Systems Development Basics: Debugging and Performance Tools

CS165: Data Systems — Fall 2015

September 15, 2015

This document provides a short introduction to tools commonly used for systems development. We cover basic source code management with Git as well as basic debugging with GDB and Valgrind.

Version Management with Git

Git is a distributed version control system that allows groups of people to concurrently work on the same documents without stepping on each other's toes. It also allows single users to efficiently manage source code changes, keeping track of all changes applied to a project and encourages users to keep multiple branches that represent different states of the same project.

Documentation
`http://git-scm.com`

Git Tutorials

CS61 Git Tutorial: `http://cs61.seas.harvard.edu/wiki/2015/Git`

SEAS Git Tutorial: `https://wiki.harvard.edu/confluence/display/USERDOCS/Introduction+To+GIT`

Giteveryday: `man giteveryday`

Getting started

There are two main approaches to getting a Git repository. To import an existing directory into Git, type the following command:

```
$ git init
```

This will initialize a new Git repository in the current working directory. A new `.git/` subdirectory will be created that contains all files necessary to manage the repository. At this point, none of the files inside the project directory are tracked, even if the directory was not empty.

If you want to copy an existing Git repository, such as your SEAS Code Repository clone of the CS165 skeleton repository[1], you can do that by using the `git clone` command. This will create a full-fledged Git repository that is completely isolated from the original repository, with its own local change history and files (unlike just being a "working copy").

[1] If you have not cloned the repository yet, you can go to `http://code.seas.harvard.edu`, search for the repository `cs165-2015-base` and click *clone repository*.

To clone the CS165 repository, type:

```
$ git clone <repository_url>
```

This will create a copy of the repository in a directory named `cs165-2015-base`. If you want to clone the repository into a repository named differently, you can specify your preferred name as follows:

```
$ git clone <repository_url> <directory_name>
```

Checking the repository state

To keep track of committed and uncommitted changes, it is often necessary to determine which files are in which state. This can be done by the following command:

```
$ git status
```

This command displays both the state of the working directory and the staging area (see *Staging modified files*). It lets you see which files are and which aren't tracked, as well as which changes have been staged for commit.

When you run the command directly after cloning the CS165 repository, you will see that there are neither untracked nor modified files in the directory. This will change once you add new files to or modify existing files in your project.

Tracking new files

Once in an initialized Git repository, you can start tracking new files by adding them to our repository:

```
$ git add <filename>
```

You can also add multiple files using the same command by either providing multiple input files or by using wildcards such as:

```
$ git add '*.c'
```

Staging modified files

Modified files that were already tracked need to be staged in order to be part of the commit. To do that, you can use the `git add` command just as you would do with untracked files. Running `git status` will now show the file as being a modified change to be committed rather than an unstaged change.

Committing your changes

When your staging area contains all the files you want, it is time to commit your changes. Remember that any files that have been changed but not added to the staging area will not go into this commit. To make sure all files are tracked and staged correctly it is useful to double check the repository state using the `git status` command. To commit the changes type:

```
$ git commit
```

This will display an editor that allows you to enter the commit message for your changes. Alternatively, you can inline your commit message using the `-m` flag:

```
$ git commit -m "Fixed memory leak in storage manager"
```

Syncing Git repositories

So far, all the changes have been committed to your local Git repository. This is fine as long as you are the only user of the repository. However, when collaborating with other developers, when using multiple machines to access your code (for instance, to benchmark the code on another machine) or even just when backing up the repository on a second machine, it becomes necessary to share changes between repositories.

In Git, local changes are published by pushing branches to other repositories (such as your SEAS Code Repository, which was already automatically set up as the `origin` branch, when you cloned the directory):

```
$ git push origin master
```

This will push the changes in the local `master` branch to the remote branch `origin`.

To see what others have contributed you need to fetch the changes from the remote branch and merge them into your local master branch For this, we use the `git pull` command:

```
$ git pull
```

This will fetch and merge the changes from the `origin` remote branch into the local `master` branch.

More features

Git supports many more useful features that we did not cover in this short introduction, such as using the `.gitignore` file to ignore files when tracking uncommitted files, which is especially useful for intermediate files such as swap files, displaying a project history using `git log` and highlighting changes between commits using the `git diff` command. If you are interested in those features, we recommend reading one of the more extensive tutorials that we link to in the margin.

Documentation
http://sourceware.org/gdb/current/onlinedocs/gdb/

Quick Reference Card
http://web.stanford.edu/class/cs107/gdb_refcard.pdf

Example Debug Session
http://www.cprogramming.com/gdb.html

Debugging code with GDB

Ever wondered what caused that heinous segmentation fault or why your program thinks 2+2 = 42 (which is not entirely wrong)? Wonder no more, GDB (the Gnu project debugger) allows you to see what is going inside your program while it executes. With GDB, you can:

- Start your program, specifying anything that might affect its behavior.
- Make your program stop on specified conditions.
- Examine what has happened, when your program has stopped.
- Change things in your program, so you can experiment with correcting the effects of one bug and go on to learn about another.

And here is how:

Pre-gdb instructions

Make sure to compile your program with `-g3` to include full level 3 debugging information. Turning off optimization (`-O0`) will ensure that line numbers in error messages are accurate.

Initial commands

A typical GDB debugging session, starts off like this:

```
gcc -Wall -Wextra -Werror -g3 program.cc -o prog
gdb prog
    (gdb)
```

Setting breakpoints

Once you have started gdb with your program, you can specify breakpoints. Breakpoints are locations within your code where your program pauses mid-execution. You can specify these locations as follows:

```
break <filename>:<line_number>
break <filename>:<func_name>
break <func_name>
```

If you have one file or a unique function name, the filename is optional.

Consider the code presented in Listing 1.

You can set a breakpoint at line 6 as `break main.c:6` or at the function `add` as `break main.c:add`. You can also get information about current breaks by using `info break`.

Listing 1: A simple C-program to demonstrate the usage of GDB

```c
#include <stdio.h>

int add(int num1, int num2){
  return num1+num2;
}

int main(int argc, char** argv){
  int i, j;
  i = 1;
  j = 2;
  int sum = add(i,j);
  printf("%d\n", sum);
  return 0;
}
```

Using Watchpoints

Watchpoints are a useful concept in GDB that allows you to stop execution when the value of an expression changes. The expression can be as simple as a variable (such as `(watch i)`) or as complex as many variables combined by operators (such as `(watch i + j < -1)`).

```
watch <expression> / w <expression>
```

Watchpoints are pretty handy in tracking variables that should not change. They are pretty handy when you realize that somewhere in your code a variable has changed. You can then set a watchpoint for that variable and see where that happened.

Running your code

Once you have set up your environment, you can use `run <args>` to start executing your code. The code will execute until it reaches a breakpoint or a watchpoint is triggered.

Navigating your code

Depending on the location of your breakpoint or watchpoint, the execution of your code is now stopped at a specific position. This could be in the middle of a function, at the start of a loop or at the end of conditional jump. GDB provides you a ton of functionality to move through the execution of your code.

continue <*count*> / ***c*** <*count*> Continues execution, if count is specified, gdb ignores this breakpoint count times.

step <*count*> **/** ***s*** <*count*> Executes until the next line is reached. If there is a function in the execution, this will step into the function. If count is specified, it will step <*count*> times.

next <*count*> **/** ***n*** <*count*> Executes the next line including any function call. If there is a function in the execution, this will not step into the function. If count is specified, it will next <*count*> times.

finish **/** ***f*** Continues execution until the current function returns.

Navigating the stack

If you set a breakpoint within a function, you will be in its stack frame. GDB gives you the ability to navigate between different stack frames.

info frame Gives you information about the current frame.

frame *n* Selects the frame number *n*. If *n* is not specified, the current frame is displayed.

up Selects a frame one frame up. For example, in the code snippet in listing 1 if you set a breakpoint at the function `add`. The program will pause execution at the start of the function. If you typed in up at this time it will take you one frame up i.e., the location in the program from where this function is called.

backtrace **/** ***bt*** **/** ***where*** Gives you information about all the frames in the stack. For example, if you had a breakpoint at `add` and typed in **bt**, you will get a message similar to Listing 1.

Listing 2: A simple backtrace as produced by GDB

```
#0  add (i=1, j=2) at program.c:4
#1  0x000000000040077c in main () at program.c:13
```

Printing

At a breakpoint, watchpoint or when you are navigating anywhere in your code, you can print the values or addresses of variables. This can help to identify uninitialized variables, unexpected values as well as any errors in pointer arithmetic.

print <*variable_name*> Prints the value of the variable. For example, in the code above, **`print i`** will print 1.

print *<*pointer_name*>**/print** *<*address*> Dereferences a pointer or address and prints its content.

info locals Prints all variables that are in the local scope of the selected frame.

Now that you have a good idea of basic gdb functionality, time to put them to the test. We will walk through use cases, where gdb can and will save the day.

GDB use case 1: Segmentation fault

Consider the code snippet in Listing 3. When executing the code, you should get a segmentation fault. This is because you are trying to access values beyond the valid indices range of an array.

Listing 3: A short code snippet that exemplifies the use of uninitialized variables

```c
#include <stdio.h>

int main(int argc, char** argv) {
  /* Stepping out of the bounds of an array */
  int foo[1000];
  for (int i = 0; i <= 1050 ; i++){
    foo[i] = i;
  }
  return 0;
}
```

To debug the program with gdb, you can do the following:

```
gdb prog
    (gdb) run
```

This will result in the gdb displaying the following message

```
Program received signal SIGSEGV, Segmentation fault
```

Now, you can type `backtrace` and gdb will display where in the program you encountered the segmentation fault.

From here onwards, you can play with the breakpoints, watchpoints and navigation commands to figure out what is going on.

GDB use case 2: Uninitialized variables

Consider the code snippet in Listing 4. On running, you won't necessary get a segmentation fault but you will get a wrong answer because a variable `j` is uninitialized.

Now let us see what is going on.

```
gdb prog
    (gdb) break 8
    (gdb) run
```

The program pauses execution at the 8th line in the program. You can now print values of the variables.

```
print i
print j
print num
```

Listing 4: A simple example of an uninitialized variable.

```
#include <stdio.h>

int main(){
        int i, num, j; /* j is uninitialized */
        printf ("Enter the number: ");
        scanf ("%d", &num );

        for (i=1; i<num; i++)
                j=j*i;

        printf("The factorial of %d is %d\n",num,j);
}
```

You will realize that the value of j is not set. It will either be 0 or some random value. You have to set it to 1 for the code snippet to work.

TUI mode

TUI mode will split the terminal into multiple sections and display the source code, the assembly and the register values simultaneously as you debug.

You can run gdb in TUI mode by using the `-tui` flag. The command is as follows:

```
gdb -tui <filename>
```

Memory Debugging and Profiling with Valgrind

Documentation
`http://valgrind.org/docs/`

Quick Start Guide
`http://valgrind.org/docs/manual/quick-start.html`

While C is a useful and powerful language, dynamic memory management and pointer arithmetic make it very easy to get things wrong. Memory errors are some of the most common bugs that occur in low-level systems programs, and are, due to their nature, particularly difficult to debug. We have already talked about GDB and how it can be used to investigate program crashes and unexpected behavior. However, many memory bugs cannot be easily debugged by stepping through the code, because they let the program behave nondeterministically or do not cause the program to return wrong results for the given input.

Valgrind is a memory debugging tool that helps you find memory bugs in your programs. Internally, Valgrind is essentially a virtual machine that interprets the original program and monitors memory accesses to detect memory bugs such as the use of uninitialized memory, accesses to memory that has already been `free`'d, accesses to illegal memory addresses as well as memory leaks (i.e., memory that is allocated but never released).

Preparing your code

In order to effectively use Valgrind, code needs to be compiled with `-g` or `-g3` to include debugging information. It is also recommended to turn off optimizations (`-O0`) to ensure that line numbers in error messages are accurate. Once compiled, Valgrind can be invoked using `valgrind <executable>`.

Interpreting Valgrind output

After executing the process, Valgrind will generate a report of its analysis, that has a structure similar to Listing 5.

The number at the start of the line is the process id (in this example: `==29524==`) and can usually be ignored.

The report is divided into three main sections:

Error messages The first and main part of the report contains a list of all errors detected by Valgrind. The first line of an error message tells you the type of the error that occurred. In this example the program wrote to a memory address it was not allowed to write to. Below that line, you will find a stack trace that tells you where in the code the error occurred. The code address can usually be ignored, but can sometimes be useful to tracking down complicated bugs.

Heap summary At the end of the report you will find two summaries.

The Heap summary contains statistical values of the execution and most importantly a list of allocation backtraces for each memory leak Valgrind finds.

Leak summary Finally, a short leak summary shows you how and how much memory was lost during execution. *Definitely lost* indicates that no valid pointer to the respective allocated memory region is available when the process ends. *Indirectly lost* means that all pointers that are pointing to that memory area are either directly or indirectly lost.

```
==29524== Memcheck, a memory error detector
==29524== Copyright (C) 2002-2015, and GNU GPL'd, by Julian Seward et al.
==29524== Using Valgrind-3.11.0.SVN and LibVEX; rerun with -h for copyright info
==29524== Command: ./main
==29524==
==29524== Invalid write of size 4
==29524==    at 0x100000F32: main (main.c:12)
==29524==  Address 0x100802868 is 0 bytes after a block of size 40 alloc'd
==29524==    at 0x1000072E1: malloc (vg_replace_malloc.c:303)
==29524==    by 0x100000EF4: main (main.c:8)
==29524==
==29524==
==29524== HEAP SUMMARY:
==29524==     in use at exit: 34,429 bytes in 414 blocks
==29524==   total heap usage: 515 allocs, 101 frees, 41,445 bytes allocated
==29524==
==29524== LEAK SUMMARY:
==29524==    definitely lost: 0 bytes in 0 blocks
==29524==    indirectly lost: 0 bytes in 0 blocks
==29524==      possibly lost: 0 bytes in 0 blocks
==29524==    still reachable: 0 bytes in 0 blocks
==29524==         suppressed: 34,429 bytes in 414 blocks
==29524==
==29524== For counts of detected and suppressed errors, rerun with: -v
==29524== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
```

Listing 5: Example report generated by Valgrind.

Example 1: Detecting use of uninitialized memory

A common mistake in low-level programs is forgetting to initialize a variable. While this will normally cause you compiler to output a warning, the code will usually compile fine and result in nondeterministic behavior[2]. Uninitialized variables can be easily detected using Valgrind. Consider the code in Listing 6.

[2] Generally, it is recommended to set up your compiler to treat warnings as errors. That way, misuses of the memory system can often be detected at compile time.

```c
#include <stdio.h>

int main(int argc, char** argv) {
  int i; /* Declared but uninitialized */
  printf("%d\n", i);
  return 0;
}
```

Listing 6: A simple program that demonstrates a common memory bug: the use of uninitialized variables.

The code will generate a Valgrind report warning us of a function call that depends on an uninitialized value:

```
==30023==    Conditional jump or move depends on uninitialised value(s)
==30023==    at 0x1001DA851: __vfprintf (in /usr/lib/system/libsystem_c.dylib)
==30023==    by 0x10020280E: __v2printf (in /usr/lib/system/libsystem_c.dylib)
==30023==    by 0x100202AE0: __xvprintf (in /usr/lib/system/libsystem_c.dylib)
==30023==    by 0x1001D89D1: vfprintf_l (in /usr/lib/system/libsystem_c.dylib)
==30023==    by 0x1001D6837: printf (in /usr/lib/system/libsystem_c.dylib)
==30023==    by 0x100000F69: main (valgrind_uninit.c:9)
```

Listing 7: Valgrind reporting the usage of an uninitialized variable.

If you run valgrind with `--track-origins=yes`, additional information will be provided about where the unitialized values came from.

Example 2: Reading past end of array

Another common mistake is accessing elements past the end of an array. Unlike managed programming languages like Java that run in a virtual machine or managed environment, your C program will not throw an exception when such accesses occur, but instead might segfault or even continue running and returning the (possibly) correct result. Especially in the case that the program continues execution, such memory access violations become very difficult to debug.

Thankfully, Valgrind makes it easy to detect such problems and will return an error message similar to Listing 8.

```
==30130== Invalid write of size 4
==30130==    at 0x100000F32: main (valgrind_array.c:12)
==30130==  Address 0x100802868 is 0 bytes after a block of size 40 alloc'd
==30130==    at 0x1000072E1: malloc (vg_replace_malloc.c:303)
==30130==    by 0x100000EF4: main (valgrind_array.c:8)
```

Listing 8: An invalid memory write as reported by Valgrind.

Example 3: Detecting double frees

Valgrind automatically detects other improper use of dynamic memory. For instance, deallocating the same memory area twice (e.g., by calling `free()` twice on the same pointer), will cause Valgrind to report an error. Valgrind will also detect improperly chosen methods of freeing allocated memory, such as using `free` to release the memory allocated with `new`. The three basic allocation methods for dynamic memory that are available in C++, `malloc`, `new` and `new[]`, should only be matched with their respective deallocation function: `free`, `delete` and `delete[]`.

Example 4: Checking for memory leaks

To check for memory leaks during the execution of an executable, try running:

```
valgrind --leak-check=full <executable>
```

This will generate a complete report of memory leaks found in the program.

Note that the leak summary is also generated when executing Valgrind without the `--leak-check=full` option. However, this report will normally not contain all memory leaks that exist in the program. So, make sure you include the option when calling Valgrind.

Summary and limitations

Although Valgrind is a very useful program for finding memory errors in your program, there are several limitation that you should keep in mind. First, as errors in Valgrind are checked dynamically at runtime, its outcome heavily depends on the input parameters. If you run program that does not cause bad memory access behaviour, it will not report any errors. It is therefore recommended to run valgrind on a variety of inputs. Secondly, it is important to know that Valgrind's `Memcheck` is not able to detect all errors involving access of static and stack-allocated data. Therefore, passing a Valgrind run without errors does not guarantee that your program is free of memory bugs.

In addition, using Valgrind will consume more memory than the original program. For memory intensive programs this can cause many problems. It's also going to take significantly longer to run a program using Valgrind than it normally does. So, if you are running an already slow program, this is something you might want to keep in mind.