ECE 1181: Microcontrollers and Embedded Systems Lab
Spring 2024 – Professor Joe Camp

Lab 8: Macros and System Calls (4/11 or 4/12)
(Due before the start of the next lab – 4/18 or 4/19)

Reminder: Each lab report should have the following.
1. Name and SMU ID
2. Lab Session number & objective of the lab assignment
3. Description of the program and if writing new code, algorithm in pseudocode (25%)
4. The actual code used comments (25%)
5. Answers to any questions on the lab description, results, screenshots, and verification (50%)

- *Reports must be submitted before the start of the next lab.*

Description:

Up to this point, we have somewhat blindly used the Linux system calls for input and output to the terminal and exiting our programs. These system calls will become more important as we start to interface the Red Pitaya to external things (e.g., LEDs and buttons) in the lab next week. Many of the Linux system call libraries consist of subroutines (topic of last lab) that can directly be called by our programs. However, other system calls are macros that allow the assembler to take a block of code from Linux system call libraries and place that code directly in our programs. Hence, this lab begins with understanding how macros work. Then, we will proceed to understanding system calls.

Key Point: Macros. A block of code that is copied by the assembler in each place where it is called. This contrasts with a subroutine that is in another location in memory from the calling program, and hence, a branch and link would be used to move program counter to the part of memory where the subroutine-based system call is located and then return back to the calling program through the link register. Instead, a macro is placed in line with the calling code, expanding it, and removing the need for the branch and link functionality to a different part of memory.

Key Point: System Calls. A functionality that is already defined by the operating system and used by the ARM assembly program through an "SVC 0" instruction. Before this "SVC 0" instruction, their appropriate arguments must be prepared in registers r0-r6 as inputs to the system call. For all Linux system calls, the r0 register specifies its type (see Appendix B for all the types of Linux system calls). A return value may or may not be used by the system call to return a value in the r0 register.

For this lab, we resume going through Chapter 6, starting with page 125 and read through the remainder of the chapter before proceeding to these steps:

1. After reading through pages 125-130, create a file called mainmacro.s from Listing 6-7 and uppermacro.s from the modified Listing 6-8 below (Listing 6-8 in the book had some functional problems).

2. Now, change your makefile from the last lab (Listing 6-5) to where main is changed to mainmacro throughout and upper is changed to uppermacro throughout.
3. Run the program and show that the two strings at the bottom of mainmacro.s are now converted to all upper-case letters.
4. The key point here is that each time that you call the "toupper" macro in your main program, the code will get copied into your machine code. Run "objdump -d uppermacro" and identify the two different instances in which the macro was filled into the main program. Include this on your lab report.
5. At this point, you are able to understand the two options to call library functions that will be used with Linux system calls – subroutines and macros. Let's begin learning about these system calls.
6. Instead of having an embedded string in the .data section at the bottom, now you will have the name of an input file and the name of an output file. To do so, take Listing 6-4 and create a file called upper.s. You may already have this from the last lab (where you modified it to use LDM/STM). However, you can take it as is without modifying the push and pop to complete this lab.
7. Next, use fileio.s from the modified Listing 7-1 below (Listing 7-1 would not function on the Red Pitaya as listed in the book) and Listing 7-2 as main.s.
8. You will need one additional file called unistd.s which is the code in Appendix B for Linux System Calls. Since this is very laborious to type in this code, you can go to the author's github directory and download it from Chapter 7: https://github.com/Apress/Raspberry-Pi-Assembly-Language-Programming/blob/master/Source%20Code/Chapter%207/unistd.s
9. Lastly, create a makefile from Listing 7-3. Make it and run it.
10. If you have done this successfully, there should now be an all-capitalized version of main.s in upper.txt.
11. Create a text file that says "My name is <first-name last-name> with student ID <student-ID>." You can call the file whatever you like. However, just make sure that you update main.s to reflect this filename.
12. Test and make sure that the output file (default name of "upper.txt") is an upper-case version of your input file.
13. Once you are convinced that your code is working, then change main.s to have the same input and output file name which is your file that has "My name is…". Rebuild your program and run the following command line "cat <input-file-name>; ./upper; "cat <input-file-name>". This should print something like this:
    My name is…
    MY NAME IS…
14. Take a screen shot and include it in your report.

ECE 1181: Microcontrollers and Embedded Systems Lab
Spring 2024 – Professor Joe Camp

Code Given: Listings 6-4, 6-7, 6-8, 7-1, 7-2, 7-3, and unistd.s (Linux System Call Numbers from Appendix B). However, Listing 6-8 has some problems, so I have fixed them below (changed lines in bold and red):

```
@
@ Assembler program to convert a string to
@ all uppercase.
@
@ R1 - address of output string
@ R0 - address of input string
@ R2 - original output string for length calc.
@ R3 - current character being processed
@ R4 - new character being stored
@
@ label 1 = loop
@ label 2 = cont
.MACRO toupper instr, outstr
    LDR R0, =\instr
    LDR R1, =\outstr
    MOV R2, R1

@ The loop is until byte pointed to by R1 is non-zero
1:    LDRB  R3, [R0], #1 @ load character and increment pointer

@ If R3 <= 'Z' then goto cont
    CMP R3, #'Z' @ is letter <= 'Z'? 2f
    MOV R4, R3
    BLE 2f @ goto to end if

@ if we got here then the letter is lower-case, so convert it.
    SUB R4, R3, #('a'-'A')

2: @ end if
    STRB R4, [R1], #1 @ store character to output str
    CMP R3, #0 @ stop on hitting a null character
    BNE 1b @ loop if character isn't null
    SUB R0, R1, R2 @ get the length by subtracting the pointers
.ENDM
```

Modified Listing 7-1 (Since the MOV instruction has a rotation scheme that would not fit the number that needs to be moved into R2, we need to add and load the number from memory. Changed lines in bold and red):

```
@ Various macros to perform file I/O
```

```
@ The fd parameter needs to be a register.
@ Uses R0, R1, R7.
@ Return code is in R0.

.include "unistd.s"
.equ O_RDONLY, 0
.equ O_WRONLY, 1
.equ O_CREAT, 0100

.data
S_RDWR: .word 0666

.macro openFile fileName, flags
        ldr r0, =\fileName
        mov r1, #\flags
        ldr r3, =S_RDWR
        ldr r2, [r3] @ RW access rights
        mov r7, #sys_open
        svc 0
.endm
.macro readFile fd, buffer, length
        mov r0, \fd @ file descriptor
        ldr r1, =\buffer
        mov r2, #\length
        mov r7, #sys_read
        svc 0
.endm
.macro writeFile fd, buffer, length
        mov r0, \fd @ file descriptor
        ldr r1, =\buffer
        mov r2, \length
        mov r7, #sys_write
        svc 0
.endm
.macro flushClose fd
@fsync syscall
        mov r0, \fd
        mov r7, #sys_fsync
        svc 0
@close syscall
        mov r0, \fd
        mov r7, #sys_close
        svc 0
.endm
```

ECE 1181: Microcontrollers and Embedded Systems Lab
Spring 2024 – Professor Joe Camp

You have completed this lab when:

Demonstrate to your lab instructor that you have completed the steps above.