Lab 3

Computer Architecture I ShanghaiTech University
Lab 2 Lab 3 Lab 4

Goals

- Perform specific bit manipulations through compositions of bit operations.
- Introduced to the C debugger and gain practical experience using gdb to debug C programs.
- Identify potential issues with dynamic memory management.

Exercises

Download the files for Lab 3 first.

Exercise 1: 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.

```
/* Return the nth bit of x.
   Assume 0 <= n <= 31 */
unsigned get_bit (unsigned x, unsigned n);

/* Set the nth bit of the value of x to v.
   Assume 0 <= n <= 31, and v is 0 or 1 */
void set_bit (unsigned *x, unsigned n, unsigned v);

/* Flip the nth bit of the value of x.
   Assume 0 <= n <= 31 */
void flip_bit (unsigned *x, unsigned n);
```

ACTION ITEM: Finish implementing `get_bit`, `set_bit`, and `flip_bit`.

Once you complete these functions, you can compile your code using:

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

Check-off

> Show your TA the output of running `bit_ops`.

Exercise 2: Catch those bugs!

A **debugger**, as the name suggests, is a program which is designed specifically to help you find bugs, or logical errors and mistakes in your code (side note: if you want to know why

errors are called bugs, look here). Different debuggers have different features, but it is common for all debuggers to be able to do the following things:

1. Set a breakpoint in your program. A breakpoint is a specific line in your code where you would like to stop execution of the program so you can take a look at what's going on nearby.
2. Step line-by-line through the program. Code only ever executes line by line, but it happens too quickly for us to figure out which lines cause mistakes. Being able to step line-by-line through your code allows you to hone in on exactly what is causing a bug in your program.

For this exercise, you will find the GDB reference card useful. GDB stands for "GNU De-Bugger." :) Compile `hello.c` with the `-g` flag:

```
$ gcc -g -o hello hello.c
```

This causes gcc to store information in the executable program for `gdb` to make sense of it. Now start our debugger, (c)gdb:

```
$ cgdb hello
```

Notice what this command does! You are running the program cgdb on the executable file hello generated by gcc. Don't try running cgdb on the source code in hello.c! It won't know what to do. If cgdb does not work, you can also use gdb to complete the following exercises (start gdb with `gdb hello`).

ACTION ITEM: step through the whole program by doing the following:

1. setting a breakpoint at main
2. using gdb's run command
3. using gdb's single-step command

Type help from within gdb to find out the commands to do these things, or use the reference card.

Look here if you see an error message like printf.c: No such file or directory. You probably stepped into a printf function! If you keep stepping, you'll feel like you're going nowhere! CGDB is complaining because you don't have the actual file where printf is defined. This is pretty annoying. To free yourself from this black hole, use the command finish to run the program until the current frame returns (in this case, until printf is finished). And **NEXT** time, use next to skip over the line which used printf.

Note: cgdb vs gdb In this exercise, we use cgdb to debug our programs. cgdb is identical to gdb, except it provides some extra nice features that make it more pleasant to use in practice. All of the commands on the reference sheet work in gdb. In cgdb, you can press ESC to go to the code window (top) and i to return to the command window (bottom) — similar to vim. The bottom command window is where you'll enter your gdb commands.

ACTION ITEM: Learn MORE gdb commands Learning these commands will prove useful for the rest of this lab, and your C programming career in general. Create a text file

containing answers to the following questions (or write them down on a piece of paper, or just memorize them if you think you want to become a GDB pro).

1. How do you **pass command line arguments** to a program when using gdb?
2. How do you **set a breakpoint** which only occurs when a **set of conditions is true** (e.g. when certain variables are a certain value)?
3. How do you **execute the next line of C code** in the program after stopping at a breakpoint?
4. If the next line of code is a function call, you'll execute the whole function call at once if you use your answer to #3. (If not, consider a different command for #3!) How do you tell GDB that you **want to debug the code inside the function** instead? (If you changed your answer to #3, then that answer is most likely now applicable here.)
5. How do you **resume the program after stopping** at a breakpoint?
6. How can you **see the value of a variable** (or even an expression like 1+2) in gdb?
7. How do you configure gdb so it **prints the value of a variable after every step**?
8. How do you **print a list of all variables and their values** in the current function?
9. How do you **exit** out of gdb?

Check-off

> Show your TA that you are able to run through the above steps and provide answers to the questions.

Exercise 3: Memory Management

This exercise uses `vector.h`, `vector-test.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: Explain why `bad_vector_new()` and `also_bad_vector_new()` are bad and fill in the functions `vector_new()`, `vector_get()`, `vector_delete()`, and `vector_set()` in `vector.c` so that our test code `vector-test.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.

For explaining why the two bad functions are incorrect, keep in mind that one of these functions will actually run correctly (assuming correctly modified `vector_new`, `vector_set`, etc.) but there may be other problems. **Hint**: think about memory usage.

ACTION ITEM: 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-test
$ ./vector-test

# 2) to check memory management using Valgrind:
$ make vector-memcheck
```

All the `vector-memcheck` rule does is run the following valgrind command on our executable. Explain to yourself what each of the flags mean.

```
$ valgrind --tool=memcheck --leak-check=full --track-origins=yes [OS SPECIFIC ARGS] ./<executable>
```

The last line in the valgrind output is the line that will indicate at a glance if things have gone wrong. Here’ s a sample output from a buggy program:

```
==47132== ERROR SUMMARY: 1200039 errors from 24 contexts (suppressed: 18 from 18)
```

If your program has errors, you can scroll up in the command line output to view details for each one. For our purposes, you can safely ignore all output that refers to suppressed errors. In a leak-free program, your output will look like this:

```
==44144== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 18 from 18)
```

Again, any number of suppressed errors is fine; they do not affect us.

Feel free to also use CGDB or add `printf` statements to `vector.c` and `vector-test.c` to debug your code.

Check-off

> Explain to your TA why `bad_vector_new()` and `also_bad_vector_new()` are bad. Also, show your TA the output of running the program as well as the output of `make vector-memcheck`.

Schwertfeger, Sören <soerensch AT shanghaitech.edu.cn>
Chundong Wang <wangchd AT shanghaitech.edu.cn>
Modeled after UC Berkeley's CS61C.
Last modified: 2020-03-12