Lab 2: Advanced C, Valgrind

Deadline: Friday, September 17, 04:00:00 PM PT

Setup

In your `labs` directory, pull the files for this lab with:

```
$ git pull starter main
```

If you get an error like the following:

```
fatal: 'starter' does not appear to be a git repository
fatal: Could not read from remote repository.
```

make sure to set the starter remote as follows:

```
$ git remote add starter https://github.com/61c-teach/fa21-lab-starter.git
```

and run the original command again.

Exercise 1: `make`

As you saw in Lab 1, 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 amount 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`.

What is `make`?

Here is an excerpt from the GNU manual for `make` to explain how this works:

> 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`

Excerpt:

> 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 target can also be the name of an action to carry out, such as 'clean' (see Phony Targets).
>
> 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 from `Lab1/exercise3`. 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
  ```
- There are more complex things that you can do with a `Makefile`. For instance, `make` offers automatic variables to shorten recipies and prerequisite lists. If you want to learn more about this, you can find more info in the `make` docs (specifically look at `$@` and `$^`).

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 for '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.

If you would like to see another example of a `Makefile`, you can find one [in the `make` docs]

Action Item

Exercises 3 and 4 of this lab require running the following commands to compile their code:

Exercise 3

```
$ gcc -o bit_ops bit_ops.c test_bit_ops.c
```

Exercise 4

```
$ gcc -o vector vector.c test_vector.c
```

Update `Makefile` so that it compiles the code in Exercises 3 and 4.

- Remember to update the `all` and `clean` targets

Exercise 2: 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.

Using Valgrind to find segfaults

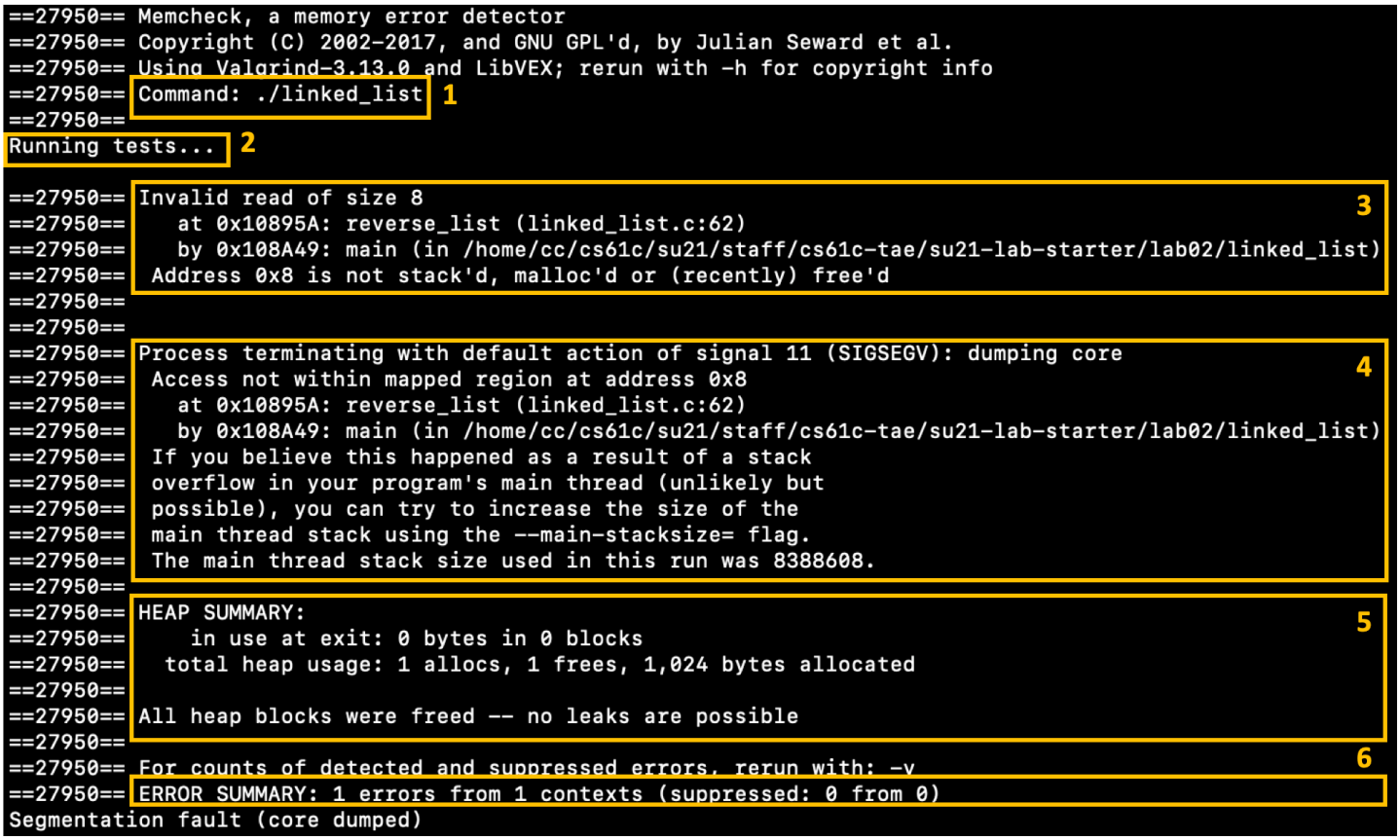
In Lab 1, you learned how to find segfaults using cgdb. You can also use Valgring to find segfaults.

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 valgrid 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.



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.

4. Copy over your solution from `Lab01/exercise3` to fix this error. Run your code through Valgrind again, and you should see that there are no errors reported by valgrind.

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 find more info about this section in the Valgrind docs

You can use the stack trace to see where the unfreed blocks were allocated. Hopefully this example will help you understand Valgrind messages when you are completing your projects!

Please note that the autograder does not verify your valgrind output. This will be manually checked by your TA/tutor/AI during checkoff

---

Exercise 3: Bit Operations

For this exercise, you will complete `bit_ops.c` by implementing the bit manipulation functions `get_bit`, `set_bit`, and `flip_bit` (shown below). You may **ONLY** use bitwise operations such as and (`&`), or (`|`), xor (`^`), not (`~`), left shifts (`<<`), and right shifts (`>>`). You may not use any for/while loops or conditional statements. You also may not use modulo (`%`), division, addition, subtraction, or multiplication for this question.

```
/* Returns the Nth bit of X. Assumes 0 <= N <= 31. */
unsigned get_bit(unsigned x, unsigned n) {
    /* YOUR CODE HERE */
    return -1; /* UPDATE WITH THE CORRECT RETURN VALUE*/
}

/* Set the nth bit of the value of x to v. Assumes 0 <= N <= 31, and V is 0 or 1 */
void set_bit(unsigned *x, unsigned n, unsigned v) {
    /* YOUR CODE HERE */
}

/* Flips the Nth bit in X. Assumes 0 <= N <= 31.*/
void flip_bit(unsigned *x, unsigned n) {
    /* YOUR CODE HERE */
}
```

ACTION ITEM

Finish implementing `get_bit`, `set_bit`, and `flip_bit`.

Once you complete these functions, you can compile and run your code using the following commands:

```
$ make bit_ops
$ ./bit_ops
```

This will print out the results of the tests.

Be ready to show this code to your TA for checkoff. They will be checking to make sure that you did not use any of the forbidden operations.

Exercise 4: Memory Management

This exercise uses `vector.h`, `test_vector.c`, and `vector.c`, 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.

Action Item

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
```

Any number of suppressed errors is fine; they do not affect us.

Feel free to also use CGDB to debug your code.

Exercise 5

Please fill out this short survey about your experience with the lab. Your responses will be used to improve the lab in the future. The survey will be collecting your email to verify that you have submitted it, but your responses will be anonymized when the data is analyzed. Thank you!

Submission

Save, commit, and push your work, then submit to the **Lab 2** assignment on Gradescope.

Checkoff

When you file your ticket, make sure that you include your partner's name in the description. If you do not have a partner, we will check you off with another student who does not have a partner.

- Show your TA your code from Exercise 3
- Explain to your TA/tutor/AI why `bad_vector_new()` and `also_bad_vector_new()` are bad
- Also, show your TA/tutor/AI the output of `make vector` and `valgrind ./vector`
