Project 1 – Strings

ECE251 – Computer Architecture

Prof. Billoo

The Cooper Union

Spring 2020

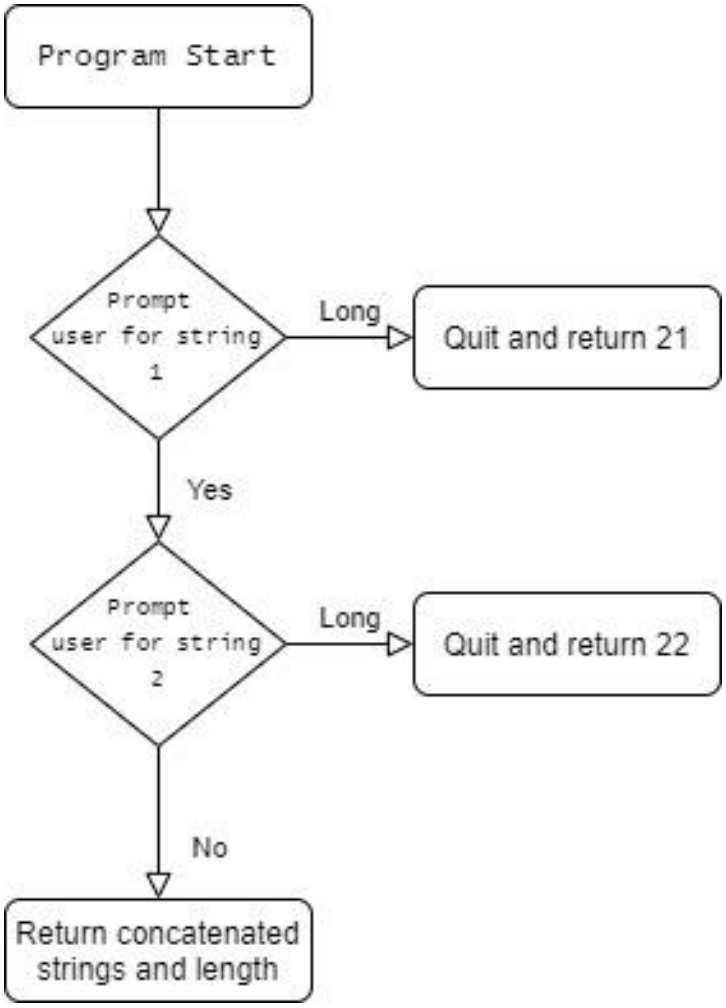
Daniel Mezhiborsky

Introduction

The project description required an ARM assembly program to be written to satisfy a few requirements:

- Accept ASCII user input as two strings
- Quit if the user inputs a string that is over 10 characters and return a specific error code
- Concatenate and print the strings, and provide their combined length

The following structure was followed:



The following report contains a breakdown of the source code with key methodology explained. All code referenced is from the included strings.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 Ubuntu 18.04.4 LTS

Files

- strings.S – main source file
- Makefile
- README

External Functions

Two external functions were used explicitly:

- **fgets**: Safer alternative to gets and scanf. Takes argument for number of characters to read, effectively preventing buffer overflow errors. Will read and include whitespace characters.
- **printf**: Useful formatting options for printing strings and numbers.

Labels – Summary

main:
  - get both strings with getstrings
  - process string1 with findlength
  - print the concatenated strings
  - process the length of the concatenated string.

getstrings:
  - prompt the user and collect string input
  - check that the string is within length bounds, handle with longstring if needed

longstring:
  - clean up, return appropriate errors and exit in the case of the string being too long

findlength:
  - finds the newline in string1 and removes it
    - length of string1 available as consequence, but unused

These four labels/functions, when combined with fgets and printf, accomplish the requirements set out in the project description.

Data Section Breakdown

- **string1, string2**:
  - 16 bytes each of space
  - filled with zeros
- **longstring_msg:**
  - Error message for a long string
  - %d parameter is 21 or 22 error code
- **prompt_msg:**
  - User input message
  - %d parameter is string number 1 or 2
- **number_fmt:**
  - String length message
  - %d parameter is combined string length
- **concat_fmt:**
  - Concatenated string message
    - Very important to have a format string here because formatting characters could otherwise be injected and parsed
  - two %s parameters are string1 and string2

Breakdown: main

```
main:
    push {lr}           @ save lr
```

-> Program begins, lr is put on the stack so it can be used after future function calls.

```
    mov r5, #0          @ r5 indicates string1 or string2 (0 or 1)
    ldr r4, =string1    @ load address of string1 into r4
    bl getstrings       @ get string1

    mov r5, #1          @ indicate string2
    ldr r4, =string2    @ load address of string2 into r4
    bl getstrings       @ get string2
```

-> r5 is used to indicate the current string. The string addresses are loaded and getstrings is called. getstrings will return to main once it is done, having read and stored the strings and checked in case they are too long.

```
    ldr r4, =string1    @ load address of string1 into r4
    bl findlength       @ go to findlength
```

-> findlength is used to remove the newline at the end of string1.

```
    ldr r0, =concat_fmt @ load concatentation printf format
    ldr r1, =string1    @ load string1 address arg
    ldr r2, =string2    @ ^ string2
    bl printf           @ print concatenated string
```

-> The strings are concatenated by printing them one after the other.

```
    sub r0, #38         @ subtract 38 from printf returned count
    mov r1, r0          @ set r1 to the count
    mov r4, r0          @ save the count in r4 for exit code
    ldr r0, =number_fmt @ load printf format
    bl printf           @ print number of characters
```

-> r0 contains the number of characters printed - a return from printf, which was called on the preceding line.

-> 38 is subtracted from r0 because this is the number of printable characters in the formatting string that are not the strings themselves (and the newline left at the end of string2).

-> r0 is copied into r1 as an argument for printf.

-> r0 is copied into r4 to preserve it across the upcoming printf call (r4 is a callee-owned register).

-> the formatting string address is loaded and printf is called.

```
    mov r0, r4          @ set return code
```

-> The exit status is copied into r0.

```
    pop {lr}            @ get lr
    bx lr               /* bye! */
```

-> lr is retrieved and the program exits (branches to libc exit).

Breakdown: getstrings

```
    push {lr}           @ save lr
```

-> lr back to main is put on the stack.

```
    ldr r0, =prompt_msg @ load address of prompt message
    mov r1, r5          @ load string number
    add r1, #1          @ 0->1 and 1->2 for user friendliness
    bl printf           @ print prompt
```

-> User is prompted to enter the string (with string number).

```
    mov r0, r4          @ copy r4 (string address) into r0
    mov r1, #13         @ fgets number of chars to read
    ldr r2, find_stdin  @ get stdin address
    ldr r2, [r2]        @ load value of stdin
    bl fgets            @ call fgets to get user input
```

-> string-to-be-written address, number of chars to read (13), and stdin are prepared for fgets.

-> 13 chars are read because this allows for a string up to 12 chars (plus a newline) to be written (12 instead of 13 should work just as well here). We will use this in the following lines.

-> We don't care if the string was truncated if it was too long, as we're going to error out soon anyways in that case.

-> fgets is called. Down the line, this causes stdout's buffer to be flushed and the user prompt is displayed (there's no newline in the prompt string!).

```
    ldrb r0, [r4, #11]  @ load single byte into r0: 12th char of string
    cmp r0, #0          @ check if that byte is nonzero (not NULL)
    bgt longstring      @ if so, exit with longsQtring
```

-> If the string is 10 characters or less, the 12th character must be zero, as there are 16 bytes of space in each string.

-> This means the longest string, at 10 characters, will have nonzero data at the first 11 bytes (accounting for the newline).

If there is nonzero data past the first 11 bytes, the string was longer than 10 characters!

-> If long-string evidence is found, branch to longstring to handle the exit.

```
    pop {lr}            @ get lr
    bx lr               @ return to main
```

-> Retrieve and follow the lr back to main.

Breakdown: longstring

```
    add r5, #21             @ get either 21 or 22 error code
```

-> r5 still contains 0 or 1, corresponding to the current string number. 21+r5 will provide the correct return code. This is left in r5.

```
    ldr r0, =longstring_msg @ load error message
    mov r1, r5              @ copy error code
    bl printf               @ print error
```

-> The error message is printed for the user.

```
    mov r0, r5              @ set return code
    pop {lr}                @ discard getstrings's lr to main
    pop {lr}                @ get exit() lr
    bx lr                   /* bye! */
```

-> The return code is copied into r0.

-> lr is popped off the stack twice to retrieve the glibc exit lr.

-> The program exits.

Breakdown: findlength

```
    ldrb r1, [r4]       @ load character at r4+i
    cmp r1, #10         @ compare with ASCII newline
```

-> a single character at r4 is loaded into r1. Initially, r4 is the starting address of the string

-> this character is compared against the newline (number 10 ASCII)

```
    moveq r3, #0        @ if equal, set r3 to 0
    streqb r3, [r4]     @ if equal, store zero at that location
    bxeq lr             @ if equal, return to main
```

-> if equal, a single zero/NULL byte is stored at that address, shortening the string by one character by removing the newline.

-> return to main

```
    add r4, #1          @ increment address pointer by 1 byte/char
    b findlength        @ ^ otherwise, loop
```

-> Otherwise, the character is incremented, and we loop back to the start of findlength.

Challenges Faced

- **scanf, fgets and how to use stdout**
  - Given scanf's behavior of stopping on whitespace, I decided that a different function would be more appropriate. After reading through libc reference documentation, I decided that fgets would be the best alternative because it stops only on newlines and has a parameter for the number of characters to read. It is also fairly lightweight.
  - fgets takes an input file/stream argument. I needed the specific syntax for how to pass stdout to fgets in ARM assembly, but I had trouble finding how to do this. I did find examples for x86 assembly.
    - To solve this problem, I wrote test code in C and compiled it to assembly. From this I isolated the syntax and was able to continue.
- **.word, .space, and memory allocation**
  - I did not have a good grasp of how much memory was allocated for a .word directive.
    - The number following .word is not the number of bytes allocated – once I understood this, I decided it would be best to look for a memory allocation directive appropriate for the situation.
    - This caused issues where one string would overflow into the memory space of the next or other data. Using the debugger, particularly with x/32s, which views 32 bytes as strings starting from an address, helped me to understand the memory space.
      - Reading ARM documentation and stackoverflow yielded .space as a useful directive. The number of bytes is specified and space filled with zeros is allocated.
- **stdout buffering**
  - Along with Jon Lu, I was confused by the buffering behavior of stdout.
    - As I found out, stdout/stdin buffering and flushing is not really guaranteed behavior, and is implementation-specific.
      - Running two fgets calls in a row where the first fgets stopped reading the string because of the character limit led to significant issues, as it would continue reading the leftover characters in stdin.
      - Understanding the stdout flushing behavior on our system – that stdout is flushed when switching from output to input with stdin, helped to understand

why some text printed by printf not containing a newline did not appear until fgets was called or a newline was printed.
        - The solution to this was really just to be careful, because the situation that causes this is one where we error out anyway (the user entered a very long string). Test routines and debugging had to be written and performed with this in mind.
- **Segfault opacity**
    - One significant issue in debugging and running code was that segfaults happening inside external function calls were opaque and difficult to trace.
        - This was generally a large source of frustration. The strategy I employed was to just try as much of the memory accesses fgets/printf might perform – dereference the pointers I pass, examine the memory space, etc.
        - This is still somewhat of an annoying problem. gdb having the source available to follow the external functions would be helpful – will investigate this.

General Notes

- I discovered that printf returns the number of characters printed after the program was already complete. This allowed for a nice optimization where my findlength function only needed to be called once, to remove the newline from string1.
    - I hoped I could use “%s\b%s” to print the two strings with no newline in between, but it turns out \b cannot be used to undo a newline.
    - Now, findlength only runs once, saving n memory accesses and n comparisons for a string2 of n length. The concatenated length is calculated by simply subtracting a constant offset from printf’s return, which is already available as a result of calling printf to concatenate the strings in the first place.
- getstrings checks for non-whitespace to ensure compliant string length instead of the presence of a character greater than a newline. This is a deliberate choice to preserve the functionality of the program for ASCII characters with a decimal value less than 10.
- fgets does not read the newline into memory if it runs into its length limit (it never gets to it in the stream!). This would be an issue because some strings would be different than others, but since I use a spot check for the length and error out directly as a result of that, this issue is avoided.

Acknowledgements

- Andrey Akhmetov, for advice and information on the details of ARM branching instructions and their behavior with THUMB.
- cppreference.com
- infocenter.arm.com
- ARMv7 Architecture Reference Manual