function `bar()`. When we leave `bar()`, it is pulled off the stack.

```
Terminal
(gdb) s
7          foo ();
(gdb) s
foo () at example.c:12
12         bar();
(gdb) s
bar () at example.c:17
17         gee();
(gdb) s
gee () at example.c:22
22         return;
(gdb) bt
#0  gee () at example.c:22
#1  0x00000000040052c in bar () at example.c:17
#2  0x00000000040051b in foo () at example.c:12
#3  0x000000000400506 in main () at example.c:7
(gdb) frame 0
#0  gee () at example.c:22
22         return;
(gdb) info frame
Stack level 0, frame at 0x7fffffffef0a0:
  rip = 0x400533 in gee (example.c:22); saved rip = 0x40052c
  called by frame at 0x7fffffffef0b0
  source language c.
  Arglist at 0x7fffffffef090, args:
  Locals at 0x7fffffffef090, Previous frame's sp is 0x7fffffffef0a0
  Saved registers:
    rbp at 0x7fffffffef090, rip at 0x7fffffffef098
(gdb) frame 2
#2  0x00000000040051b in foo () at example.c:12
12         bar();
(gdb) info frame
Stack level 2, frame at 0x7fffffffef0c0:
  rip = 0x40051b in foo (example.c:12); saved rip = 0x400506
  called by frame at 0x7fffffffef0e0, caller of frame at 0x7fffffffef0b0
  source language c.
  Arglist at 0x7fffffffef0b0, args:
  Locals at 0x7fffffffef0b0, Previous frame's sp is 0x7fffffffef0c0
  Saved registers:
    rbp at 0x7fffffffef0b0, rip at 0x7fffffffef0b8
(gdb)
```

Figure 1e

The “frame” command lets you select a specific frame on your stack by

```
(gdb)frame <stack_frame_number>
```

In the figure above, we select frame 0 and use

```
(gdb)info frame
```

to print out the current information on that stack frame.

Then we select frame 2 and use the “info frame” command again to print out the current stack frame information.

```
Terminal
(gdb) s
7          foo ();
(gdb) s
foo () at example.c:12
12          bar();
(gdb) s
bar () at example.c:17
17          gee();
(gdb) bt
#0  bar () at example.c:17
#1  0x000000000040051b in foo () at example.c:12
#2  0x0000000000400506 in main () at example.c:7
(gdb) frame 2
#2  0x0000000000400506 in main () at example.c:7
7          foo ();
(gdb) up
Initial frame selected; you cannot go up.
(gdb) down
#1  0x000000000040051b in foo () at example.c:12
12          bar();
(gdb) down
#0  bar () at example.c:17
17          gee();
(gdb) down
Bottom (innermost) frame selected; you cannot go down.
(gdb) up
#1  0x000000000040051b in foo () at example.c:12
12          bar();
(gdb) up
#2  0x0000000000400506 in main () at example.c:7
7          foo ();
(gdb) up
Initial frame selected; you cannot go up.
(gdb)
```

Figure 1f

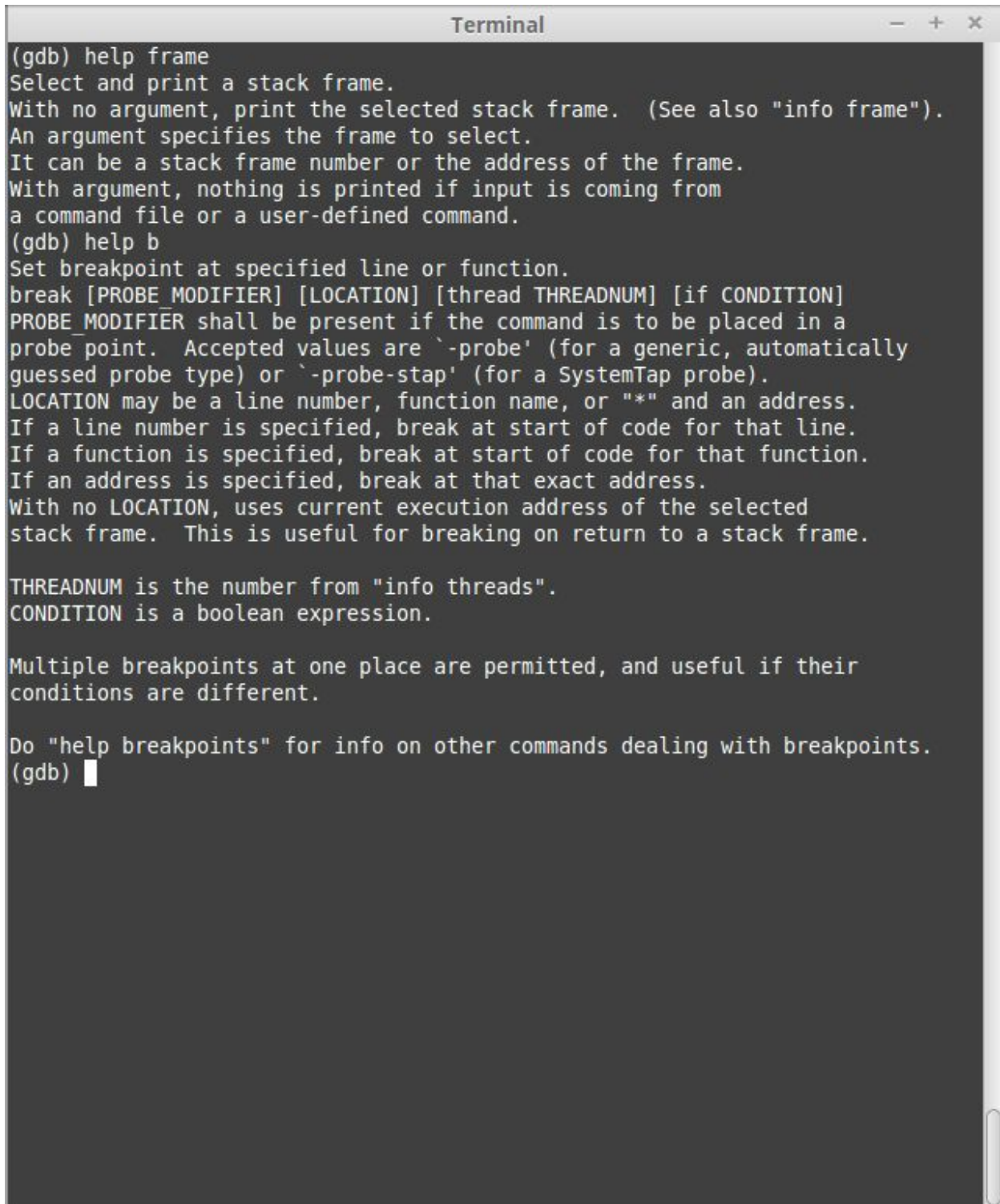
Once we select a specific stack frame with
(gdb)frame <stack_frame_number>
we can use

(gdb)up

(gdb)down

to select the stack frame above or below the current as an alternative to

(gdb)frame <stack_frame_number>

A terminal window titled "Terminal" with standard window controls (minimize, maximize, close) in the top right corner. The window has a dark background with light-colored text. The text inside shows the output of the GDB 'help' command for 'frame' and 'b'.

```
(gdb) help frame
Select and print a stack frame.
With no argument, print the selected stack frame.  (See also "info frame").
An argument specifies the frame to select.
It can be a stack frame number or the address of the frame.
With argument, nothing is printed if input is coming from
a command file or a user-defined command.
(gdb) help b
Set breakpoint at specified line or function.
break [PROBE MODIFIER] [LOCATION] [thread THREADNUM] [if CONDITION]
PROBE MODIFIER shall be present if the command is to be placed in a
probe point.  Accepted values are '-probe' (for a generic, automatically
guessed probe type) or '-probe-stap' (for a SystemTap probe).
LOCATION may be a line number, function name, or "*" and an address.
If a line number is specified, break at start of code for that line.
If a function is specified, break at start of code for that function.
If an address is specified, break at that exact address.
With no LOCATION, uses current execution address of the selected
stack frame.  This is useful for breaking on return to a stack frame.

THREADNUM is the number from "info threads".
CONDITION is a boolean expression.

Multiple breakpoints at one place are permitted, and useful if their
conditions are different.

Do "help breakpoints" for info on other commands dealing with breakpoints.
(gdb) █
```

Figure 1g

The last command in the lectures is the “help” command. It is very straightforward. Just write
(gdb)help <command>
to get help for a specific command.

Again, if you want to clear your current gdb terminal screen, use “Control-L”. For a list of more commands with concise description, visit the following URL:

<https://www.cs.swarthmore.edu/~newhall/unixhelp/howto_gdb.html>

Revision Control System

Revision Control Systems(RCS) allow developers to easily collaborate with others on a project. It allows developers to obtain a copy of the current source code, update it and then check in their changes. It even allows developers to go back in history and use the code from then. Not only is it used for maintaining codebases, people use it for other type of documents such as papers. Let's start by talking about the benefits of using a RCS.

The benefits of RCS:

- It allows you to go back in history and check something from that time. This is helpful when you find out something is broken and want to revert what has happened since then.
- It also can be used as a source of backups. If your local machine dies, there will still be another master copy on a different machine with all of the data.
- It allows you to see the changes that have occurred on file or even the whole project. This is very helpful when something breaks and you want to know what caused that.
- Lets you work together with multiple people at the same time. This is very important for working on big projects and working with many people. Otherwise you would have a big headache when trying to work to update the same file.
- Lets you create a “snapshot”. This is used when you are sending out a build or know this is a good state in your code.
- It allows you to work in different branches. Branches are copies of master where you can work on without affecting the master branch. Once you are done with your changes and know they are working fine you can merge them back to master. This allows you to work on your branch without having to worry it will break everything for other developers.

The master copy is stored in a repository. This is main place where everything is stored. The repository may be in the form of a database or even binary files. The repository is different from the copy you are editing, also known as the working copy. To edit, you need to first check out from the repository. This creates a local copy of the source code tree. You can make as many changes as you want on the working copy but none of this will be saved and reflected to others until you perform a commit operation.

Every revision is available to see and checkout. When you commit to a repository, the changes you make since the last commit are saved to the repository. Later on you are able to see the diff

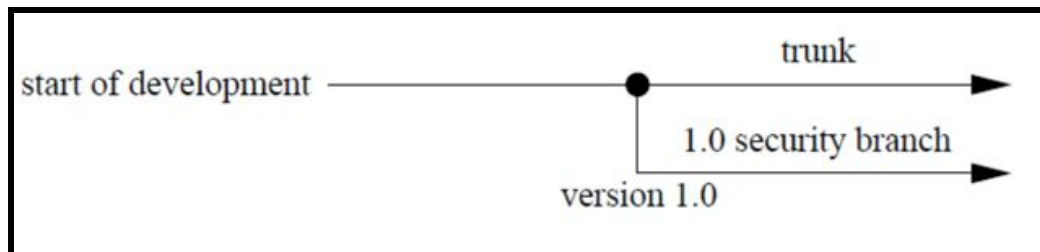
between each of these commits. This is helpful for seeing what changes were made on what files in the commit. The diffs are normally shown in the unix diff format. An example of this is below and the picture was taken from the lecture notes slides. The “-” is used to show lines that were removed and “+” is used to show lines that were updated and added. So in this example you start with the original file. Then you update the file to be what is in modified file. If you apply diff on these two files, this will be the output.

Original file	Modified file
This is a test file	This is a test file
It started with two lines	It no longer has two lines
	it has three

The “universal” diff (diff -u) between the two files is:

```
--- file      Mon Aug 28 11:31:53 2006
+++ file2     Mon Aug 28 11:32:07 2006
@@ -1,2 +1,3 @@
 This is a test file
-It started with two lines
+It no longer has two lines
+it has three
```

RCS allow multiple branches of development. These branches are used for many things. One example is for people to work on multiple things at a time. They can update one branch with some changes but not merge that with master if they think more needs to be done. It also is used when a project makes a release of a version. For example, you just finished the first version of a library. Now you want to work on the second version of the library. In the second version you most likely will be adding features. If you see any major security flaws you should go back and update the branch with version 1.0’s code. An example of this can be seen below and this picture was taken from the lecture slides.



One of most important features of RCS is the ability to do concurrent development. This allows multiple developers to work on the same project at the same time. Each starts off by checking out the code onto their local machines and they can all start editing at the same time. Now if one developer has a change, they can commit their changes without having to worry about others having committed. They simply will just have to commit and update. This normally works well but there are some cases where you may have a conflict. This usually is when two developers edit the same line. Then the RCS will prompt you and let you know there is a conflict. Usually the developer can figure out based off of the context of their commit what to change to fix the conflict.

One of the big revision control systems is Subversion(SVN). SVN is an open source control system used around in many different places. We will now go over the basics of using SVN.

Essential SVN commands:

- Create a repository - `svnadmin create [REPOSITORY NAME]`.
- Checkout a file - `svn checkout file: ///[PATH]/[REPOSITORY NAME]`
- Adding a file to a repository - `svn add [FILENAME]`
- Adding a directory to the repository - `mkdir [DIRNAME]; svn add [DIRNAME];`
- Committing changes - `svn commit -m "[COMMIT MESSAGE DESCRIBING COMMIT]"`
- Update to latest version - `svn update`

Other SVN commands:

- See the diff on a file - `svn diff [FILENAME]`
- Delete a file - `svn delete [FILENAME]`
- Status - `svn status`
- Log of history - `svn log`
- Creating snapshots - `svn copy`
- Blame to see author - `svn blame`

There are also other popular RCS systems such as Bitkeeper, Git, Visual SourceSafe, and Perforce.