Number Sort Project

Cooper Union

ECE-251

Professor Billoo

Ali Ghuman, Xue Ru Zhou, Husam Almanakly, Layth Yassin

April 2021

Contents

1 Objective

The objective is to sort a given set of unsigned ints that are a maximum of 32 bits. One hundred values is the maximum number that can be inputted. The unsorted input is to be read in from an input file declared by the user, and the sorted output placed into another file declared by the user. The output will be a list of newline separated values.

2 Architecture

2.1 Reading in the file

To read in input, the user is first prompted to enter a file name. `printf()` is used to retrieve the input file name. The file is then opened using `fopen()`. If the file cannot be opened, an error is thrown. This is done by comparing the return value of `fopen()` with null. Using the file pointer returned by `fopen()` upon success, the input file is looped through and the values are read into the input array using `fscanf()`. These values are read until either 100 values are read or the EOF is encountered. To check for EOF, `foef()` will be used. If more than 100 values are present, an error will be thrown. The fully populated array is then sorted.

2.2 Writing the output file

Once the input is sorted into the output array, the output file declared by the user is opened. This file is then populated with newline separated values by looping through the sorted array and using `fprintf()`. The size of the array is kept in a register, and an iterator that increments as the array is outputted is compared with this register. If they are equal, the array has been finished outputting. Once completed, the output file is closed and the program ends.

2.3 Selection Sort

To sort the numbers in ascending order, the selection sort algorithm was implemented. The numbers from the input file are stored into an array called `unsorted`. The algorithm uses both a current element index and a current minimum index. At the end of an iteration through the entire loop, the value of the array at the minimum index is stored into the array called `sorted` – at the leftmost open index. The value that was stored is now replaced by a null in the `unsorted` array. This continues until all array elements are stored in `sorted`.

2.3.1 Registers

r0	address of the sorted array (in `load_minimum`)
r4	address of the unsorted array
r5	index/length of the sorted array
r6	value of current minimum
r7	element in unsorted array at index in r11
r8	index of the first non null element in unsorted array
r9	length of the unsorted array
r10	index of current minimum
r11	index of the unsorted array

Note: The indices of the array are not by element but by byte.

2.3.2 Outer Loop

In `outer_loop`, the index of the current minimum and the current index of the unsorted array are initialized to 0 and the unsorted array is loaded into r4. The value of the minimum is set to the first non null element in the unsorted array.

Then, the current minimum is compared with null. If it is a null, then break to `non_null`. If the current minimum is not a null, then the index of the first non null element is set as the current minimum. Break to `sort`

In `non_null`, the index in r8 is incremented by 4 bytes (each int is 4 bytes) and then jumps back to `outer_loop`.

2.3.3 Load Minimum

The purpose of this section is to store the minimum value into the leftmost available index in the sorted array. If the length of the sorted array equals that of the unsorted array, then break to `output`. If not equal, then the minimum is stored into the sorted array and then replaced with a null in the unsorted array.

The index of the sorted array increments by 4. Break to `outer_loop` to iterate through the array again.

2.3.4 Sort

The current index of the unsorted array is compared to the length of the array. If equal, that means that we are at the end of the array. Break to `load_minimum` to load the current minimum value (after comparing with all the other non null values) into the sorted array.

If not at the end of the array yet, increment the index by 4 bytes.

Compare value where the index is pointing to with null. If equal, do nothing (loop `sort` again). If not equal, then the element is not null. Compare current minimum value to the value in the unsorted array. If less than, loop `sort` again.

Otherwise, branch to `index`, which replaces the minimum with the current element in the unsorted array. After completing this task, branch back to `sort`.

2.3.5 Alternative Solution: Swapping

An alternative way of implementing selection sort is swapping values in the same array until sorted. This would rid the need of having another array to store the sorted array in. In order to do this, the comparison with null is invalid; we must relate iteration number (how many times we have gone completely through the array) to the starting index so as to not start from the left sorted section of the array.

Even though this was already written up in a separate version of the project with a different input/output method, in the end, the current sorting implementation works best with the current input/output method. The other version is more prone to bugs (as seen through many test runs).

2.3.6 Alternative Solution: Increment by Element

We can also increment the indices by element (+1), and then using an instruction such as `ldr r9, [r8, r3, LSL #2]`, which multiplies the index by $2^2 = 4$.

This would be fine as well, but could be more confusing than the current method for some people.

2.4 Print Error Messages

The given program was designed to only read in and sort the first 100 numbers of an input file. If the input file included more than 100 lines, the code would exit and print an update to the user, stating "100 number limit exceeded. Only the first 100 numbers will be sorted."

If the inputted file name could not be opened, the program would exit and print a message to the user explaining that the text file could not be found in the current directory stating, "Error: Could not open inputted file name."

3 Design Considerations

3.1 Assumptions

3.1.1 Limiting amount of numbers in input file

The maximum amount of numbers is set to 100. We were not given a specific limit in the project objective, but implementing a limit is necessary due to array allocation. For this project, 400 bytes were allocated for the array($100 ints * 4 bytes$).

3.1.2 Signed Ints

Choosing between signed ints and unsigned ints, we decided to go with signed ints. Even though unsigned ints allow more positive values (up to 4,294,967,295 in 32-bit), throwing away the possibility of sorting negative numbers is meaningless. Besides, the prevalence of negative numbers is far greater than numbers above 2,147,483,647 (maximum positive signed 32-bit int).

3.2 C Functions

Throughout the program, existing C std library functions are leveraged to complete difficult tasks. To output messages prompting the user for the names of the input and output files, the std library function `printf()` is used. The function `printf()` function outputs to the function `stdout` (i.e., the terminal), which allows the user to see the printed outputs. Writing the code for a function that accomplishes the same tasks as function `printf()` in assembly is complex and impractical. This is mainly because several cases and test conditions would have to be accounted for. In addition, writing a function `printf()` function without the use of any libraries would require knowledge of machine code.

To complete the task of taking in the user-specified input and output files from `stdin` (i.e., the keyboard), `scanf()` is used. Writing a function to take user input in assembly would be difficult for similar reasons as the function `printf()` function.

The first input that the system prompts the user to input is the file containing the unsorted numbers. To access the numbers in the file, the file must be opened. This is accomplished using the `fopen()` function. The `fopen()` function is ideal for this system because it takes in the file name pointer and the file access mode pointer as parameters, which are both easily accessible. In addition, the `fopen()` function allows for simple error handling. If the return value is null, this means `fopen()` did not properly work. The system checks for the return value and if it is null, an error message will be printed. The `fopen()` function is also used to open the user-specified output file.

The input and output files are closed once all desired reading and writing is complete using the `fclose()` function. The only parameter of the function is the pointer to the stream, which is easily accessible. The `fclose()` function returns an integer value when called. If the return value is 0, this means the specified file was successfully closed. If the return value is non-zero, then the specified file was not closed properly. This proved to be vital during debugging.

In order to perform operations on the unsorted numbers, they are stored in an array. To accomplish the task of reading in the numbers from the input file, the `fscanf()` function is used. The `fscanf()` function takes in as parameters the pointer to the stream, the format specifier, and memory addresses to store what is read by the `fscanf()`. In this system, each number (i.e., each line of the input file) is read one at a time. This number is loaded into an array which contains the unsorted numbers. The `fscanf()` function is ideal as it returns the number of input items successfully matched and assigned, which allows the system to identify when the end of the input file is reached in a simple manner. If the return value is not equal to 1, then no input is matched and assigned, which means the end of the file is reached.

4 Problems and Solutions

4.1 Input Data

One of the first major issues faced in this project was reading in the data file with the 100 numbers. The goal was to ultimately allow the user to type the name of an input file that would be sorted. At first, `fgets()` was used to read in the file name, however it was quickly realized that `fgets()` would not be the best option as it reads in the newline character. When `fopen()` was called, the file name was not recognized and the text file would not open. This was discovered when the error branch was included and showed clearly that the file name was never recognized.

Another error that occurred when reading in the input data involved using the `feof()` function. The initial plan was to, read in the first 100 numbers, however a conditional was also required when the input file was less than 100 lines. The proposed solution was `feof()`. While this seemed to function correctly, upon testing using a print statement to show each number from the input file, the last number would be read twice. This involved an issue with the function `feof()` itself, and thus was avoided by checking if the `fscanf()` function returned a 0 or a 1, which indicated whether or not the last number had been reached.

4.2 Sorting Iterator

The second major challenge emerged during the sorting of the input data. The register r12 was arbitrarily chosen as the iterator of the output array, from which r12 would be compared to the length of the input file in order to determine when the sorting had been completed. This resulted in the sorting algorithm leaving the program in an infinite loop, and after analysis at each iteration it was discovered that r12 had been changing inconsistently upon each loop. It was furthermore realized through the ARM calling conventions, that r12 was used as the intra-procedure call scratch register. The iterator was then updated to use a different register (specifically r5) and the sort completed successfully.

4.3 Memory Structure

Lastly, another error that had to be dealt with involved the memory structure previously determined. The sort was functioning successfully for the small test files, however once the input data was increased to greater than 25 lines, the program would crash without sorting. After observing the 25 number limit, it was noted that the memory allocated to each array only included 100 bytes of memory, which only allotted space for 25 int data types. This was updated to include at least 400 bytes of memory (for the goal of 100 ints, each with a maximum of 4 bytes in memory).

5 Deliverables

5.1 README

To build and run the assembly code, use command ”make all” while on the micro-lab server, and it should create an executable named a.out. To execute, run ./a.out from the current directory and enter the prompted input.

5.2 MakeFile

The Makefile automatically removes a pre-existing executable named ”a.out” and builds the given source code. If no file named a.out exists, the remove command does nothing and the source code continues to build the source code normally.

```
# Makefile

all: a.out

a.out: proj1.S .clean
	arm-linux-gnueabi-gcc $< -o $@ -ggdb3 -static

.clean:
	rm -f a.out
```

6 Conclusion

The input was successfully sorted and performed error checking to make sure the program functioned appropriately. An error was returned if the files could not be opened, if the names of the files were too long, or if more than 100 values were inputted. Both positive and negative integer values were sorted, as unsigned ints were utilized. Deeper knowledge was attained concerning the workings of registers and C functions utilized in assembly. As many C functions were used, testing with different values and researching various methods of implementation led to greater understanding of how these functions were employed and which tools were the best for the given task. Lastly, by experimenting with various methods of reading input and writing output, debugging numerous segmentation faults, and finally arriving at success, understanding of computer architecture was heightened.