1 Lab 6

Date: Sep 26, 2019

This document first describes the aims of this lab. It then provides necessary background. It then describes the exercises which need to be performed.

1.1 Aims

The aim of this lab is to familiarize you with the use of dynamic memory allocation in C. After completing this lab, you should be familiar with the following topics:

- Some feel for the strategies used by typical memory allocation libraries.
- Recognizing some symptoms of memory allocation errors.
- An introduction to some tools for diagnosing memory allocation errors.

1.2 Background

Possibilities for the lifetime of program data include the following:

Lifetime Coincident with Execution of the Entire Program The data comes to life when the program first starts up and stays alive as long the program is running. This kind of data can be stored at "fixed" memory addresses; these addresses are computed by the compiler.

Note that data which has lifetime coincident with that of the program is often referred to as *static data*; this is not identical to data declared using C's `static` keyword; all data declared as `static` is *static data*, but not all *static data* need be declared `static` (it can also be implicitly or explicitly declared `extern`).

Lifetime Coincident with Execution of a Block During runtime, data which has lifetime coincident with the execution of a block can be allocated on a LIFO stack. The data comes to life when the block is first entered and disappears once the exits.

Unfortunately, neither one of these strategies will suffice for data with other kinds of lifetime.

Most modern operating systems provide facilities for a running program to request non-stack memory dynamically while the program is running.

Windows Programs can call `HeapAlloc()` given a heap handle to an existing memory heap.

Unix Most Unix-based systems (Linux, Solaris, Android, OS X, ios, BSD) provide a system call `sbrk()` which can be used to control the upper limit of the data area. Memory can also be allocated using anonymous memory mappings.

Of course, the details of these specific system calls are hidden from a C programmer as long as the programmer is willing to use the memory allocation facilities (`malloc()` and friends) provided by the standard C library. The area of memory from which `malloc()` allocates memory is referred to as the *heap*.

1.3 Exercises

1.3.1 Starting up

Make sure you have followed the tips for setting up your environment for this lab. Get a copy of all the lab6 files in your `work/lab6` directory by copying the `~/cs220/labs/lab6/files` directory into your newly created `lab6` directory:

```
$ mkdir work/lab6
$ cd work/lab6
$ cp -r ~/cs220/labs/lab6/files/* .
```

Start logging your terminal interaction:

```
$ script -a lab6.log
Script started, file is lab6.log
```

In the sequel, when shell commands are shown with a `$` prompt, it means that the corresponding command was run using a `sh`-based shell like `bash`. So unless you have a strong preference otherwise, it is probably a good idea to use `bash` as your shell; so if you see your prompt containing a `%` or `>`, enter `bash` by simply typing `bash`.

1.3.2 Exercise 1: Observing Process Memory

This exercise demonstrates how one can monitor the memory size of a process under Unix systems.

Change over to the directory process-memory. Build the `process-memory` program by typing `make`. The program simply allocates `argv[1]` MB of memory and then sleeps for 1 minute.

Run the program but put it in the background using a trailing `&` by typing something like `./process-memory 10 &`. The system will output two numbers followed by the command-line prompt; the first number enclosed within square brackets `[]` is is the *job id* and the second number is the *process id* or PID. This

will be followed by a message from the background process saying that memory was allocated.

You can get back a fresh command-line prompt by simply typing a return.

Given the PID *pid* for the process, look at the memory size of the process using `ps -Fp` *PID*. You will get a couple of lines printed out: a header followed by the values for the specified process. Do `man ps` to get the *ps man page*; the relevant fields for memory are `SZ` which gives the total size of the process (in pages) and `RSS` which gives the number of pages which are resident in memory.

Repeat with `./process_memory 100 &` and observe the increase in `SZ`, whereas the `RSS` stays roughly the same since the allocated memory is totally inactive.

Note that when the background processes you started using the `&` shell operator terminate, the shell will print a message on your terminal.

1.3.3 Exercise 2: Buggy Program

Change over to the directory bug-program where bug-program.c contains four memory allocation bugs. Look at the program which should be reasonably understandable.

Build the program by typing `make`. Even with the multiple bugs, the program will probably run without a problem!! The fact that it may do so illustrate the insiduous nature of such bugs in that such buggy programs seem to work most of the time.

Can you understand why the indexes of the words are printed out in descending order?

The following sections will introduce tools which will help you spot the bugs, but see if you can find one-or-more bugs without using the tools. Some hints:

- The amount of memory needed to store a string **must** include space for the terminating `'\0` NUL‘ character.
- When `malloc()`'ing memory for some type `T`, the normal call will look like `T *pointerToT = malloc(sizeof(T));`.
- All allocated memory should be `free()`'d. Hence for every memory allocation call there should be a call to `free()`.
- Memory should not be accessed once it has been `free()`'d.

1.3.4 Exercise 3: Using Heap-Consistency Checking

Stay in the bug-program directory, but now run the program using

```
$ MALLOC_CHECK_=1 ./bug-program
```

[Note the trailing underscore in `MALLOC_CHECK_`].

The program should run ok, but the program output should also be followed by memory diagnostics (and possibly terminate with a segmentation fault). The way this works is that setting the `MALLOC_CHECK_` environmental variable links in a special version of the C library which does some consistency checking on the memory allocations. In the above case it may report some inconsistencies.

Look at the listed inconsistencies, and see if you can find any of the bugs which result in the memory inconsistency. Possibly firing up `gdb` and tracing the operation of the program may help.

1.3.5 Exercise 4: Using valgrind

You may or may not have been successful in fixing memory allocation bugs in the previous exercise. The problem with the diagnostics produced in that exercise is that they did not point you to specific source lines. An alternative is to use the `valgrind` tool which produces much better diagnostics at the source level.

Stay in the bug-program directory, but now run the program using

```
$ valgrind -v --leak-check=yes ./bug-program 2>bug-program.valgrind
```

This should record the standard error diagnostics in the `bug-program.valgrind` file. Now look at that file and the docs for valgrind and try to figure out the bugs. Look at the specific lines mentioned for `bug-program.c` to figure out the problems.

Since the report is quite long, search the file for errors mentioning `bug-program¬.c`. Look at the line specified by the line number mentioned in valgrind report. Consider the error reported by valgrind for that line and try to find the bug which may cause it. Note that the same bug may result in multiple errors.

Since it is very likely that each of the words in the Jabberwocky poem cause the same bugs, it may be a good idea to comment out all words other than the first and then run valgrind once again. This way the report will be shorter and it may be easier to find the bugs.

Once you have identified the bugs, fix them. Run a valgrind report so that valgrind reports a clean output without **any** errors.

1.4 References

Text, section 9.11.

Valgrind at <http://valgrind.org>.

Gcc Heap Consistency Checking.