Shahreer Al Hossain, Brian Khaimov, Giuseppe Quaratino, Fakharyar Khan
ECE-251: Computer Architecture
Professor Billoo

Project # 2

In this project, our group created a program that reads in a file containing a list of 32 bit integers and outputs a sorted version of the list to another file. The file to be read is formatted so that each line contains only one number, and the same format is produced for the output file. The code can support a file size of up to 6144 (1024 32-bit integers and 1023 newline characters) and bytes which means that the file can have no more than 1024 integers.

Our program is separated into a data section (.data) and a code section (.text). In the (.data) section, the arguments for the various functions imported from the C standard library are initialized. Further, memory is allocated for the array that will contain the integers from the file to be read. The code section was broken up into three segments, which also contained two to three branches each. The first segment opens the file and reads through it and stores each number in the array. The second segment which does most of the “heavy lifting” for this program, performs a selection sort on the array, placing them in order. The final segment opens the output file and parses through the array, writing each element to the file.

Segment #1: File I/O and Array Initialization

```
.data

.balign 4
        pattern: .asciz "%14[^\n]%*c"
.balign 4
        input: .asciz "Please enter the name of the input file: "
.balign 4
        finished: .asciz "\nThe file is successfully sorted. Please check the contents of output.txt.\n\n"
.balign 4
        badopenmsg: .asciz "This file cannot be opended. Please check if the name of the file in entered
correctly and if the file exists in the same directory.\n"
.balign 4
        tooLarge: .asciz "This file has more than 1024 integers. Please reduce the size of your list.\n"
.balign 4
        readmode: .asciz "rb"
.balign 4
        writefile: .asciz "output.txt"
.balign 4
        writemode: .asciz "wb"
.balign 4
        return: .word 0
.balign 4
        scanformat: .asciz "%li"
.balign 4
        printformat: .asciz "%li\n"
.balign 32
        array: .space 4096
.balign 4
        filename: .space 32
.balign 4
        filepointer: .word 4
```

C functions from standard C library used in the program:

fopen( ): opens the file whose name is specified in the parameter filename and associates it with a stream that can be identified in future operations by the file pointer returned
fprintf( ): writes the C string pointed by format to the stream
fclose( ): closes the file associated with the stream and dissociates it
fscanf( ): reads data from stdin and stores them according to the parameter format into the specified locations
printf( ): writes the C string pointed by format to the standard output (stdout)
feof( ): checks whether the end-of-file indicator associated with stream is set, returning a value non-zero if it is.
fscanf( ): reads data from the stream and stores them according to the parameter format into the locations

```
main:

// Storing the link register to call it back at the end
        ldr r1, =return
        str lr, [r1]

        ldr r0, =input
        bl printf


// Scanning in the name of the input file
        ldr r0, addr_of_pattern
        ldr r1, addr_of_filename
        bl scanf

// Opening the input file using the fopen C function
        ldr r0, addr_of_filename
        ldr r1, =readmode
        bl fopen
        mov r4, r0 /* File Pointer is returned by the fopen C function to refister r0 */
        ldr r1, =filepointer
        str r0, [r1]

// Checking if the file is openable
        ldr r0, =filepointer
        ldr r0, [r0]
        cmp r0, #0       /* If the file pointer is null then exit the program */
        beq badopen

// Initialize the readloop counter (register r5)
        mov r5, #0
        ldr r6, =array
```

```
readloop:
        mov r0, r4
        bl feof
        cmp r0, #0
        bne sort
        cmp r5, #4096
        bgt SizeError
        mov r0, r4
        ldr r1, =scanformat
        mov r2, r6
        bl fscanf
        add r5, r5, #4
        add r6, r6, #4
        b readloop
```

As writing and reading to and from files are high level processes, our group decided to import C libraries to accomplish this. In order to open the file, the arguments for fopen, the memory addresses of the name of the text file to be read, and the filemode, were loaded onto R0 and R1respectively. Then, the function fopen was called and this returned a file pointer which was saved in R4 for later use. The filemode informs the fopen how the user wants to interact with the file and since we only wanted to read in the contents of the file, this was set to “rb”. Once the file was opened, the registers needed to parse through the file, and the array was set up. R5 would store the current index of the array and ultimately will be used to determine the number of integers in the file which would also be the length of the array. R6 stores the memory address of the array and will be used to move through the array.

Since memory was allocated for the array in the data section, there was no way to determine how much memory would be needed to fully process the file. Therefore, we decided to have the array be initialized with enough memory to handle 4096 bytes of memory or 1024 32-bit integers. Although this sets an upper bound on how many integers the program can handle, if we hadn’t done this, there would still be a built in limit to how many integers our program can handle, albeit a much larger one. Once the file was read, we wanted to free up any remaining space in the array by deallocating memory from the array but we couldn’t figure out how to do this. Instead, the array is kept as is but the program doesn’t interact with unoccupied memory in the array meaning that in order to sort files with only 2 integers, the program would need to use 4096 bytes of memory.

While parsing through the file, the function feof was used to determine when the end of the file is reached. It takes in the file pointer as an argument and returns 0 if it reached the end of the file and some nonzero integer if it did. So in the loop label, the file pointer located in R4 is moved to R0, and feof is called. R0 now containing the return value of feof is compared to 0. If it’s 0, the program branches to the sorting segment and if not, it executes the rest of the loop.

After this, the arguments for fscanf are loaded onto their appropriate registers. Fscanf reads from the current position in the file, as indicated by the file pointer, and then scans in a specified unit of data which is determined by the scanformat variable. Since each number in the file is a long, the li specifier was used in the scanformat variable. Then, the memory address of the current position in the array, stored in R6, is moved to R2. The integer scanned from the file is stored in the block of memory that starts with what's in R2. Then the index register is incremented by 1 and R6 is also moved up by 1 which moves the register value to the next available memory address in the block of memory in the array. Since the end of the file hasn't been reached yet, the program branches back to the loop label.

Checking for bad files and large files

If there are more than 1024 integers in the file, then the code will branch to the *SizeError* branch, which prints out an error message and exits out of the program. When the fopen C function is unable to open the input file, it returns the value 0 to the R0 register. So, the value of the register R0 is compared with 0 to check if the file is successfully opened or not. If it is not successful, the code branches to the *badopen* branch which prints out an error message and exits out of the program.

```
badopen:
        ldr r0, =badopenmsg
        bl printf
        ldr lr, =return
        ldr lr, [lr]
        bx lr

SizeError:
        ldr r0, =tooLarge
        bl printf
        ldr lr, =return
        ldr lr, [lr]
        bx lr
```

Segment #2: Selection Sort Algorithm and Its Implementation

The following C selection sort algorithm was followed when writing the selection sort algorithm in the ARM assembly language. The outer loop traverses the given array from its first index to its second to the last index. The current minimum index is set at i. The innerloop traverses the array from its (i+1)th index to its last index to check if the value of the array at the current minimum index is greater than the value of the array at its jth index or not. If a smaller value is found, the minimum index is changed to the current index in the inner loop. As a result, the innerloop helps determine the minimum value in the unsorted section of the array. Afterwards, the minimum value of the array is stored in the ith index of the array, and the value of the array at ith index is stored in the current minimum index of the array. After the inner loop terminates and the minimum is placed in its "correct" position, part of the array is sorted. This process is repeated until the whole array is sorted. Since we are checking if the array[curr_min_index] is greater than array[i] or not, we end up with an array that is sorted in ascending order.

```c
// C program for implementation of selection sort

#include <stdio.h>

void selectionSort(int arr[], int n)
{
        int i;
        int j;
        int min_idx;

        // One by one move boundary of unsorted subarray
        for (i = 0; i < n-1; i++)
        {
                // Find the minimum element in unsorted array
                min_idx = i;

                for (j = i+1; j < n; j++){
                        if (arr[min_idx] > arr[j]){
                                min_idx = j;
                        }
                }
                // Swap the found minimum element with the first element
                swap(&arr[min_idx], &arr[i]);
        }
}

void swap(int *xp, int *yp)
{
        int temp = *xp;
        *xp = *yp;
        *yp = temp;
}
```

This is the Arm assembly implementation of C selection sort implementation. Here, the *sort* branch initializes the indexes of the inner loop, the outer loop, the size of the array, the offset variables, and the temp variables. The *outerloop* branch and the *innerloop* branch follow almost the same logic as the loops included in the C algorithm. The *increment* branch works similarly to the swap function that is included in the C algorithm. Since a long contains 4 bytes, the offset variables are incremented by 4. This allows us not to use additional registers and other shift instructions.

```
// A selection sort is used to sort the contents of the array that were read from the input file.
// The sorting algorithm sorts positive and negative integers in an ascending order.
sort:

    SUB r9, r5, #4      /* stores length of array */
    mov r5, #0          /* array parser for first loop */
    mov r6, #0          /* stores index of minimum */
    mov r7, #0          /* temp */
    mov r8, #0          /* array parser for second loop */
    ldr r10, =array     /* loading the address of the array */
    mov r11, #0         /* used to obtain offset for min */
    mov r12, #0         /* used to obtain offset for second parser access */

outerloop:
    cmp r5, r9          /* check if first parser reached end of array */
    beq write           /* if it did the array is sorted and braches to write subroutine */
    mov r6, r5          /* set the min index to the current position */
    mov r8, r6          /* set the second parser to where first parser is at */
    b innerloop         /* start looking for min in this subarray */

innerloop:
    cmp r8, r9          /* if the loop reaches end of list, then min is found */
    beq increment       /* get out of this loop and increment 1st parser */

    ADD r7, r10, r6     /* adds offset to r10 address storing it in r7 */
    ldr r11,  [r7]      /* loads value of min in r11 */

    ADD r7, r10, r8     /* adds offset to r10 address storing it in r7 */
    ldr r12, [r7]       /* loads value of second parse into r12 */

    cmp  r11, r12       /* compare current min to the current position of 2nd parser */
    movgt r6, r8        /* set new min to current position of 2nd parser */
                        /* if value of 2nd parser is samller than min */
    add r8, r8, #4      /* increment second parser */
    b innerloop         /* repeat */

increment:
    ADD r11, r10, r5    /* adds offset to r10 address stored in r11 */
    ldr r8, [r11]       /* loads value in memory address in r11 to r8 */
    ADD r12, r10, r6    /* adds offset to r10 address stored in r12 */
    ldr r7, [r12]       /* loads value in memory address in r12 to r7 */
    str r8, [r12]       /* stores value of first parser where min was */
    str r7, [r11]       /* store value of min where first parser was */
    add r5, r5, #4      /* increment the first parser */
    b outerloop         /* branch to outerloop1 instructions */
```

Segment #3: Writing to Output File

The file point of the input file was stored in the R4 register. The file pointer stored in this register is then moved to the first R0 parameter register. The fclose C function is used to close the read file. A new output file is created and opened using the fopen C function. A for loop is used to write the contents of the array to the output file. The fprintf C function is used to write the integers to the array in the output file with a line containing only one integer. When the loop counter finally becomes equal to the size of the array, the for loop ends.

```
// The input file is closed is using fclose C function.
// A new output file is created, and it is opened using the fopen C function.
// The file pointer of the output file is returned to register r0.
write:
        mov r0, r4
        bl fclose

        ldr r0, =writefile
        ldr r1, =writemode
        bl fopen
        mov r4, r0
        mov r5, #0
        ldr r6, =array

// A for loop is used to write the contents of the array to the output file.
// The fprintf C function is used to write the contents line by line.
// When the loop counter reaches the size of the input file, the loop terminates.
writeloop:
        cmp r5, r9
        beq end
        mov r0, r4
        ldr r1, =printformat
        ldr r2, [r6]
        bl fprintf
        add r5, r5, #4
        add r6, r6, #4
        b writeloop
```

Challenges Faced & Problems Encountered

Reading and writing to files: When we opened the input file using the fopen C function, we didn't initially realize that the input file pointer was being returned to the R0 register. So when we tried to use the fscanf function below to read the content of the input file, our program was not reading the values correctly. Sometimes it was reading in the hexadecimal representation of numbers, and sometimes it was reading in random numbers. Storing the file pointer in the R0 register and using it as a parameter for fscanf and fclose solved the problem.

Selection sort algorithm: Implementing the sorting algorithm presented the toughest obstacles in this project. At first, the algorithm wasn't working at all, the values were not getting stored in the proper format causing the output file to have completely different values from the input file. We then assumed that the problem came from loading and storing the values. To account for this we used a different syntax for these commands: strsb and ldrsb. After implementing these, the program ran fine except for when the inputs were larger than 8 bits. The project description stated that the program should be able to handle up to 32 bits so we had to find a way around this. We now thought the issue was with scanning in the numbers, however, after checking what numbers were scanned in before they got sorted we found that the scan function wasn't the issue. We also knew it couldn't have been the amount of storage allocated because we had room for up to 1024 integers. After a few hours of debugging, we found that the problem came from the *increment* branch in the sort algorithm. We were offsetting the value of R5 by one each time when we should have been offsetting by 4 since we're dealing with long data types. Once this was changed the program ran as expected.