Project 2 – Sort

ECE251 – Computer Architecture

Prof. Billoo

The Cooper Union

Spring 2020

Daniel Mezhiborsky

Introduction

The project description required an ARM assembly program to be written to satisfy the following requirements:

- Read a file containing 32-bit integers on each line, with at most 100 lines
- Sort the numbers
- Write the numbers to a separate file

Given the constraints, the following behavior was implemented:

- Read a file containing 32-bit integers on each line. The filename must be specified If non-integer content is detected, or if the file contains more than 1000 integers, an error will be printed, and the program will exit.
  - The integers may be signed or unsigned, and this must be specified.

- Sort the numbers using a selection sort. The sorting direction may be specified.

- Write the numbers to a file in the same directory as the executable named “sorted.txt”.

- The parameters are passed as command-line arguments. Usage instructions are built-in.

The following report contains a breakdown of the source code with key methodology explained. All code referenced is from the included sort.S source file.

Key Specifications

- Language: 32-bit ARM assembly, EABI5 version 1
- Assembler: cross-gcc (arm-linux-gnueabi-gcc) 7.4.0
- Environment: qemu-arm 2.11.1 on Ubuntu18.04.4 LTS

Files

- sort.S – main source file
- Makefile
- README
- test_random.py – basic fuzzer

External Functions

Six external functions were used explicitly:

- **fscanf()**: Used to read the input file. Its return codes are important for determining EOF and improper input data.
- **fprintf()**: Used to write the output file.
- **fopen()**: Used to open the file stream.
- **fclose()**: Used to close the file streams.
- **__errno_location()**: Used to find the location of errno. Determined from compiled C file.
- **strerr()**: Used to find the error message corresponding to a given errno.

Data Section

```
.data
/* ------------------------------------------------------ */
file_pointer_read:      .space 4
file_pointer_write:     .space 4
number_list:            .space 404
read_filename_ptr:      .space 4
write_filename:         .asciz "sorted.txt"
read_mode:              .asciz "rb"
write_mode:             .asciz "wb"
scanf_fmt:              .asciz "%dz"
scanf_fmt_unsigned:     .asciz "%uz"
printf_fmt:             .asciz "%d\n"
printf_fmt_unsigned:    .asciz "%u\n"
error_file_open_read:   .asciz "An error was encountered while reading the file \'%s\': "
error_errno_append:     .asciz "%s\n"
error_long_file_msg:    .asciz "File must be 100 integers at most, one integer per line.\n"
error_not_int_msg:      .asciz "Only integers allowed!\n"
usage:                  .asciz "Usage: ./sort.out [filename] [r or n] [s or u]\nr==reverse, n==nor
mal (ascending) ||| s==signed, u==unsigned\n"
done:                   .asciz "Done. Check output [sorted.txt]\n"
size_read:              .space 4
flags:                  .space 4
temp:                   .space 4
/* ------------------------------------------------------ */
```

The names in the data section are expressive. Clarification:

- **read_filename_ptr**: pointer to the filename location.
- **error_errno_append**: formatting fragment used in error_bad_file
- **flags**: contains number signifying reverse/normal and signed/unsigned configuration
- **temp**: temporary storage

Breakdown: main

```
main:
    push {r4-r12, lr}

    cmp r0, #4                  @ are there at least three args?
    bne error_missing_args
    bleq process_args

    bl open_file_read           @ open and read input file
    bl sort_list
    bl open_file_output         @ open and write output file

    pop {r4-r12, lr}
    bx lr
```

-> r4-r12 are preserved, as they are caller-owned. Argument count is checked and then functions are called to parse the arguments, read the list, sort the list, and write it back out as needed.

Breakdown: Argument Processing

```
error_missing_args:
    ldr r0, =usage              @ print usage and leave
    bl printf
    mov r0, #1
    b exit
```

-> Any problem with the arguments branches here. This prints usage and exits.

```
process_args:
    push {lr}

    ldrge r2, [r1, #4]          @ load arg 1: filename
    ldrge r3, =read_filename_ptr
    str r2, [r3]

    ldr r2, [r1, #8]            @ load arg 2: reverse sort flag
    ldrb r2, [r2]
    cmp r2, #114
    blne process_check_reverse
    ldreq r3, =flags
    moveq r2, #1
    strge r2, [r3]
```

-> The filename is loaded and saved. The reverse sort flag is loaded and compared against ASCII ‘r’. 1 is added to flags to indicate the sort should be reversed. **process_check_reverse** is called if the flag is not ‘r’ to make sure that it’s ‘n’ instead.

```
process_get_sign:
    ldr r2, [r1, #12]           @ load arg 3: unsigned list flag
    ldrb r2, [r2]
    cmp r2, #117
    blne process_check_signed
    ldreq r3, =flags
    ldreq r2, [r3]
    addeq r2, #10
    strge r2, [r3]

    pop {lr}
    bx lr
```

-> The list sort order flag is loaded and compared against ASCII ‘u’. **process_check_signed** is if the flag is not ‘u’ to make sure that it’s ‘s’ instead. 10 is added to flags to indicate the list is unsigned. This returns to main when done. The scheme for encoding the flags creates four combinations (0,1,10,11) in a single number that can be easily checked in later routines without actually having two separate flags.

```
process_check_reverse:
    cmp r2, #110                @ since the flag wasn't r, let's make sure it's n
    blne error_missing_args
    b process_get_sign          @ skip to next arg

process_check_signed:
    cmp r2, #115                @ since the flag wasn't u, let's make sure it's s
    blne error_missing_args
    pop {lr}                    @ back to main; we're done
    bx lr
```

-> The aforementioned check functions. One returns to process the next argument, and the other returns to main.

-> Development note: It doesn't work here to link to these labels when branching and to return to that link. These labels perform comparisons – the results of which will interfere with the remaining conditional instructions from the preceding comparison.

Breakdown: Reading the Input File

```
open_file_read:
    push {r5, r6, lr}           @ fopen the input file
    ldr r0, =read_filename_ptr
    ldr r0, [r0]
    ldr r1, =read_mode
    bl fopen

    cmp r0, #0              @ make sure that was successful (not null pointer)
    ldreq r0, =error_file_open_read
    ldreq r1, =read_filename_ptr
    ldreq r1, [r1]
    bleq printf
    beq error_bad_file

    ldr r1, =file_pointer_read      @ save the pointer
    str r0, [r1]
```

-> The file is opened. The pointer is checked to make sure this was successful, and a branch to **error_bad_file** will occur in the case of a null pointer. **error_bad_file** will exit and prevent the potential null pointer dereference.

```
read_file:
    ldr r5, =number_list            @ load addr of the number_list block to which we will rea
d
    mov r6, #0
read_file_loop:
    cmp r6, #400                @ make sure we haven't exceeded 100 lines
    bgt error_long_file

    ldr r0, =file_pointer_read
    ldr r0, [r0]

    ldr r1, =flags              @ choose signed or unsigned read
    ldr r1, [r1]
    cmp r1, #10
    ldrlt r1, =scanf_fmt
    ldrge r1, =scanf_fmt_unsigned

    mov r2, r5              @ r2=base address of number_list plus offset (loop counter)
    add r2, r6
    bl fscanf
```

-> The loop is introduced by setting the counter to zero and loading the address to which to save the integers. A check for a file longer than 100 lines is included (the value of 400 is explained below). The signed or unsigned version of the format string for fscanf is selected, and fscanf is called.

```
    cmp r0, #0              @ make sure fscanf worked as expected
    addgt r6, #4
    bgt read_file_loop
    blt read_file_end
    beq error_not_int
```

-> The counter is incremented by 4 because each integer is four bytes away from the next in memory. This makes it so that an extra multiplication instruction and register are not needed, and is useful across the entire program. Checks are included for the end of the file (return code = -1), a non-integer entry (return code = 0), and a successful read (which will repeat the loop; return code > 0).

```
read_file_end:
    ldr r5, =size_read          @ end loop; close file and return to main
    str r6, [r5]
    ldr r0, =file_pointer_read
    ldr r0, [r0]
    bl fclose
    pop {r5, r6, lr}
    bx lr
```

-> This loop end will save the size of data read, close the file, and return to main.

```
error_not_int:
    ldr r0, =error_not_int_msg      @ Not an integer
    bl printf
    mov r2, #3
    b exit
error_long_file:
    ldr r0, =error_long_file_msg        @ File too long
    bl printf
    mov r1, #2
    b exit
```

-> Two error handler routines. Both print a message and exit, as well as return an associated return code.

Breakdown: Sort

-> The sort section is an assembly implementation of the following C selection sort algorithm:

```
for (int n1=0; i<size_read; n1=n1+4) {
    min = n1+number_list;
    for (int n2=0; n2<offset-n1, n2=n2+4) {
        if (*(number_list+n1+n2) > *(min)) {
            min = number_list+n1+n2;
        }
    }
    temp1 = *min;
    temp2 = *(n1+numlist);
    *min = temp2;
    *(n1+numlist) = temp1;
}
```

-> A slight change is the inclusion of procedures for signed/unsigned and ascending/descending sort.

```
/*    ~~~ Selection sort ~~~

    r0=outer_counter
    r1=inner_counter
    r2=n2
    r3=min
    r4=size_read
    r5=inner_loop_max
    r6=numbe_list+n1+n2
    r7=deref_min
    r8=number_list
    r9=deref_number_list+n1+n2
    r10=swap_temp1
    r11=swap_temp2
    r12=flags
                */
sort_list:
    push {r4-r12, lr}       @ sort intro
    mov r0, #0
    mov r1, #0
    ldr r4, =size_read
    ldr r4, [r4]
    ldr r8, =number_list
    ldr r12, =flags
    ldr r12, [r12]
    sort_loop1:         @ <<outer loop>>
        cmp r0, r4          @ n1<size_read
        bge sort_end
        add r3, r8, r0          @ min=n1+number_list
```

```
        mov r1, #0
        sort_loop2:         @ <<inner loop>>
            sub r5, r4, r0          @ size_read-n1
            cmp r1, r5          @ n2<side_read-n1
            bge sort_loop1_end

            mov r6, r8          @ = number_list address
            add r6, r0          @ += n1
            add r6, r1          @ += n2
            ldr r9, [r6]            @ deref number_list+n1+n2

            ldr r7, [r3]            @ deref min

            cmp r12, #0         @ branch to correct signing and ordering
            beq new_minimum
            cmp r12, #1
            beq new_minimum_rev
            cmp r12, #10
            beq new_minimum_unsigned
            cmp r12, #11
            beq new_minimum_unsigned_rev
        sort_loop2_end:
            add r1, r1, #4
            b sort_loop2
    sort_loop1_end:         @ swap elements
        ldr r10, [r3]
        ldr r11, [r8, r0]
        str r11, [r3]
        str r10, [r8, r0]

        add r0, r0, #4
        b sort_loop1
sort_end:
    pop {r4-r12, lr}
    bx lr
```

-> These routines were observed in Ghidra to decompile to a comprehensible, equivalent function to the given C selection sort.

```
new_minimum:                @ ascending signed
    cmp r9, r7
    movle r3, r6
    b sort_loop2_end
new_minimum_rev:            @ descending signed
    cmp r9, r7
    movge r3, r6
    b sort_loop2_end
new_minimum_unsigned:           @ ascending unsigned
    cmp r9, r7
    movls r3, r6
    b sort_loop2_end
new_minimum_unsigned_rev:       @ descending signed
    cmp r9, r7
    movhs r3, r6
    b sort_loop2_end
```

-> These routines set the new minimum. One of the four is called based on the command-line options.

Breakdown: Write

```
/* WRITING OUTPUT--------------------------------------------------------------------------
open_file_output:
    push {r4-r6, lr}            @ open output file
    ldr r0, =write_filename
    ldr r1, =write_mode
    bl fopen

    cmp r0, #0              @ make sure that was successful
    ldreq r1, =write_filename
    beq error_bad_file
    ldr r1, =file_pointer_write
    str r0, [r1]
```

-> This routine is analogous to the previously-described **open_file_read.**

```
print_list:
    ldr r5, =number_list            @ print list loop intro
    ldr r4, =size_read
    ldr r4, [r4]
    mov r6, #0
print_list_loop:
    ldr r0, =file_pointer_write     @ choose signed or unsigned, print, and iterate
    ldr r0, [r0]
    ldr r1, =flags
    ldr r1, [r1]
    cmp r1, #10
    ldrge r1, =printf_fmt_unsigned
    ldrlt r1, =printf_fmt
    ldr r2, [r5, r6]
    cmp r6, r4
    beq print_list_end
    bl fprintf
    add r6, #4              @ counter
    b print_list_loop
```

-> The address of the list and the size read are loaded, and the loop iterates until it has printed all of the numbers in the list. The signed or unsigned format is chosen according to the command-line arguments.

```
print_list_end:
    bl fclose
    ldr r0, =done               @ print message and return to main
    bl printf
    pop {r4, r5, r6, lr}
    bx lr
```

-> The file is closed, the “done” message is printed, and the routine returns to main. This is the last routine before main returns to exit().

SPECIAL

```
// General fopen error
// Prints message associated with current errno
error_bad_file:
    bl __errno_location         @ gcc told me
    ldr r0, [r0]
    bl strerror
    mov r1, r0
    ldr r0, =error_errno_append
    bl printf
    bl exit
```

-> This function retrieves the associated error message with the current errno and prints it. The __errno_location reference method was found using a test C function and gcc, compiling to assembly.

Challenges Faced and Notes

- **sort implementation**
  - I wrote an implementation of selection sort that was fundamentally unlike C nested-for-loop implementation, but I didn't realize that I couldn't treat the implementation I wrote the same way as the C version. My frame of thinking made it such that I had a very hard time debugging it.
  - Various corner cases would fail, and when I would change something to attempt to fix them, other corner cases would fail instead. One pervasive problem was the unintentional duplication of elements in the list.
    - The solution to these challenges was really to write a more coherent implementation, so I rewrote it to be as close as possible to the C version. This was pretty much immediately successful.
- **testing**
  - It became tiring and difficult to test various cases, so I wrote a simple Python fuzzer program. The final binary of the sort was tested with about 40,000 iterations of randomly sized lists of random numbers, using combinations of normal and reverse soring as well as signed and unsigned lists. Zero iterations failed.
  - Various other test cases were checked by hand:
    - Non-integer inclusions: PASS
    - List > 100 elements: PASS
    - List of zero elements: PASS (returns empty output)
    - Single-element list: PASS
    - Input file doesn't exist: PASS
    - Input file unreadable: PASS
    - Output file unwritable: PASS

- Signed and unsigned lists
  - The integers in the list are interpreted as twos-complement numbers. They are translated directly into binary form. In *unsigned* mode, a -1 in the list, for example, is treated as 4,294,967,295.
    - This is the intended behavior.
- ldr pseudo-instruction
  - the form ldr, r0, =some_data_item is used frequently for brevity. It is expanded by the compiler, and is described here: http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.kui0100a/armasm_babbfdih.htm

Acknowledgements

- Andrey Akhmetov, for advice on Ghidra
- Jonathan Lu, for testing assistance
- cppreference.com
- infocenter.arm.com
- ARMv7 Architecture Reference Manual