Digital Design Verification

FYP/Internship Program

Lab # 05

Memory Management & Pointers

Release: 1.1

Release Date: 12-August-2024

NUST Chip Design Centre (NCDC), Islamabad, Pakistan

Revision History

Revision Number	Revision Date	Revision By	Nature of Revision	Approved By
1.0	26/02/2024	Hira Sohail, Muhammad Bilal	Complete manual	-
1.1	13/08/2024	Hira Sohail	Revision in manual	Deputy PD

Contents

Objective

The purpose of this lab is to:

- Learn creating Makefile and performing tasks using it.
- Resolve bugs using Valgrind
- Resolve segmentation faults using cgdb and valgrind
- Perform tasks using pointers

Tools

- Linux
- GCC
- Valgrind
- GDB

Introduction

Make

As you have observed, compiling C programs in the terminal is a tedious and time-consuming operation. While this is doable for simple C programs, for larger and more complex programs with dozens of files and dependencies, this gets rather unwieldy quickly. Additionally, if we are editing code in one file in a large code base, we would like to rebuild the minimal number of files rather than the entire project whenever we want to run our code. To solve these issues, we can use a program called make.

The make utility automatically determines which pieces of a large program need to be recompiled, and issues commands to recompile them. To prepare to use make, you must write a file called the makefile that describes the relationships among files in your program and provides commands for updating each file. In a program, typically, the executable file is updated from object files, which are in turn made by compiling source files.
Once a suitable makefile exists, each time you change some source files, this simple shell command:

```
make
```

suffices to perform all necessary recompilations. The make program uses the makefile data base and the last-modification times of the files to decide which of the files need to be updated. For each of those files, it issues the recipes recorded in the data base.

Structure of a Makefile

A simple makefile consists of “rules” with the following shape:

```
target … : prerequisites …
        recipe
        …


        …
```

A target is usually the name of a file that is generated by a program; examples of targets are executable or object files.

A prerequisite is a file that is used as input to create the target. A target often depends on several files.

A recipe is an action that make carries out. A recipe may have more than one command, either on the same line or each on its own line. Please note: you need to put a tab character at the beginning of every recipe line! This is an obscurity that catches the unwary.

A rule, then, explains how and when to remake certain files which are the targets of the particular rule. Make carries out the recipe on the prerequisites to create or update the target. A rule can also explain how and when to carry out an action.

Example

We have created a Makefile that compiles the code of linked_list file attached with the manual. Read over the code in the Makefile.

- There are two variables defined at the top of the file. To access these variables, we use this syntax: $(var_name).
- The first target all specifies the default goal, meaning which targets make will consider when no target is specified on the command line, all currently specifies linked_list. If you want to add more targets to default goal, you should add them to this line

No target specified:

```
make
```

Target specified:

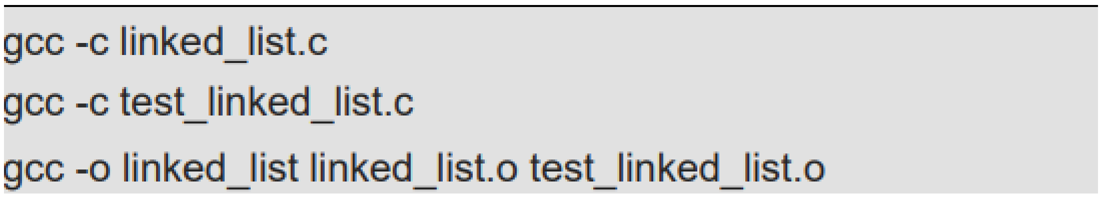
```
make linked_list
```

- We tried to make this file as simple as possible to make it easier to understand. There are more complex things that you can do with a Makefile. For instance, make offers automatic variables to shorten recipes and prerequisite lists.

1. Run the following command to compile the code and generate an executable called linked_list.

```
make
```

When you run make, you can see that it echoes the list of commands that are executed. Your output should look something like this:



```
gcc -c linked_list.c
gcc -c test_linked_list.c
gcc -o linked_list linked_list.o test_linked_list.o
```

2. Run make again, it should output something like this:

```
make: Nothing to be done ‘all
```

None of the files that linked_list depends on were updated since the last time you invoked make, so make did not do anything.

3. The Makefile also contains a target for deleting the generated executable and object files. To execute this rule, use the following command.

```
make clean
```

Again, make echoed the commands that it ran. It should look something like this:

```
rm linked_list linked_list.o test_linked_list.o
```

4. Run make now, it will recompile everything.

5. Make an edit to linked_list.c and then run make. You should see that linked_list.c is recompiled, but test_linked_list.c is not recompiled.

TASK 1 – Valgrind

Even with a debugger, we might not be able to catch all bugs. Some bugs are what we refer to as “bohrbugs”, meaning they manifest reliably under a well-defined, but possibly unknown, set of conditions. Other bugs are what we call “heisenbugs”, and instead of being determinant, they’re known to disappear or alter their behavior when one attempts to study them. We can detect the first kind with debuggers, but the second kind may slip under our radar because they’re (at least in C) often due to mis-managed memory. Remember that unlike other programming languages, C requires you (the programmer) to manually manage your memory. We can use a tool called Valgrind to help catch to help catch “heisenbugs” and "bohrbugs". Valgrind is a program which emulates your CPU and tracks your memory accesses. This slows down the process you’re running (which is why we don’t, for example, always run all executables inside Valgrind) but also can expose bugs that may only display visible incorrect behavior under a unique set of circumstances. Please install valgrind on your machine using the following commands:

For Ubuntu:

```
sudo apt update
```

```
sudo apt install valgrind
```

For Redhat:

```
sudo yum install valgrind
```

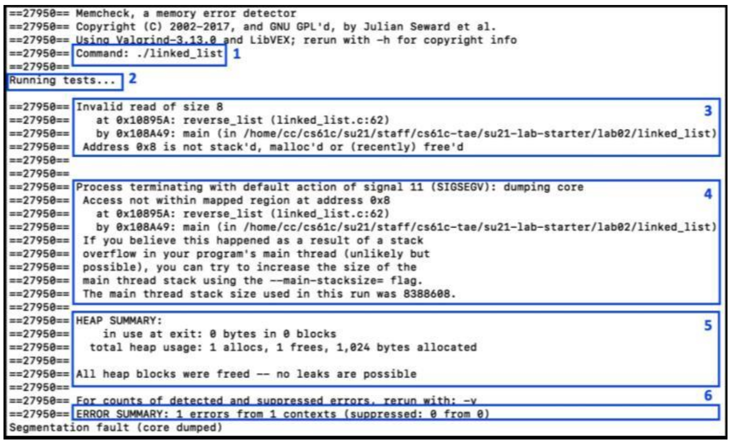
Using Valgrind to find segfaults

You can also use Valgrind to find segfaults.

1. Edit the Makefile to include the -g flag in CFLAGS to provide debugging information to Valgrind.
2. Compile linked_list.c and test_linked_list.c by executing make.
3. Run valgrind on the executable using the following command:

```
valgrind ./linked_list
```

By default, memcheck is the tool that is run when you invoke Valgrind. The documentation on Valgrind's memcheck is very useful, as it provides examples of the most common error messages, what they mean, and some optional arguments you can use to help debug them. Your output should look something like this. There is a lot of information here, so let's parse through it together.



```
==27950== Memcheck, a memory error detector
==27950== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==27950== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright info
==27950== Command: ./linked_list 1
==27950==
Running tests... 2

==27950== Invalid read of size 8                                                               3
==27950==    at 0x10895A: reverse_list (linked_list.c:62)
==27950==    by 0x108A49: main (in /home/cc/cs61c/su21/staff/cs61c-tae/su21-lab-starter/lab02/linked_list)
==27950==  Address 0x8 is not stack'd, malloc'd or (recently) free'd
==27950==
==27950==
==27950== Process terminating with default action of signal 11 (SIGSEGV): dumping core
==27950==  Access not within mapped region at address 0x8                                      4
==27950==    at 0x10895A: reverse_list (linked_list.c:62)
==27950==    by 0x108A49: main (in /home/cc/cs61c/su21/staff/cs61c-tae/su21-lab-starter/lab02/linked_list)
==27950==  If you believe this happened as a result of a stack
==27950==  overflow in your program's main thread (unlikely but
==27950==  possible), you can try to increase the size of the
==27950==  main thread stack using the --main-stacksize= flag.
==27950==  The main thread stack size used in this run was 8388608.
==27950==
==27950== HEAP SUMMARY:                                                                        5
==27950==     in use at exit: 0 bytes in 0 blocks
==27950==   total heap usage: 1 allocs, 1 frees, 1,024 bytes allocated
==27950==
==27950== All heap blocks were freed -- no leaks are possible
==27950==
==27950== For counts of detected and suppressed errors, rerun with: -v                         6
==27950== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
Segmentation fault (core dumped)
```

Box 1. This shows us the command that we are running through Valgrind.

Box 2. This is a print statement from our program.

Box 3. We are reading 8 bytes from an invalid memory address on linked_list.c line 62.

Box 4. Our program received a segfault by accessing invalid memory on linked_list.c line 62.

Box 5. There were no memory leaks at the time that the program exited.

Box 6. We encountered 1 error.

We will Fix this error in coming task of segmentation fault.

Using Valgrind to detect memory leaks

1. Let's cause a memory leak in test_linked_list.c. Comment out the two lines that call free_list.

2. Run make to compile your code.

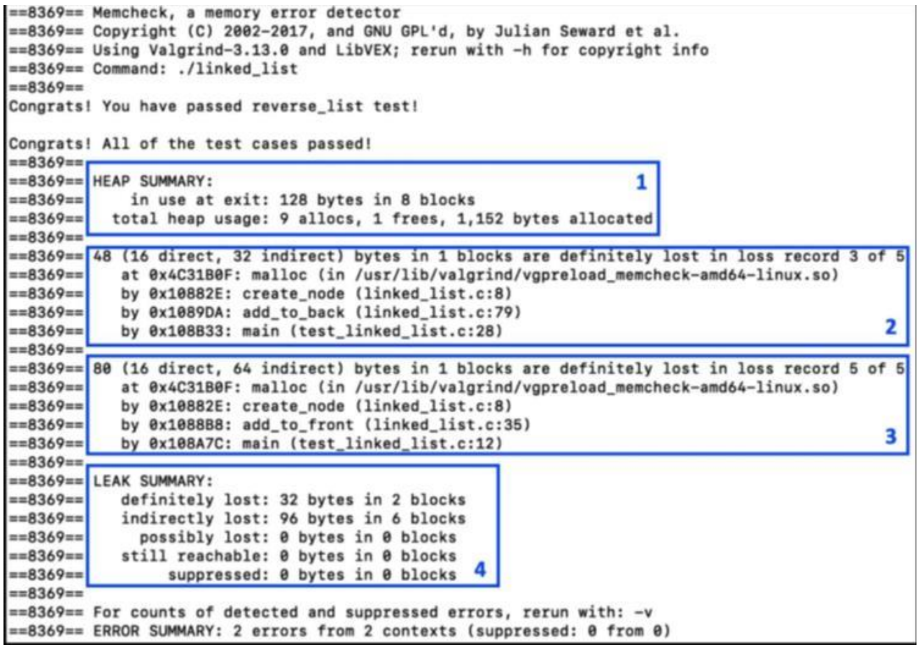
3. Run valgrind

```
valgrind ./linked_list
```

4. We can see that our program is still producing the correct result based on the printed messages "Congrats..."; however, we are now experiencing memory leaks. Valgrind tells us to "Rerun with --leak-check=full to see details of leaked memory", so let's do that

```
Valgrind –leak-check=full ./linked_list
```

Your output should look something like this. There is a lot of information here, so let's parse through it together.



```
==8369== Memcheck, a memory error detector
==8369== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==8369== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright info
==8369== Command: ./linked_list
==8369==
Congrats! You have passed reverse_list test!

Congrats! All of the test cases passed!
==8369==
==8369== HEAP SUMMARY:                                                  1
==8369==     in use at exit: 128 bytes in 8 blocks
==8369==   total heap usage: 9 allocs, 1 frees, 1,152 bytes allocated
==8369==
==8369== 48 (16 direct, 32 indirect) bytes in 1 blocks are definitely lost in loss record 3 of 5
==8369==    at 0x4C31B0F: malloc (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==8369==    by 0x10882E: create_node (linked_list.c:8)
==8369==    by 0x1089DA: add_to_back (linked_list.c:79)
==8369==    by 0x108B33: main (test_linked_list.c:28)                              2
==8369==
==8369== 80 (16 direct, 64 indirect) bytes in 1 blocks are definitely lost in loss record 5 of 5
==8369==    at 0x4C31B0F: malloc (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==8369==    by 0x10882E: create_node (linked_list.c:8)
==8369==    by 0x1088B8: add_to_front (linked_list.c:35)
==8369==    by 0x108A7C: main (test_linked_list.c:12)                              3
==8369==
==8369== LEAK SUMMARY:
==8369==    definitely lost: 32 bytes in 2 blocks
==8369==    indirectly lost: 96 bytes in 6 blocks
==8369==      possibly lost: 0 bytes in 0 blocks
==8369==    still reachable: 0 bytes in 0 blocks
==8369==         suppressed: 0 bytes in 0 blocks  4
==8369==
==8369== For counts of detected and suppressed errors, rerun with: -v
==8369== ERROR SUMMARY: 2 errors from 2 contexts (suppressed: 0 from 0)
```

Box 1. Summary of heap usage. There were 128 bytes allocated in 8 different blocks the heap at the time of exit.

Box 2 and 3. Stack traces showing where the unfreed blocks were allocated.

Direct blocks are those which are root nodes (blocks of memory that the programmer has direct access to, ex stack/global pointer to the heap).

Indirect blocks are those which are not root nodes (ex a pointer inside of a struct).

Box 4. Summary of leak. You can use the stack trace to see where the unfreed blocks were allocated.

TASK 2 – Segmentation Faults

In this section, we will be debugging the recursive implementation of reversing a linked list.

1. Read over the code in /linked_list.c file attached in LMS.
2. In this step, you will learn how to find segfaults using cgdb. In lab1, you used cgdb to debug your code. This section will resolve bug(s) along the way. Make sure to fix the bug(s) in the code before moving on.
    1. Compile and run your code.
    2. We can see that there is a segfault, but we do not know where it is occurring.
    3. Let's open up the code in cgdb (make sure that you compiled it with the -g flag).
    4. Recall the run command will run your code until something stops it (like a segfault or a breakpoint). Execute the run command.
    5. The command window shows us that the program encountered a segfault ("Program received signal SIGSEGV") at line 62. Now we know the line where the segfault occurred.
    6. Another command that can be useful when you are debugging segfaults is backtrace. This will allow you to see the call stack at the time of the crash. Execute the following command to see the call stack

```
backtrace
```

or

```
bt
```

In this case, we can see that reverse_list was called from main. We already knew this, so backtrace was not helpful in this case, but it can be helpful in the future if you have longer call stacks.

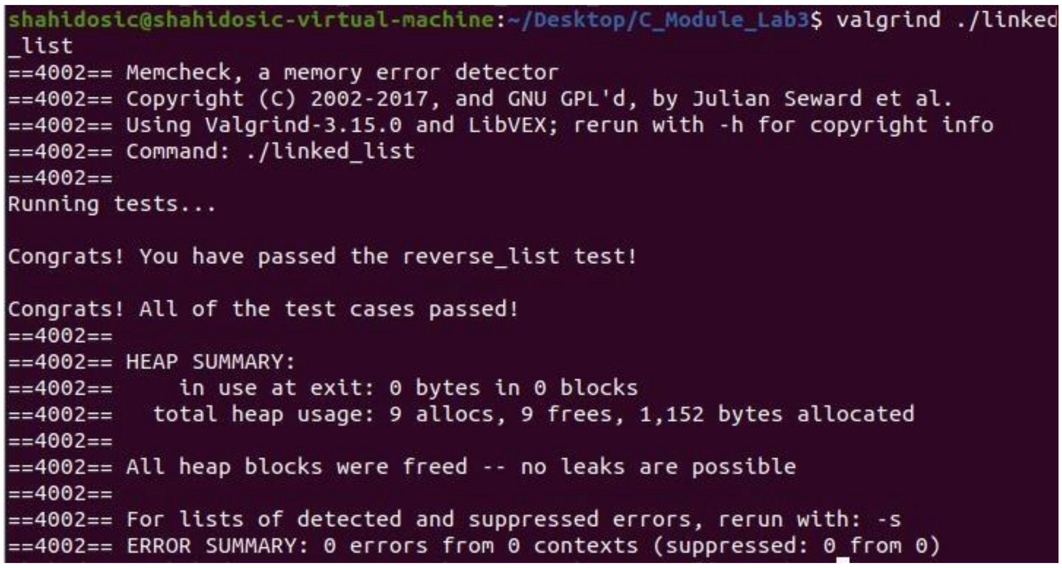
7. Let's examine the state of the program when the segfault occurred by printing some variables. Print out head.
8. Remember that head is a double pointer, so we expect to see an address when we print it out. head looks fine. Let's print out the dereferenced head (*head).
9. It looks like *head is NULL. If you look at the line of code that the program is segfaulting on, you can see that we are trying to access (*head)->next. We know that trying to access a member of a NULL struct will result in a segfault, so this is our problem.
10. What does it mean for *head to be NULL? It means that the list is empty. How do we reverse an empty list? The reverse of an empty list is just an empty list, so we do not need to modify the list. At the top of the function, we can add a check to see if the list if empty and return if so. Modify the first line of the function to resolve the error:

```
if (head == NULL) {
```

11. Compile and run your code. It should pass the reverse list tests now.

3. Debug add_to_back on your own using the commands you just learned. Make sure that you are using cgdb so that you can get practice using it. This will help you with your projects.

Your console should look like this:



```
shahidosic@shahidosic-virtual-machine:~/Desktop/C_Module_Lab3$ valgrind ./linked
_list
==4002== Memcheck, a memory error detector
==4002== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==4002== Using Valgrind-3.15.0 and LibVEX; rerun with -h for copyright info
==4002== Command: ./linked_list
==4002==
Running tests...

Congrats! You have passed the reverse_list test!

Congrats! All of the test cases passed!
==4002==
==4002== HEAP SUMMARY:
==4002==     in use at exit: 0 bytes in 0 blocks
==4002==   total heap usage: 9 allocs, 9 frees, 1,152 bytes allocated
==4002==
==4002== All heap blocks were freed -- no leaks are possible
==4002==
==4002== For lists of detected and suppressed errors, rerun with: -s
==4002== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

TASK 3 - Memory Management

This task uses vector.h, test_vector.c, and vector.c (attached on the LMS), where we provide you with a framework for implementing a variable-length array. This exercise is designed to help familiarize you with C structs and memory management in C.

1. For checkoff, be prepared to explain why bad_vector_new() and also_bad_vector_new() are bad.

Hint: One of these functions will actually run correctly (assuming correctly modified vector_new, vector_set, etc.) but there may be other problems.

2. Fill in the functions vector_new(), vector_get(), vector_delete(), and vector_set() in vector.c so that our test code test_vector.c runs without any memory management errors.

Comments in the code describe how the functions should work. Look at the functions we've filled in to see how the data structures should be used. For consistency, it is assumed that all entries in the vector are 0 unless set by the user. Keep this in mind as malloc() does not zero out the memory it allocates.

Test your implementation of vector_new(), vector_get(), vector_delete(), and vector_set() for both correctness and memory management (details below).

```
# 1) to check correctness

make vector

./vector
# 2) to check memory management using Valgrind:

valgrind ./vector
```

- show the output of make vector and valgrind ./vector

Submission:

Please submit all the .c files of all the tasks along with the screenshots of outputs on LMS in a proper report. Report contains marks.