Linked List in C

In this lab we will learn how to:

1. Work with and create structures in C
2. Work with pointers
3. Allocate and free memory

We write a program that lets a user create and manipulate a doubly linked list.

It is a good idea to partner up for this lab so that you can discuss and reflect upon what you are doing.

Setup

Follow the post on piazza to create your github classroom lab repository.

Log into your UNIX environment and clone the repository and change directories into the repository working copy.

The lab repository includes both the skeleton code `ll.c`, a completed solution `sol.c`, along with a trivial `Makefile`. The goal is to complete the `ll.c` file in order to develop your own version of the solution. We encourage you to avoid looking at the solution until you have attempted to develop your own.

Note we have also included a file called `csetup.gdb` to setup gdb in a way that is nice for debugging programs that were built from C source code.

Getting started

This lab continues to help get us started working in ‘C’. It ties together several aspects we touched upon in assembly but via their more friendly C counter parts:

1. heterogeneous data types : C structures
2. using addresses to data items: C pointer types
3. dynamic heap memory: libc malloc and free calls

Building and running the binary

The goal is to complete the unimplemented functions in `ll.c`, as discussed below. To do you should edit and compile the file. To create an executable program from `ll.c` you will need to compile it with the 'C' compiler driver: 'gcc':

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

This will run all the steps of the tool-chain we have discussed: the pre-processor, compiler, assembler, and linker. If there are no errors it will produce a binary executable `ll`.

The '-g' option tells the compiler to build the executable with the ability to be run in the debugger. The '-o' option tells the compiler to create the executable in a file called 'll' in the current directory. If there are no errors this will create an executable file ll in the current directory that can be executed with the following command:

```
$ ./ll
```

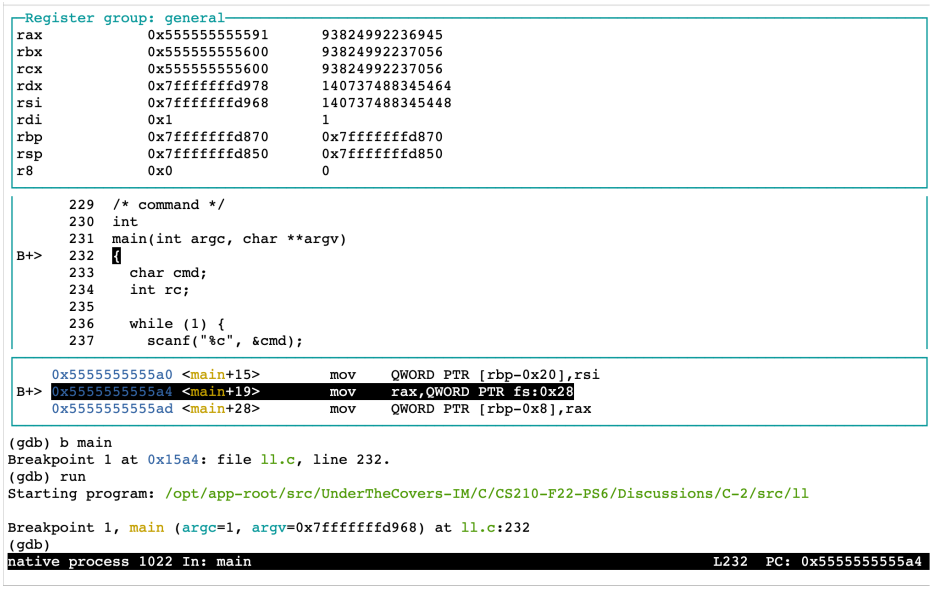
As always we can use make to automate the build commands. We have included an simple example Makefile. To build using the Makefile simply run `make` and to cleanup you can run `make clean`. The clean target will delete the binary.

Source level debugging

The following will start gdb with our binary and also run our csetup gdb commands.

```
$ gdb -x csetup.gdb ll
```

From here we can now set break points on lines of source code and functions. Eg. `b main` will set a break point that will get triggered when the main function is executed With this break point in place, if we then issue the `run` command we would expect to see something like the following`



One of the big values of gdb is that it knows the relationships between the assembly code, memory and the C source code. You can:

- set break points by specifying line numbers of the C Source
- you can single step source lines (`step`)
- you can use `next` to execute a function call as if it were a single line
- you can directly print the value of a variable using its C name: eg. p cmd

The ll program

Your task is to complete a C program that allows a user to manipulate a doubly linked list that contains integer numbers. The program starts with an empty list and maintains the notion of a current node where a node is a single element of the list.

The user can issue one of the several commands:

```
d)isplay current, D)isplay all, f)orward, b)ackward, I)nsert before,
i)nsert after, r)emove, c)lear, and q)uit:
```

Our job is:

1. Read the code to understand the basics of how such a program is written in C.

- visualize how the code builds and manages the linked list

1. Implement **i)nsert after, r)emove, and b)ackward** You have been provided with code implementing all other functions.

The following outlines what the program should do for each menu operation including those that you must implement.

- **d**: Information about the current node should be displayed. This includes the following:
  - memory address of the node
  - values all fields of the node
- **D** : information for all nodes on the list should be displayed, starting from the beginning – the node for which there is no previous node. This operation does not affect what node is current.
- **i** and **I** add a new node to the list after or before the current node respectively. If the list is empty then both will simply add the new node as the current node. To add a node you must use *malloc* to allocate sufficient memory for the new node, set it values, and then insert it correctly into the list.
- **f** and **b** move the current node forward or backward by one node respectively. The current node should not become null so you should never set the current node to NULL.
- **r** and **c** will remove the current node or all nodes respectively from the list and use *free* to release the associated memory. The *r* should update the current node based on the following rules:
  - If there is a previous node to the current node then after the removal the previous node should be the new current node.
  - If there is no previous node but there is a next node then after the removal the next node should be the new current node.
  - If there is no previous node and no next node, the current node is the only node on the list, so after the removal the new current node should be *NULL* indicating an empty list.
  - If the current node is *NULL*, the list is empty, removal should have no effect and hence the new current node should continue to be *NULL*.
- **q** should terminate the program.

The following is an example of using the program :

```
$ ./ll
I10I2I4
D
Node: 0x561fc7b9b6d0 : num=2    next=0x561fc7b9b6f0     prev=(nil)
Node: 0x561fc7b9b6f0 : num=4    next=0x561fc7b9b6b0     prev=0x561fc7b9b6d0
Node: 0x561fc7b9b6b0 : num=10   next=(nil)      prev=0x561fc7b9b6f0
q
$ ./sol
I10
D
Node: 0x5654815906b0 : num=10   next=(nil)      prev=(nil)
I2I4I7
D
Node: 0x565481590ae0 : num=2    next=0x565481590b00     prev=(nil)
Node: 0x565481590b00 : num=4    next=0x565481590b20     prev=0x565481590ae0
Node: 0x565481590b20 : num=7    next=0x5654815906b0     prev=0x565481590b00
Node: 0x5654815906b0 : num=10   next=(nil)      prev=0x565481590b20
I56Dq
Node: 0x565481590ae0 : num=2    next=0x565481590b00     prev=(nil)
Node: 0x565481590b00 : num=4    next=0x565481590b20     prev=0x565481590ae0
Node: 0x565481590b20 : num=7    next=0x565481590b40     prev=0x565481590b00
Node: 0x565481590b40 : num=56   next=0x5654815906b0     prev=0x565481590b20
Node: 0x5654815906b0 : num=10   next=(nil)      prev=0x565481590b40
$
```

Memory Bugs

Unlike other programming languages, in C we have complete control over the dynamic memory our program uses. This means as a C programmer we must explicit write the code to asks for memory via the `malloc` library function when we need it. It is also our job to write the code to keep track of the dynamic memory by recording the address that `malloc` returns. Further we must also explicitly write the code to `free` memory that it no longer needs. It is important to note it is our job to write correct code that:

1. does not access memory that has not been explicitly allocated
2. does not “leak” memory
3. does not attempt to free the same memory multiple times
4. does not attempt to use memory that it has freed

Leaking memory

A common kind of problem that arises is called *memory leaks*. A leak is when a program fails to free memory that it no longer needs and rather it continues to constantly allocate more and more memory. Eventually such a program can consume all the memory of the computer and will crash.

Lets explore what a memory leak looks like and see how we can use a tool like `valgrind` tool to find memory leaks.

Using Valgrind

First step is to use valgrind on our working version of the program to see what it tells us:

```
$ echo "I10i20i3DcDq" | valgrind --leak-check=full ./ll
==1152== Memcheck, a memory error detector
==1152== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==1152== Using Valgrind-3.15.0 and LibVEX; rerun with -h for copyright info
==1152== Command: ./ll
==1152==
Node: 0x4a4d080 : num=10        next=0x4a4d140  prev=(nil)
Node: 0x4a4d140 : num=3 next=0x4a4d0e0  prev=0x4a4d080
Node: 0x4a4d0e0 : num=20        next=(nil)      prev=0x4a4d140
==1152==
==1152== HEAP SUMMARY:
==1152==     in use at exit: 0 bytes in 0 blocks
==1152==   total heap usage: 5 allocs, 5 frees, 5,192 bytes allocated
==1152==
==1152== All heap blocks were freed -- no leaks are possible
==1152==
==1152== For lists of detected and suppressed errors, rerun with: -s
==1152== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Force a memory leak

Let's force our program to have a memory leak by removing the call to `free` in the `clear()` function. Can you see why this should lead to a memory leak.

Using Valgrind to find the leak

Now rerun valgind on leaky version:

```
$ echo "I10i20i3DcDq" | valgrind --leak-check=full ./ll
==1165== Memcheck, a memory error detector
==1165== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==1165== Using Valgrind-3.15.0 and LibVEX; rerun with -h for copyright info
==1165== Command: ./ll
==1165==
Node: 0x4a4d080 : num=10        next=0x4a4d140  prev=(nil)
Node: 0x4a4d140 : num=3 next=0x4a4d0e0  prev=0x4a4d080
Node: 0x4a4d0e0 : num=20        next=(nil)      prev=0x4a4d140
==1165==
==1165== HEAP SUMMARY:
==1165==     in use at exit: 72 bytes in 3 blocks
==1165==   total heap usage: 5 allocs, 2 frees, 5,192 bytes allocated
==1165==
==1165== 72 (24 direct, 48 indirect) bytes in 1 blocks are definitely lost in loss
record 2 of 2
==1165==    at 0x483B7F3: malloc (in /usr/lib/x86_64-linux-
gnu/valgrind/vgpreload_memcheck-amd64-linux.so)
==1165==    by 0x109201: insertbefore (ll-leak.c:27)
==1165==    by 0x10967A: docmd (ll-leak.c:245)
==1165==    by 0x10972C: main (ll-leak.c:282)
==1165==
==1165== LEAK SUMMARY:
==1165==    definitely lost: 24 bytes in 1 blocks
==1165==    indirectly lost: 48 bytes in 2 blocks
==1165==      possibly lost: 0 bytes in 0 blocks
==1165==    still reachable: 0 bytes in 0 blocks
==1165==         suppressed: 0 bytes in 0 blocks
==1165==
==1165== For lists of detected and suppressed errors, rerun with: -s
==1165== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
```

Other kinds of problems

There are many other problems that can arise when have complete control of dynamic memory. Three common ones are:

1. Accessing memory that was not explicitly allocated (eg. calling malloc to obtain 20 bytes for an array but then writing values beyond the 20 bytes : `char *array = malloc(20); array[21]=0;`).
2. Double frees : call free multiple times on the same address Eg. `char *array = malloc(20); free(array); free(array);`
3. Accessing memory that has been freed. Eg. `char *array = malloc(20); free(array); array[0]=12;`

Can you modify your code to create some of these bugs? After adding the bug run Valgrind and can help detect the bug.

By Jonathan Appavoo