Lab 2

Computer Architecture I @ ShanghaiTech University

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 2 first.

Exercise 1: git init; git add; git commit

After extracting the tar file with `tar` command, initialize a git repository in the lab2 directory and make your first commit.

Exercise 2: queue<double>

First read `queue.h` and `queue.c`
The four functions in `queue.h` are declared as follows:

- `Queue *queue_create(void);`
- `void push(Queue *queue, double element);`
- `double back(Queue *queue);`
- `void queue_free(Queue *queue);`

Make sure you understand what every line of code is doing before moving on.

> Tell the TA what `malloc()`, `realloc()`, and `free()` do.

This exercise will give some subtasks, You should do a git commit after each subtask to track your changes, and commit only the source code (no executables or .o files).
You must compile your code with `Makefile` we've provided.

1. Consider the cases where `malloc()` and `realloc()` fails, do necessary changes to make code work under these circumstances.
2. The code never checks the `Queue *queue` passed to it is valid, modify the code to add a null check at the beginning of each function.
3. In function `back()`, it is possible that the user may access an empty queue, you should handle this situation.
4. Some implementation suggests `void another_queue_free(Queue **queue)`, add this to your code. Explain why this could be beneficial.

> Show your TA the git commits after each subtask and make necessary explanations.

Exercise 3: Catch those bugs!

Installation

In Ubuntu, we can take advantage of package managers: `sudo apt install cgdb` or `sudo apt install gdb`.

Describe

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:

- 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 that you can take a look at what's going on nearby.
- 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." :) Use the code you've written in the previous exercise to try the debugger.

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

```
$ cgdb ./lab2
```

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! GDB 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.

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?

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

Exercise 4: Memory Management

Valgrind is an open-source framework built for dynamic analysis. In this course, we will use a very popular tool, memcheck, to detect any possible memory-related issue.

Installation

In linux, we can take advantage of package managers: `sudo apt install valgrind` (in Ubuntu) for minimal installation.

Example usages

When there are no memleaks

```
// File name: safe.c
#include <stdio.h>

int main() {
  int a = 42;
  printf("%d\n", a);
  return 0;
}
```

We compile with `gcc -Wpedantic -Wall -Wextra -Wvla -Werror -std=c11 -g -O0 safe.c -o safe` to make sure there are no warnings, which means that compiler (`gcc`) is unable to find any potential issues.
We can easily point out that this program with output `42` is memory safe.

```
$ valgrind --tool=memcheck --leak-check=full --track-origins=yes ./safe
```

Here's what `Valgrind` gives us, which indicate that no memleaks are detected.

```
==67518== HEAP SUMMARY:
==67518==     in use at exit: 0 bytes in 0 blocks
==67518==   total heap usage: 1 allocs, 1 frees, 1,024 bytes allocated
==67518==
==67518== All heap blocks were freed -- no leaks are possible
==67518==
==67518== For lists of detected and suppressed errors, rerun with: -s
==67518== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

When there are memleaks:

```
// File name: memleak.c
#include <stdio.h>
#include <stdlib.h>

int main() {
  int a = 42;
  char *normal = malloc(135);
  char *leak = malloc(246);
  leak[0] = 'q';  // Avoid unused variable warning
  printf("%d\n", a);
  free(normal);
```

```
  return 0;
}
```

In this example, we allocated memory from heap for variable `leak` , but the call to `free()` for the variable is missing, causing us to fail Gradescope test :(
We can compile using the following command.

```
$ gcc -Wpedantic -Wall -Wextra -Wvla -Werror -std=c11 -g -O0 memleak.c -o memleak
```

Now we harness the power of `Valgrind` to detect the issue:

```
$ valgrind --tool=memcheck --leak-check=full --track-origins=yes ./memleak
```

```
==67769== HEAP SUMMARY:
==67769==     in use at exit: 246 bytes in 1 blocks
==67769==   total heap usage: 3 allocs, 2 frees, 1,405 bytes allocated
==67769==
==67769== 246 bytes in 1 blocks are definitely lost in loss record 1 of 1
==67769==    at 0x483B7F3: malloc (in /usr/lib/x86_64-linux-gnu/valgrind/vgpreload_memcheck-amd64-linux.so)
==67769==    by 0x1091B3: main (memleak.c:8)
==67769==
==67769== LEAK SUMMARY:
==67769==    definitely lost: 246 bytes in 1 blocks
==67769==    indirectly lost: 0 bytes in 0 blocks
==67769==      possibly lost: 0 bytes in 0 blocks
==67769==    still reachable: 0 bytes in 0 blocks
==67769==         suppressed: 0 bytes in 0 blocks
==67769==
==67769== For lists of detected and suppressed errors, rerun with: -s
==67769== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
```

We can tell from above that:

1. When we exit the program, there are still 246 bytes on the heap that belongs to us.
2. The missing 246 bytes were allocated in `main` with `malloc()`.
3. ...

By adding more options/flags of `Valgrind`, there are absolutely more messages to reveal. **What's more:**

```
// File name: overflow.c
#include <stdio.h>

int main() {
  int a[5] = {13, 24, 35, 46, 57};
  printf("%d\n", a[5]);
  return 0;
}
```

With `Valgrind`, we can find those uninitialized visits effortlessly (only a fraction of the message is shown here):

```
==67943== Conditional jump or move depends on uninitialised value(s)
==67943==    at 0x48DD958: __vfprintf_internal (vfprintf-internal.c:1687)
==67943==    by 0x48C7D3E: printf (printf.c:33)
==67943==    by 0x1091BC: main (Overflow.c:6)
==67943==  Uninitialised value was created by a stack allocation
==67943==    at 0x109169: main (Overflow.c:4)
```

> Time is limited, so we **won't** check upon this part, but please make sure that you understand the usage of `Valgrind` and the messages it gives you.
> Throughout the course, you will be asked to submit code with no memory issues.

The following TA(s) are responsible for this lab:
Jiale Wang <wangjl12024 AT shanghaitech.edu.cn>