CSE30 - HW #2

Array List Lab

Introduction

This is an assignment in two parts. In the first part you are given more practice with GDB, the GNU Project Debugger and a lot of practice with pointers by writing C code that uses them heavily.

The second part of the assignment focusses on translating your C code to Assembly. After completing the assignment you should know many of the important ARM instructions (particularly those that interact with memory) and understand what goes on under the hood of your C program.

This assignment is subject to the style guidelines given in lab 01.

All parts of this assignment should be completed and tested on the Raspberry Pi hardware or emulation environment. You are welcome to complete the C programming on the ieng6 servers, however you must test them on your Pi before making a submission.

1 Obtaining the starter code

Click on this link to create your repo and get the starter code for the assignment:

Lab 02 starter code

If clicking on the link doesn't work, copy and paste the url on your browser:

```
https://classroom.github.com/group-assignment- \
invitations/4ae1e652b2c303827eb1a8c14b143e78
```

If you are working with a partner, both of you will have to click on the above link. However, only one of the partners has to create a joint repo. The other partner only needs to join the repo following the instructions provided by the above link.

If you are working in a pair create your repo following the naming convention:

```
<partner1firstname-lastinitial>-<partner2firstname-lastinitial>
```

For example if the partners were Joe Smith and Lily Monroe, the naming convention should be JoeS-LillyM. Note that the partner whose first name appears earlier lexicographically should appear first in this naming convention. Choosing LillyM-JoeS as the repo name would be incorrect in this case. Do NOT include the keywords partner1 and partner2 in your repo name.

If you are working individually follow the naming convention:

```
<firstname-lastinitial>
```

Once you have create a new repo, or joined an existing repo, you will have access to the starter code and you can start working on the assignment.

Remember the general guidelines emphasized in class: compile and run your code often. Develop your code using test driven development (TDD). Submit to gradescope often. Following these guidelines allows you to be in control of the outcome of this assignment!

In class we spoke about how to develop code that is correct using TDD. Even after following TDD, you cannot avoid bugs in your code (they just won't be as scary). To deal with bugs you need debuggers like gdb....

Part I

2 GDB: Debugging practice

GDB allows you to inspect and modify the program as it runs. You can set breakpoints, inspect memory and registers, and much more. In this section you will do a couple of warmup exercises with gdb and in the process you are required to solve two puzzles. Provide your answers to the two puzzles in gdb_answers.c. As we walk you through these exercises we will ask you questions. Try to answer these questions for yourself. You don't have to commuicate the answers to us. The only thing that you need to submit are the answers to the two puzzles.

In order to use the debugger effectively, you must compile the source with with the "`-g`" flag. This includes debugging information in the compiled file such as function names, line numbers, etc.. In general, it's a good idea to always use "`-g`" unless you're compiling for public production. Also include the "`-O0`" flag (that's dash-oh-zero). This flag tells the compiler to turn off optimizations and translate your code as exactly as possible.

You may find this GDB cheatsheet from UT helpful:

`http://users.ece.utexas.edu/~adnan/gdb-refcard.pdf`

To complete the subsequent sections, first make sure your Pi is connected to internet. Then open a terminal and type the following command:

```
sudo apt-get install libc6-dbg
```

If you plan to work in the emulation environment, please use the setup on ieng6. gdb doesn't behave the same way in the docker environment as it does on the hardware. For more information refer to this Piazza post on using GDB:

https://piazza.com/class/isulbhfp4er74o?cid=189

2.1 Inspecting and Looking at Data

The file `gdb2.c` contains two static constants and three functions. Read the functions and figure out what they do. Here are some hints: `argv` is an array containing the strings that were passed to the program on the command line (or from gdb's run command); `argc` is the number of arguments that were passed. By convention, `argv[0]` is the name of the program, so `argc` is always at least 1. The `malloc` line allocates a variable-sized array big enough to hold `argc` integers.

`gdb2.c` has 2 puzzles which you need to decode by inspecting and looking at the data using gdb. The answers to each of the puzzle can be a string, an integer, character, floating point number or an array of integers, floating point numbers. You are essentially looking for an interpretation of the data in a format that means something to us humans even though its just ones and zeros for the machine. Fill in your solution to each puzzle in `gdb_answers.c`. You have to express your answer to each puzzle in string format even though it can be an integer, float or something else. If your answer is an array of integers or floating point numbers, separate all the elements in the array with a comma (,) and represent them as a string.

2.2 Puzzles

Compile the file `gdb2.c` into `gdbexec` using the appropriate flags (-g and -O0), and run gdb on this new executable.

Gdb provides you lots of ways to look at memory. For example, type "`print puzzle1`" (something you should already be familiar with). What is printed? That wasn't very useful, was it? Sometimes it's worth trying different ways of exploring things. How about "`p/x puzzle1`"? What does that print? Is it more edifying? You've just looked at `puzzle1` in decimal and hex. There's also a way to treat it as a string, although the notation is a bit inconvenient. The "`x`" (examine) command lets you look at arbitrary memory in a variety of formats and notations. For example, "`x/bx`" examines bytes in hexadecimal. Let's give that a try. Type "`x/4bx &puzzle1`" (recall that the "`&`" symbol means "address of"; it's necessary because the `x` command requires addresses rather than variable names). How many bytes of data is printed by the command "`x/4bx &puzzle1`"? Notice that the first byte at location `&puzzle1` is the same as the least significant byte of `puzzle1`. The reason for this should clear to you after we discuss byte ordering and endianness in class. In fact based on your observations so far (and knowledge on endianness), you should be able to conclude that the byte ordering on your machine is little endian.

OK, that was interesting and a bit weird, but we still don't know what's in `puzzle1`. We need help! Fortunately gdb has help built in. So type "`help x`". Then experiment on `puzzle1` with various forms of the `x` command. For example, can you figure out a way to modify the"`x/4bx &puzzle1`" command to print all the bytes in puzzle1 in the order that they appear in gdb2.c? That doesn't help you figure out the puzzle but it helps you understand that with examine you can view the bits stored at the granularity of bytes, halfwords and words and further using different representations. Continue experimenting with the different options available with `x` - you might try "`x/16i &puzzle1`". (`x/16i` is one of our favorite gdb commands—but since here we suspect that `puzzle1` is data, not machine instructions, the results might be interesting but probably not correct.) Keep experimenting until you find a sensible value for `puzzle1`. What is the human-friendly interpretation of `puzzle1`? (**Hint 1**: Although `puzzle1` is declared as an integer, it need not be an integer. Note that on a 32-bit machine a `long long` is 8 bytes, 4 halfwords, or 2 words.) (Express your answer for puzzle1 by initializing the string "puzzle1" in `gdb_answers.c`). Do not change any other code in that file.

Having solved `puzzle1`, we can now move on to `puzzle2`. It pretends to be an array of `integers`, but you might suspect that there is more to it. (**Hint** : You may have to apply two different formats to different parts of `puzzle2` to get a human friendly interpretation. Note that you can look at any arbitrary memory location with `x`, as in "`x/wx 0x8048500`".) First examine `puzzle2` as an array of characters. Next try to examine bytes starting at a memory offset from the base address using the command `x/bx puzzle2+4`. To understand the output of `x/bx puzzle2+4` recall pointer arithmetic and how it can be used to access array elements. Using your new found skills, figure out `puzzle2`. What is the human-friendly interpretation of `puzzle2`? (Express your answer for puzzle2 by initializing the string "puzzle2" in `gdb_answers.c`) Now submit your gdb_answers.c to gradescope to the assignment PA2_Puzzles to see if you are correct. Then revisit how these puzzles where initialized in gdb2.c and see if you can make sense of how those initializations relate to your human friendly interpretations of the the puzzles.

2.3 More Debugging

We've done all this without actually running the program. But now it's time to execute! Follow these steps to get more practice with debugging.

1. Set a breakpoint in `fix_array` by typing `b fix_array`. Run the program with the arguments `1 1 2 3 5 8 13 21 44 65` by typing `r 1 1 2 3 5 8 13 21 44 65`. When it stops, print `a_size` and verify that it is 10. Did you really need to use a `print` command to find the value of `a_size`? (**Hint**: look carefully at the output produced by gdb.)

2. Print the value of `a`. To execute the next statement you can step through your code by typping `step` or just `s`. Step six times. You'll note that one of the lines executed is a right curly brace; this is common in gdb and often indicates the return from a function. After returning, print the value of `a`. Step again (a seventh time). Print the value of `a` now. By how much did it change since you last printed it? What is the value of `i`?

3. At this point you should (again) be at the call to `swapper`. You already know what that function does, and stepping through it is a bit of a pain. The authors of debuggers are aware of that fact, and they always provide two ways to step line-by-line through a program. The one we've been using (`step`) is traditionally referred to as "step into"—if you are at the point of a function call, you move stepwise into the function being called. The alternative is "step over"—if you are at a normal line it operates just like `step`, but if you are at a function call it does the whole function just as if it were a single line. Let's try that now. In gdb, it's called `next` or just `n`. So, type 'next'. what line did you just finish executing? (Incidentally, in gdb as in most debuggers, the line shown is the next line to be executed.) Use `n` a second time to step past that line, verifying that it works just like `s` when you're not at a function call. You should be back at the line where `swapper` is going to be called.

4. It's often useful to be able to follow pointers. Gdb is unusually smart in this respect; you can type complicated expressions like `p *a.b->c[i].d->e`. Here, we have kind of lost track of `a`, and we just want to know what it's pointing at. Type "`p *a`". You should get 1.

5. Often when debugging, you know that you don't care about what happens in the next three or twelve lines. You could type "`s`" or "`n`" that many times, but we're computer scientists, and CS types sneer at work that computers could do for them—especially mentally taxing tasks like counting to twelve. So on a guess, type "`next 12`". (Note that you have typed n twice already). What line are you at (this should be the line number that will be executed next)?

6. What is the value of `*a` now? Is it 21? How did I know :)?

2.4 Just scratched the surface...

GDB is an incredibly powerful tool that can give you instruction-level insight into programs, even when they've been heavily optimized. We haven't looked at GDB under these circumstances, but you should keep it in mind for your own future projects. We hope you have enjoyed the warmup exercises with gdb and will continue to use it as you develop code in the subsequent sections of the assignment and your future projects.

3 Fun with Pointers (Optional)

This is a highly recommended exercise that you can expect to see on an exam. However, it does not carry any points on this assignment and you don't have to turn it in.

3.1 Pointer mechanics

Variable	Value	Address
x	0x0000	0x0002
y	0x0005	0x0004
z	0xFFFF	0x0006

1. Given the data shown above from a 16-bit system, with integer x, unsigned integer y, and integer z, what will the following code snippet print out? (Note that this is a thought experiment. Don't try to run it until you have thought it through.)

Listing 1: Pointer Mechanics Problem 1 Snippet

```
int *p = &x;
printf("%d\n", *p);
p++;
x = (int) p;
printf("%d\n", x);
printf("%d\n", *p);
printf("%d\n", *(&(*p)));
printf("%d\n", *(p-1) + 1);
```

4 Array Implementation of Sorted List

Submission file: arraysort.c. Submit to the gradescope assignment PA2_C.

4.1 C Implementation

You were introduced to linked lists in class, where the elements of the list were not at contiguous locations in memory. In this part you are asked to implement a list with elements arranged in sorted order (also called a sorted list). However, instead of using a linked-list implementation, implement your list as an array of integer elements. The specific functions that you should implement are declared in the header file arraysort.h in your skeleton code. The header file also contains the declaration of a structure called `list` which has two member variables:

- `int * sortedList` is a pointer to an array of integers in sorted order, which forms the sorted list
- `int size` is the number of elements in the list
- `int maxSize` is the maximum elements that can be stored in the list

Your implementation should go into the file `arraysort.c` which includes `arraysort.h` and implements the following functions. Do not modify `arraysort.h`. You should not use any of the sorting functions available with the C library in the implementation of your functions. Before you begin implement a skeleton test_arraysort.c program that has the functions test_createlist(), test_insert(), test_remove_val(), test_get_max_value(), test_get_min_value(), test_search(), test_pop_min() which test the corresponding functionalities of your C code. Follow the TDD style of developing your code. Please also read the section: "Checking for memory leaks" before beginning your implementation. Make the following assumptions for your array list:

- Only non-negative numbers are stored in your arraylist.
- Duplicate elements are allowed

1. `list* createlist(int maxElements)` - This function creates an empty list that can store up to ‘maxElements’ elements and returns a pointer to the list. The function should dynamically allocate the space required to store elements in the new list.

2. `int insert(list *ls, int val)` - This function takes a pointer to the list and an integer value as input. It should insert the value ‘val’ into sortedList, update the number of elements in the list and return the index where the element was inserted. If the list is full before inserting the element, it should increase the size of the list to double its previous size and then insert the element. Note that the resulting list should be sorted and there should be no information loss. Insertion of duplicate elements are allowed. The function should return -1 if no valid list was passed to it.

3. `int remove_val(list *ls, int val)` - This function takes a pointer to the list and an integer value as input. It should delete all instances of elements in the sortedList with value ‘val’ (i.e. all duplicates should be removed), updates the number of elements remaining in the list and return the number of elements that were deleted. Once again the resulting list should be sorted. The maximum elements that the list can hold should remain unchanged.

4. `int get_max_value(list *ls)` - This function takes a pointer to the list as input and returns the maximum value in the list OR -1 if the list is empty. You will implement this function in ARM assembly (see Section 3.2).

5. `int get_min_value(list *ls)` - This function takes a pointer to the list and returns the minimum value in the list OR -1 if the list is empty. You will implement this function in ARM assembly (see Section 3.2).

6. `int search(list *ls, int val)` - This function returns the index of the first occurrence of ’val’ in the list. It returns -1 if the value ’val’ is not present in the list.

7. `int pop_min(list *ls)` - This function returns the minimum value from the list and removes it from the list. It returns -1 if the list is

empty.

8. `void print(list *ls)` - This function prints the contents of the sorted list on a single line as follows:

```
1 4 6 7 8
```

5 Checking for memory Leaks

Information in this section is useful for completing both Part I and Part II of the assignment. Valgrind is a memory mismanagement detector. It shows you memory leaks, deallocation errors, etc. Actually, Valgrind is a wrapper around a collection of tools that do many other things (e.g., cache profiling); however, here we focus on the default tool, memcheck. Memcheck can detect:

- Use of uninitialised memory
- Reading/writing memory after it has been free'd
- Reading/writing off the end of malloc'd blocks
- Reading/writing inappropriate areas on the stack
- Memory leaks – where pointers to malloc'd blocks are lost forever
- Mismatched use of malloc/new/new [] vs free/delete/delete []
- Overlapping src and dst pointers in memcpy() and related functions
- Some misuses of the POSIX pthreads API

To use this on our example program, test.c (provided in public/hw2_skeleton directory), try

```
gcc -o test -g test.c
```

This creates an executable named test. To check for memory leaks during the execution of test, try

```
valgrind --tool=memcheck --leak-check=yes --show-reachable=yes
--num-callers=20 --track-fds=yes ./test
```

This outputs a report to the terminal like

```
==13296== Memcheck, a memory error detector
==13296== Copyright (C) 2002-2011, and GNU GPL'd, by Julian Seward et al.
==13296== Using Valgrind-3.7.0 and LibVEX; rerun with -h for copyright info
==13296== Command: ./test
==13296==
==13296==
==13296== FILE DESCRIPTORS: 3 open at exit.
==13296== Open file descriptor 2: /dev/pts/0
==13296==    <inherited from parent>
==13296==
==13296== Open file descriptor 1: /dev/pts/0
==13296==    <inherited from parent>
==13296==
==13296== Open file descriptor 0: /dev/pts/0
==13296==    <inherited from parent>
==13296==
==13296==
==13296== HEAP SUMMARY:
==13296==     in use at exit: 35 bytes in 2 blocks
==13296==   total heap usage: 3 allocs, 1 frees, 47 bytes allocated
==13296==
==13296== 16 bytes in 1 blocks are definitely lost in loss record 1 of 2
==13296==    at 0x4835978: malloc (vg_replace_malloc.c:263)
==13296==    by 0x849B: main (test.c:9)
==13296==
==13296== 19 bytes in 1 blocks are definitely lost in loss record 2 of 2
==13296==    at 0x4835978: malloc (vg_replace_malloc.c:263)
==13296==    by 0x844F: main (test.c:5)
==13296==
==13296== LEAK SUMMARY:
==13296==    definitely lost: 35 bytes in 2 blocks
==13296==    indirectly lost: 0 bytes in 0 blocks
==13296==      possibly lost: 0 bytes in 0 blocks
==13296==    still reachable: 0 bytes in 0 blocks
==13296==         suppressed: 0 bytes in 0 blocks
==13296==
==13296== For counts of detected and suppressed errors, rerun with: -v
==13296== ERROR SUMMARY: 2 errors from 2 contexts (suppressed: 13 from 6)
```

Let's look at the code to see what happened. Allocation 1 (19 byte leak) is lost because p is pointed elsewhere before the memory from Allocation 1 is free'd. To help us track it down, Valgrind gives us a stack trace showing where the bytes were allocated. In the 19 byte leak entry, the bytes were allocate in test.c, line 5. Allocation 2 (12 bytes) doesn't show up in the list because it is free'd. Allocation 3 shows up in the list even though there is still a reference to it (p) at program termination. This is still a memory leak! Again, Valgrind tells us where to look for the allocation (test.c line 9).

Use valgrind to make sure your C implementation doesn't have memory leaks.

Part II

This section is due later. See the assignment calendar

5.1 ARM Assembly Implementation

In this section of the assignment, you must implement all the sorted list functions in ARM assembly. You should use the same test cases you used to test your C implementation to test the ARM versions. Simply replace the calls to the C versions with the names of the ARM implementations.

Use the header file `arraysort_ARM.h` and the provided ARM assembly stubs to complete your implementation. To be explicit, you must complete the following functions:

1. int get_max_ARM(list *ls)
2. int get_min_ARM(list *ls)
3. createlist_ARM.s
4. insert_ARM.s
5. pop_min_ARM.s
6. print_ARM.s
7. remove_val_ARM.s
8. search_ARM.s

The expected behavior of these functions was given in the previous sub section. You will implement each function in separate .s files. We have provided you with the skeleton code for the ARM implementation and a makefile to compile your C and ARM code. Please expand on the makefile to compile with flags you may need, (like -Wall and -g for gdb).

6 Submission

Commit and push all your code on github. Make sure you do this often.

For part I, submit your `gdb_answers.c` populated with your answers to the puzzles to the gradescope assignment `PA2_Puzzles`. Submit the C file `arraysort.c` to the gradescope assignment `PA2_C`. You must submit both these before the deadline for Part I (see the assignment calendar)

For part II, you must submit each of the following files individually to the gradescope assignment `PA2_ARM`:

- createlist_ARM.s
- get_max_ARM.s
- get_min_ARM.s
- insert_ARM.s
- pop_min_ARM.s
- print_ARM.s
- remove_val_ARM.s
- search_ARM.s

Acknowledgements

Thanks to Geoff Kuenning at HMC and Michael DeBlasio at UCSD for the original versions of the assignment.