

Lab 4

16 Apr 2009

There are two objectives for today's lab: learning a bit about a standard debugger, especially for tracking down memory issues, and learning to use dynamically allocated memory.

1 Running gdb

First, copy the file `lab4_buggy.c` from the course directory into your own. Compile it and run it with some command line argument: it should leave you with our friend the bus error.

Now run

```
gdb a.out
```

and after it loads, type

```
run arg1
```

This causes the program to run with the command line argument “arg1”, but here inside the GNU debugger, when it gets the error, it halts before clearing out memory. You can type things here, like

```
where
```

which gives a representation of the current call stack. However, right now it's not giving as much information as it could. Hit Ctrl-D to quit the debugger.

Back at the command line, recompile the file with the option `-g` (in addition to the C99 flags—if you've made an alias, you can just type `cc -g lab4_buggy.c`). This causes the compiler to keep extra information in the executable file to make gdb more useful. Once again, start up gdb and run the program inside it. Now when you type `where`, some of the functions (yours) have info about what the arguments are and what line the program was on.

But wait, there's more. Type

up 4

This descends into the call stack and actually shows you the line of the file that is currently running. If you type `where` again, you'll see that the deeper functions are still there, but as they are library calls they're not of interest to you. Here, though, you are 'in' your running `main` function. Type

```
print nums[2]
print *nums
print nums
print argc
print argv[1]
```

and you'll see the values of some of the expressions in the code. Type

```
print filename
```

and you'll stumble on the source of the problem: the value of this pointer is 4, rather than a pointer that points someplace useful, and if that pointer gets dereferenced an error will occur.

Another gdb trick we can use is setting breakpoints—giving us access to the running process *before* things go south. Type

```
break lab4_buggy.c:6
```

and then re-run the process. This time it should stop on line 6 (the assignment of `filename`). If you type

```
print filename
```

you get junk, because at this point the memory is declared but not init'ed yet; if you type

```
next
print filename
```

you'll see the assignment has been executed and the filename is still intact. (You may well see what the bug is by now, but don't fix it yet—we're on a roll here with gdb and there's a bit more to see.)

We're now heading into the `for` loop, and it seems likely that we'd want to track changes in several of the variables. Type

```
display filename
display i
display nums
```

and you'll see the current values of those variables, but more importantly, you've set up a recurrent display. Type

```
next
```

again and the updated values of those variables will be auto-displayed. Simply hit enter and the previous command (`next`) will be repeated, and in this way you can track the changes to the variables. Keep doing this until you see precisely where the `filename` variable goes bad, and you should now see, if you hadn't already, the source of the problem. Fix it and verify that the corrected code works (as in, doesn't crash—the program doesn't do much of anything).

The GNU debugger is very powerful and has lots more to it, but these basics—`run`, `break`, `next`, `print`, and `display`—are enough for now; make use of them as you debug your code in the rest of the lab, and get comfortable with them.

2 Using malloc

A few weeks ago, we saw one way of allocating arrays of strings: as a 2D array of `char`. The disadvantage of this is that you're generally allocating many more bytes than you need. The alternative is to allocate an array of `char` pointers (on the stack, as a plain old array), and then for each element of that array, set it to point to a piece of memory that has been separately allocated (on the heap, with `malloc`).

Write another version of `tail` now, that will read in lines from the terminal, continuing until the user ends input by hitting Ctrl-D. (This will register as an end-of-file; see the manpage for `fgets`.) Each line is guaranteed to be less than 80 characters. In this version, you will store lines in such a way that at the end of the program, you can print out the last ten of them.

The trick here is, at any given time you should only be allocating what you really need. So you can make an 80-char array for the purpose of reading them in, but if the user types one character per line, you shouldn't be allocating 80 characters for that line. Furthermore, your array to hang on to lines shouldn't be longer than ten. So you'll need to allocate space to hang on to each line (with `malloc`), and then you'll need to release it later (with `free`). You can make use of `gdb` to make sure your pointers are pointing to the things you think they are during the running of the program.

(This approach to `tail` would not be very efficient on large files, where you can just seek to the end as we discussed in class, but if the input is from the terminal, you have to read it all anyway and you *can't* seek backwards, so you need to buffer it somehow. This is not the most efficient way to do that, either, but it's not too bad.)