CSE30 - PA 4

Strings, recursion and stacks!

1 Introduction

This assignment will give you a lot more practice with assembly programming. You will work heavily with procedures and subroutines as well as exploring recursion in ARM.

2 Obtaining the starter code

To get the starter code for the assignment click on this link to assignment.

Your repo should be automatically populated with the starter code upon creation. However, if that doesn't happen you can obtain the starter code from the following repo: lab04 starter code

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.

To create your repo follow the same naming conventions as instructed in the previous labs.

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!

3 Strings in ARM

3.1 Substrings

For this part of the assignment you will implement a function that tests if either of two strings are substrings of each other. A string is a substring of another string if it is contained anywhere in the other string. For example:

- "CSE30" is a substring of "In hexspeak, CSE30 is 0xC5E30"
- "raspberry" is not a substring of "RASPBERRY PI"
- "pi" is a substring of "pi", but not "ip"

The function has the following signature:

```
int substring(char * s1, char * s2)
```

Implement this function in ARM and place your implementation in the file `substring.s`. Do not change the function signature. Write test code in C in the file `substring_test.c` to test your ARM implementation.

Your `substring` function should return 1 if either string is a substring of the other, and 0 otherwise. This means that the function should always have the same output if the arguments are switched. If either of the two input strings is NULL, your function should return "0". The empty string (string of length 0) will be considered a substring of any other string (provided that neither of them are NULL pointers).

You are not allowed to use the `strstr` or `strcmp` function, or any other built-in C functions or library functions which determine whether one substring is part of or equal to another one. Submissions that make use of such functions will not receive credit. You are permitted, however, to use the `strlen` function if you find that useful.

3.2 String to Int

The function `atoi()` is used to convert a string of digits to integers (e.g., "25" to 25). This function takes a string, interprets it as a decimal representation of an integer, and returns the corresponding integer. Your task for this part of the assignment is to implement a function `str_to_int()` with similar behavior. The function has the following signature:

```
int str_to_int(char * s, int * dest)
```

If the input is properly formatted, the function should place the integer represented by `s` at the location pointed to by `dest` and return 1 to indicate success. If the translation process was unsuccessful, you should return 0. Your function should resolve both positive and negative numbers. That is, if the input string "s" starts with "-", your result should be negative.

The function should indicate failure (by returning 0) in the following circumstances:

- Either of the arguments is NULL; or
- The string pointed to by `s` is not a valid representation of an integer in decimal form.

In order for `s` to be a valid representation of an integer in decimal form,

- it must contain at least one digit (this excludes the null terminator);
- it may contain one negative sign, but only at the very beginning of the string; and
- it may not contain any characters besides the digits 0-9 and the negative sign.

You will not need to check for overflow during the computation of the value. We will not test your function on valid integer strings whose value does not fit in 32 bits. However, your function should correctly handle all strings representing an integer which *should* fit in a 32-bit integer.

The function must be written both in C and ARM assembly.

We've provided the files "`str_to_int_ARM.s`" and "`str_to_int_test.c`" for you. Please create the file "`str_to_int.c`" which should contain your C implementation. Please write your C function in the "`str_to_int.c`" file, and your ARM implementation of the same function in the `.s` file. Test your code thoroughly in the file "`str_to_int_test.c`".

We've provided you with a function signature. Please do not change it.

You are not allowed to use `atoi`, `strtol`, or any other built-in C functions or C library functions that convert strings to integral types. Submissions which make use of these functions will not receive credit.

4 Recursive Functions and the Stack

In this section, you are given a problem, and you are expected to solve it recursively using assembly programming. In addition, we you will learn how to use the stack to store local variables. We have given you C implementations of a recursive function, and you are required to translate it to ARM assembly. In addition, we have placed the following constraints on your solutions: you may only use registers r0 - r7 to store values, and you may NOT use `malloc` or `calloc`. If you run out of space to store values, you must use stack locations to store temporary values or local variables. Below, we give a short tutorial on how this is done.

4.1 Reserving Stack Space for Local/Temporary Values

In the ARM architecture, the stack pointer is stored in register `sp`. It holds the memory address of the lowest memory location of the current stack frame. As we learned in class, ARM has a full descending stack. This means that whenever a function is called, a stack frame is allocated beneath the calling function's stack frame for the new function's local variable space. So far, we have used stack frames to store the values of r4-r11 from the calling function's context. This is the purpose of the `push` and `pop` pseudo-instructions, which implicitly decrease the stack pointer by an amount large enough to store the register values. If we need additional stack space for local variables, we need to decrement the stack pointer manually.

For example, below we have a function which reads two integers from `stdin` and returns their sum:

```
int read_and_sum() {
    int l1, l2;
    scanf("%d%d", &l1, &l2);
    return l1 + l2;
}
```

The `scanf` function is the reverse of `printf`, reading input and interpreting, rather than formatting data and outputting it. The two "%d" tokens tell the program to interpret the next two space-separated inputs as integer, and the other arguments to `scanf` tell the program where in memory to store the values. The corresponding ARM code would be:

```
    .data
    .align 3
scan_ints:
    .asciz "%d%d"

    .text
    .align   2
    .global read_int
    .type    read_int, %function
read_int:
    push {r4-r11, ip, lr}
    sub sp, sp, #8       @ reserve 8 bytes for the 2 local integers

    ldr r0, =scan_ints   @ format string
    mov r1, sp           @ address of l1
    add, r2, sp, #4      @ address of l2
    bl scanf

    ldr r0, [sp]         @ read l1
    ldr r1, [sp, #4]     @ read l2
    add r0, r0, r1

    add sp, sp, #8       @ de-allocate 8 bytes, restore sp to
                         @ value after push
    pop {r4-r11, ip, lr}
    bx lr
```

As you can see, it is very simple to reserve stack space: simply decrement the stack pointer at the beginning of your function by an amount large enough for the space you need, restore this value before you return. You should decrement the stack pointer after you push, and increment it before you pop. The addresses of these local spaces will be measured in offsets from sp.

There is one additional caveat. Normally, the ARM ISA requires that the stack pointer be 8-byte aligned. This means the size of your stack frame should be a multiple of 8, and it should begin and end on 8-byte aligned addresses. So you may need to reserve extra space that you don't end up using.

4.2 Binary Search in ARM

Now that we know how to use the stack, let's start coding. Implement the `binary_search()` function below in ARM assembly. Your solution should find and return the index of the desired element in the sorted array (if it exists in the list) recursively, as described below. This source code is provided in `binary_search.c`. Please put your implementation in `binary_search_ARM.s`, and your testing code in `binary_search_test.c`.

Listing 1: binary_search()

```
/* int binary_search(int *data, int toFind,
*                    int start, int end)
*  Args:
*       int *data: address of element 0 of an integer
*                  array sorted in ascending order.
*       int toFind: integer to find index of
*       int start: index of start of search area
*       int end: index of end of search area
*  Returns:
*       int: index of element in array
*                 OR
*            -1 if element not in array
*/
int binary_search(int *data, int toFind,
                  int start, int end)
{
    int mid = start + (end-start)/2;
    if (start > end)
        return -1;
    else if (data[mid] == toFind)
        return (mid);
    else if (data[mid] > toFind)
        return binary_search(data, toFind, start, mid-1);
    else
        return binary_search(data, toFind, mid+1, end);
}
```

Note that we do not require that you check whether `data` is NULL. This function will NOT be tested on NULL inputs. (This is because in practice, since this method is recursive, checking whether the array is NULL produces many redundant checks. So any function using this method would have to check wether the array is NULL *before* passing it to the binary search function.) Nor will we test the function for `start` or `end` values which are out of range for the size of `arr` (though there are corner cases where you may need to handle such inputs anyway).

4.3 Majority Element

Below, we have given the pseudocode for a recursive algorithm that finds the count of the majority element in an array of integers. The majority element of an array is the value that appears in more than half of the entries. If the array doesn't have a majority element, the function simply returns a count of "0".

```
procedure majority_element(arr[0...(n-1)])
    // arr: array of n integers
    if n == 0:
        return NO_MAJORITY
    if n == 1:
        return arr[0]

    left_maj = majority_element(arr[0...(n/2)])
    right_maj = majority_element(arr[(n/2)...(n-1)]

    if left_maj != NO_MAJORITY:
        c = count(arr[0...(n-1)], left_maj)
        if c > n/2:
            return left_maj

    if right_maj != NO_MAJORITY:
        c = count(arr[0...(n-1], right_maj)
        if c > n/2:
            return right_maj

    return NO_MAJORITY
```

Note the helper function `count`, which simply counts the number of times a specified value occurs in an array of integers. We recommend that you implement this as a helper function, but this is not a requirement; you may perform the counts within the function explicitly if you wish.

In the file `majority_count.c`, we give a C implementation of this function with slightly different behavior. Note that it returns the *count* of the majority element rather than the majority element itself, though the majority element can be stored in a secondary location specified by the caller. For this section you must implement the `majority_count()` function in ARM in the file `majority_count_ARM.s`.

Listing 2: majority_count()

```
/* int majority_count(int * arr, int len, int * result)
 * Args:
 *      int * arr: address of element 0 of an integer array
 *      int len: the length of "arr"
 *      int * result: if not NULL, the location at which the
 *          value of the majority element should be stored.
 * Return:
 *        int: the count of the majority element in the
 *            array. 0 if no majority element. An array
 *            of length 0 has no majority.
 */
int majority_count(int * arr, int len, int * result) {

    if(len == 0) {
        return 0;
    }
    if(len == 1) {
        if(result) {*result = arr[0];}
        return 1;
    }

    int left_majority, right_majority, c;
    int left_majority_count = majority_count(arr, len/2,
        &left_majority);
    int right_majority_count = majority_count(arr+len/2,
        len-len/2, &right_majority);

    if(left_majority_count) {
        c = count(arr, len, left_majority);
        if(c > len/2) {
            if(result) {*result = left_majority;}
            return c;
        }
    }
    if(right_majority_count) {
        c = count(arr, len, right_majority);
        if(c > len/2) {
            if(result) {*result = right_majority;}
            return c;
        }
    }

    return 0;
}
```

If `result` is not NULL, then the value of the majority element (not its count) should be stored at that address after the function returns. This means that a NULL value is an acceptable input for this argument, if the user does not wish to know the actual majority element, but merely its count.

As in the previous section, we do not check whether `arr` is NULL. Since this function is recursive, this would lead to many redundant checks. We will not test this function for NULL values of arr. (So any function which calls this method would have to ensure that `arr` is not NULL *before* passing it to `majority_count`.) Nor will we test this function on values of `len` which are larger than the actual space allocated for `arr` (nor will we test negative values of `len`).

You may notice that this particular C implementation has room for improvement. Feel free to make optimizations, so long as the functionality remains the same, and so long as you adhere to the constraints we have placed for this part of the assignment.

5 Submission

You should submit your solution and test code to the PA4 assignment on gradescope. You may submit as many times as youd like, but only the final submission will be recorded. Submit the following files individually:

- substring.s
- substring_test.c
- str_to_int.c
- str_to_int_ARM.s
- str_to_int_test.c
- binary_search_ARM.s
- binary_search_test.c
- majority_count_ARM.s
- majority_count_test.c

We will use our own Makefile, so while you may make modifications to the supplied Makefile for your own purposes, we will not be using it for grading, and you don't have to turn it in.

Late assignments will not be accepted, so make sure to turn in your work by the deadline! Moreover, remember not to submit new versions of your homework after the deadline – new submissions overwrite old ones, and thus you will have submitted your assignment late.