

**CIS453**  
Operating System Concepts  
Lab 1

# Introduction to Unix and System Dev Tools

Submitted by  
Seth Konynenbelt

Instructed by  
**Professor Dulimarta**

School of Computing  
GVSU  
Allendale, Michigan  
April 18, 2022

# 1 Results And Questions

- 1. (2 pts) Browse a number of manual pages of your choice from section 1, 2, and 3. Briefly describe the purpose of any two “headings”**

I chose to browse the manual pages of topics I am somewhat familiar with. Among these, some of the man pages included ‘git-submodule’, ‘fstatfs’, ‘syscalls’, ‘sysctl’, ‘printf’, ‘isgreater’, and ‘inifinite’. Upon looking through these topics, there is multiple headings involved within each man page. First, the synopsis portion of the man page is a very concise explanation of different ways the command. The synopsis portion gives the command, with options and arguments following the command. This section is important because it gives you a very concise way to to use the command. For example, a small part of the synopsis command is as follows for the git-submodule man page.

```
git submodule [--quiet] add [<options>] [--] <repository> [<path>]
git submodule [--quiet] status [--cached] [--recursive] [--] [<path>]
git submodule [--quiet] init [--] [<path>...]
git submodule [--quiet] deinit [-f|--force] (--all|[--] <path>...)
```

In the above example, one can see some different commands, as well as options and how you might use the command.

Another heading noticed was the “BUGS” section that was part of some man pages. This was important because it provides a place where some known issues and bugs are documented for the command. The “BUGS” heading can be valuable when developers might be dealing with a well-known issue with the call or subroutine. For example, the “BUGS” heading of the printf man page contains information about floating point numbers losing precision when translating to ASCII and back.

- 2. (2 pts) Describe the difference between the Unix user command time and the System call time()**

The Unix user command ‘time’ is quite different than the system call time(). The Unix user ‘time’ is used with a given command to display information about resources used by the command. For example, running the command ‘time cat hello.c’ returned information about the user, system, and CPU time resources. In this case, the total time was .003 second. Changing the command to ‘time vim hello.c’, making a file change in vim, then closing the file resulted in a much longer total time representing the time it took for the vim command to run.

On the other hand, the ‘time()’ system call returns the time as the number of seconds since Epoch time. This command is not used in the same way to capture system times and resources like the ‘time’ Unix user command is. The ‘time()’ system call could have applications where it is used to timestamp programs.

**3. (1 pt) What is the meaning (not its integer value) of the stream-based SEEK\_CUR macro?**

To find the meaning of the stream-based SEEK\_CUR macro, a keyword search on the man pages using ‘seek’ was completed. The man page ‘fseek’ was found, which the description that ‘fseek’ will reposition a stream. Upon reviewing the manpage, it looks like the library stdio.h is needed to reference those functions calls. To find the macro SEEK\_CUR, the following C code was written.

```
1 #include <stdio.h>
2
3 int main() {
4     /* empty main which does nothing */
5     return 0;
6 }
```

The C code was compiled with the following command ‘gcc -E -CC -dD main.c | less’. Upon navigating the output using the less command, the comment for SEEK\_CUR was “Seek from current position”. Upon further man page reading, this means that the SEEK\_CUR as the ‘whence’ parameter in fseek() to specify the offset is relative to the current position of the file position pointer.

**4. (1 pt) What is the command to list the contents of a directory in “long mode”, including hidden files?**

Knowing the command ‘ls’ is used to list the contents of a directory, the man page for ‘ls’ was used to determine how to list the contents of a directory in long mode, while including hidden files. The flag ‘-l’ is used to print in a long listing format, and the flag ‘-a’ is used to also include hidden files. The final command is ‘ls -a -l’, which can be shortened to be ‘ls -al’.

**5. (1 pt) What is the command syntax to make a directory readable/writable only by you?**

Knowing the command ‘chmod’ is used to handle file permissions, the man page for ‘chmod’ was used to determine the final command used is ‘chmod *u = rw, g = rwx, o = rwx* < the – directory – name >’. This can also be shortened (but less readable) using the same command ‘chmod 600 < the – directory – name >’.

**6. (2 pts) Perform the following steps: Create the above program and print debug screenshots as part of the lab report.**

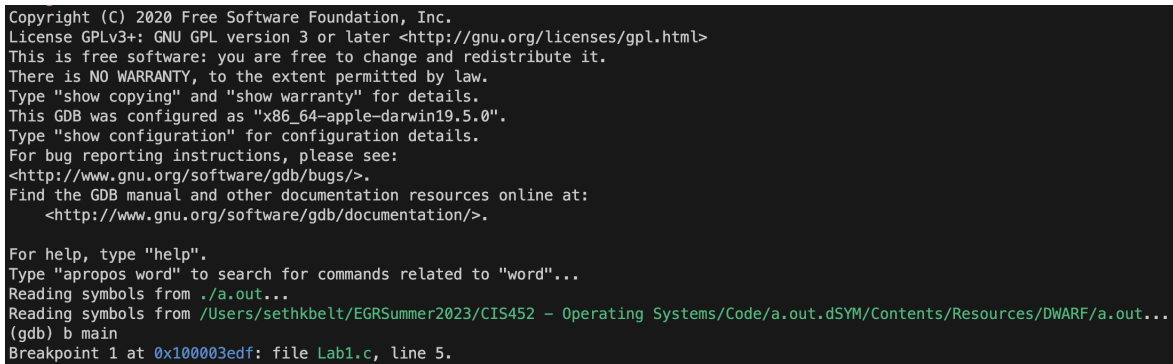
First, the below program was created.

```

1 #include <stdio.h>
2 #include <math.h>
3
4 int main() {
5     double num = 0.0;
6     printf ("Hello , world.\n");
7     num = pow (2, 28);
8     printf ("You are the %f person to write this program!\n", num);
9     return 0;
10 }

```

The program was compiled using the command ‘clang -Wall -g Lab1.c’. The program was first run without a debugger to verify the expected output, which was ‘Hello, world. You are the 268435456.000000 person to write this program!’. Then, the debugger was started and a breakpoint was set at main using the command ‘b main’ within the gdb debugger. This can be seen below in Figure 1.1.



```

Copyright (C) 2020 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-apple-darwin19.5.0".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from ./a.out...
Reading symbols from /Users/sethkbelt/EGRSummer2023/CIS452 - Operating Systems/Code/a.out.dSYM/Contents/Resources/DWARF/a.out...
(gdb) b main
Breakpoint 1 at 0x100003edf: file Lab1.c, line 5.

```

Figure 1.1: Screenshot of Debug Session Setting Main Breakpoint

Next, the program was run and stepped through. To exercise the debugger, the value of ‘num’ was printed before and after it was changed. The screenshot of this can be seen below in Figure 1.2.

```

There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-apple-darwin19.5.0".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from ./a.out...
Reading symbols from /Users/sethkbelt/EGRSummer2023/CIS452 - Operating Systems/Code/a.out.dSYM/Contents/Resources/DWARF/a.out...
(gdb) b main
Breakpoint 1 at 0x100003edf: file Lab1.c, line 5.
(gdb) run
Starting program: /Users/sethkbelt/EGRSummer2023/CIS452 - Operating Systems/Code/a.out
[New Thread 0xe03 of process 45168]
[New Thread 0x1703 of process 45168]
warning: unhandled dyld version (16)

Thread 2 hit Breakpoint 1, main () at Lab1.c:5
5      double num = 0.0;
(gdb) n
6      printf ("Hello, world.\n");
(gdb) p num
$1 = 0
(gdb) n
Hello, world.
7      num = pow (2, 28);
(gdb) n
8      printf ("You are the %f person to write this program!\n", num);
(gdb) p num
$2 = 268435456
(gdb)

```

Figure 1.2: Screenshot of Debug Session Showing Value of Num

## 7. (2 pts) Describe precisely (nature of problem, location) the memory leak(s) in the above program.

To begin, the code seen below was copied from the lab handout. The code contains memory leak and the command ‘valgrind –leak-check=full ./a.out’ was used to detect memory leaks.

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <string.h>
4
5 #define SIZE 16
6
7 int main()
8 {
9     char *data1, *data2;
10    int k;
11    do {
12        data1 = malloc(SIZE);
13        printf ("Please input your EOS username: ");
14        scanf ("%s", data1);
15        if (! strcmp (data1, "quit"))
16            break;
17        data2 = malloc(SIZE);
18        for (k = 0; k < SIZE; k++)
19            data2[k] = data1[k];
20        free (data1);
21        printf ("data2 :%s:\n", data2);

```

```

22     } while(1);
23     return 0;
24 }

```

The code above has errors because there are multiple memory leaks. The memory leaks occur in two places. First, on line 17, the ‘malloc()’ command is used for the pointer ‘data2’. Yet, ‘data2’ is never freed. The pointer keeps getting allocated data each loop, which causes the memory leak. The other memory leak found in this code is when the string “quit” is entered and the program returns out of the loop and then exits the program. This is a memory leak because the program breaks out of the while loop before data1 can be freed. To fix this, two function calls to free were needed. The line ‘free(data2)’ was added at the end of the while loop and ‘free(data1)’ was added before the program returns and exits. This code can be seen below.

```

1  /*****
2  * Title: CIS452 Lab1B
3  * Author: Seth Konynenbelt
4  * Created: January 11, 2024
5  * Description: Use valgrind to detect memory leaks
6  *               for dynamic memory debugging practice.
7  *****/
8  #include <stdio.h>
9  #include <stdlib.h>
10 #include <string.h>
11
12 #define SIZE 16
13
14 int main()
15 {
16     char *data1, *data2;
17     int k;
18     do {
19         data1 = malloc(SIZE);
20         printf ("Please input your EOS username: ");
21         scanf ("%s", data1);
22         if (! strcmp (data1, "quit"))
23             break;
24         data2 = malloc(SIZE);
25         for (k = 0; k < SIZE; k++)
26             data2[k] = data1[k];
27         printf ("data2 :%s:\n", data2);
28         // free data1 and data 2 pointer, moved here for readability
29         free(data1);
30         free(data2);
31     } while(1);
32     // free data1 before we exit
33     // we know only data1 malloced if we are here
34     free(data1);
35     return 0;
36 }

```

To verify this code worked properly, valgrind was run again on the revised code. Valgrind detected no memory leaks, as seen below in Figure 1.3.

```
==6597==  
==6597== HEAP SUMMARY:  
==6597==   in use at exit: 0 bytes in 0 blocks  
==6597==   total heap usage: 7 allocs, 7 frees, 2,128 bytes allocated  
==6597==  
==6597== All heap blocks were freed -- no leaks are possible  
==6597==  
==6597== For lists of detected and suppressed errors, rerun with: -s  
==6597== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Figure 1.3: Valgrind Output

**8. (1 pt) How many times is the write() system call invoked when running Sample 2?**

Upon running the original code containing memory leaks using strace, a hefty amount of output was created. Reading the manpage for strace to filter out results to just syscalls, the command ‘strace -e write ./a.out’ was used. In the figure below, before entering quit, I entered 4 inputs 1,2,3,4 as examples for my EOS login. Every time I input, the write() system call was evoked 2 times. The formula for how many times write() is called can be described as ‘ $2 * n$ ’, where  $n$  is how many times a user makes an entry.

**9. (1 pt) Use the example the source code and experiment to answer the question: what is the primary C library subroutine that causes the write() system call to be invoked by Sample 2?**

In the original code, the primary C library subroutine that causes the write() system call is printf(). Printf() is run twice each for loop and evokes write(). This also makes sense since printf is the C subroutine to handle outputting to the standard output buffer.