5η Εργαστηριακή Άσκηση για το μάθημα Σχεδιασμός Ενσωματωμένων Συστημάτων

Ομάδα 29

Μέλη : Αριστοτέλης Γρίβας

Σάββας Λεβεντικίδης

Ερώτημα 1ο : Μετατροπή εισόδου από τερματικό

Σκοπός της άσκησης είναι να γραφεί πρόγραμμα σε assembly του επεξεργαστή ARM το οποίο θα λαμβάνει ως είσοδο από τον χρήστη μέσω τερματικού, μια συμβολοσειρά μεγέθους έως 32 χαρακτήρων. Αν η είσοδος είναι μεγαλύτερη, οι χαρακτήρες που περισσεύουν, να αγνοούνται. Στην συμβολοσειρά αυτή γίνονται οι εξής μετατροπές :

- Αν ο χαρακτήρας είναι κεφαλαίο γράμμα της αγγλικής αλφαβήτου τότε να μετατρέπεται σε πεζό και αντίστροφα.
- Αν ο χαρακτήρας βρίσκεται στο εύρος [‘0’, ‘9’], θα πραγματοποιείται η ακόλουθη μετατροπή:

‘0’ → ‘5’
‘1’ → ‘6’
‘2’ → ‘7’
‘3’ → ‘8’
‘4’ → ‘9’
‘5’ → ‘0’
‘6’ → ‘1’
‘7’ → ‘2’
‘8’ → ‘3’
‘9’ → ‘4’

- Οι υπόλοιποι χαρακτήρες να παραμένουν αμετάβλητοι.

Το πρόγραμμα πρέπει να είναι συνεχούς λειτουργίας και να τερματίζει όταν λάβει ως είσοδο μια συμβολοσειρά μήκους ένα που θα αποτελείται από τον χαρακτήρα ‘Q’ ή ‘q’. Ο κώδικας που αναφέρεται στην μετατροπή πρέπει να γραφεί ως ξεχωριστή συνάρτηση. Στόχος είναι η ελαχιστοποίηση των γραμμών κώδικα που απαιτούνται για το σώμα της εν λόγω συνάρτησης.

Ακολουθεί η διαδικασία επίλυσης του προβλήματος (Η υλοποίηση του προγράμματος έγινε με χρήση system calls του Linux (read, write, ….)) :

1) Αρχίζουμε με ένα write system call, όπου τυπώνουμε στο stdout το μήνυμα για να ενημερώσουμε το χρήστη να εισάγει νέα συμβολοσειρά (“ Input a string of up to 32 chars long:”)

```
.text
.global main

main:
    push {ip,lr}

start:

    mov r0, #1          @ stdout
    ldr r1, =out_mes    @ message to write to output
    mov r2, #mes_len    @ length of output message
    mov r7, #4          @ write syscall
    swi 0
```

2) Καλούμε το read system call, όπου παίρνουμε από το stdin τη συμβολοσειρά που δίνει ο χρήστης, αποθηκεύοντας τα πρώτα 33 bytes της στη μεταβλητή inp (= πίνακας 33 bytes αρχικοποιημένα με τον κενό χαρακτήρα).

```
mov r0, #0      @ stdin
ldr r1, =inp    @ location of inp_str in memory
mov r2, #33     @ read 32 characters + \n from input
mov r7, #3      @ read syscall
swi 0

mov r3, #0      @ r3 used to count how many characters we convert

@ r1 now has the input characters, and we will load them to r0 to make the conversions
@ the converted values will be stored again in r1, and printed
```

3) Στη συνέχεια έχοντας αποθηκεύσει τη συμβολοσειρά εισόδου στον καταχβρητή r1, ελέγχουμε αν αυτή αποτελείται από ένα χαρακτήρα, και, αν ναι, αν ο χαρακτήρας είναι ‘q’ ή ‘Q’. Σε περίπτωση που κάποιο από τα προηγούμενα ισχύει, το πρόγραμμα τερματίζει (exit), διαφορετικά μπαίνουμε στη **συνάρτηση που κάνει όλους τους μετασχηματισμούς, start_conversion.**

```
   cmp r0, #2                @ check if input is 1 character + \n
   bne start_conversion      @ if it is more than 1 character, we begin the conversion
   ldrb r0, [r1]             @ else if input is 1 character
   cmp r0, #113              @ check if it is q
   beq exit                  @ if it is q exit
   cmp r0, #81               @ else check if it is Q
   beq exit                  @ if it is Q exit, else convert the character
```

4) Η συνάρτηση αυτή, η οποία μετασχηματίζει τους χαρακτήρες έναν-έναν, αρχίζοντας προφανώς από τον πρώτο χαρακτηρα του r1 και τελειώνοντας

στον τελευταίο του, αρχίζει στη γραμμή 35 και τελειώνει στη γραμμή 80. Κάνει τα εξής :

- Αρχίζει αυξάνοντας έναν counter η τιμή του οποίου είναι αποθηκευμένη στον καταχωρητή r3 (αρχικοποιημένος στο 0 σε παραπάνω βήμα). Αν η τιμή του counter γίνει ίση με 33 (αρχικοποιήθηκε στο 0), τότε έχουμε κάνει 32 μετασχηματισμούς, και αφού δε δεχόμαστε παραπάνω χαρακτήρες, προχωράμε στην εμφάνιση του αποτελέσματος μέσω της συνάρτησης print. Αν δεν είναι ίσος με 32, τότε συνεχίζουμε στον επόμενο μετασχηματισμό.
  Αντίστοιχα, ελέγχουμε αν ο χαρακτήρας προς μετασχηματισμό είναι ο \n (EOF). Αν ναι, τότε η όλη η συμβολοσειρά έχει μετασχηματιστεί καθώς φτάσαμε στο τέλος της και πηγαίνουμε στην print, διαφορετικά εκτελούμε κανονικά το μετασχημτισμό.

```
start_conversion:
    add r3, #1          @ we convert a new character, increase r3
    cmp r3, #33         @ if r3 is 33, we have converted 32 characters (all the input)
    beq print           @ if so exit
    ldrb r0, [r1]       @ else load the input character to r0
    cmp r0,#10          @ if the character is \n, the conversion is over
    beq print           @ and we print to output
```

- Στο σημείο αυτό, καλούμαστε να κάνουμε το μετασχηματισμό του εκάστοτε χαρακτήρα. Έτσι κάνουμε τους ελέγχους μας :
  Αν ο χαρακτήρας είναι αριθμός, τότε αυξάνουμε ή μειώνουμε τον ascii κωδικό του κατά 5 θέσεις (δηλαδή αποθηκεύουμε τον αριθμό που έχουμε ±5, αναλόγως αν ο χαρακτήρας μας είναι μικρότερος ή μεγαλύτερος του 5). Διαφορετικά ελέγχουμε αν είναι κεφαλαίο γράμμα.

```
@first checking if character is number
    cmp r0,#57          @ else check ascii number
    bgt check_capital   @ if ascii number is over 57, the character is not a number, check if it is a capital letter
    cmp r0,#48
    blt store_content   @ if ascii number is lower than 48, the character is not a number and not a letter, so we store it as it is


    cmp r0,#53              @ if it is a number, we compare it with number 5
    subge r0, #5            @ if it is greater than 5, subtract by 5
    addlt r0, #5            @ if it is lower than 5, increase by 5
    strb r0, [r1], #1       @ store it
    b start_conversion      @ check next character
```

  Διαφορετικά αν ο χαρακτήτας είναι κεφαλαίο γράμμα, αυξάνουμε τον κωδικό ascii του χαρακτήρα κατά 32 θέσεις, και έτσι μετατρέπεται σε πεζό. Διαφορετικά, ελέγχουμε αν ο χαρακτήρας είναι πεζός.

```
check_capital:
    cmp r0,#90          @ check ascii number
    bgt check_lower     @ if it is more over 90, the character is not a capital letter, check if it is a lower case letter
    cmp r0,#65
    blt store_content   @ if ascii number is lower than 65, the character is not a number or a letter, so we store it

    add r0,r0,#32           @ if it is a capital letter, make it lower
    strb r0, [r1], #1       @ and store it
    b start_conversion      @ check next character
```

  Διαφορετικά αν ο χαρακτήτας είναι πεζός, τότε μειώνουμε τον κωδικό ascii κατά 32 θέσεις, και μετατρέπεται σε κεφαλίο γράμμα.

```
check_lower:
    cmp r0,#122         @ check ascii number
    bgt store_content   @ if it is more over 122, the character is not a number or a letter, so we store it
    cmp r0,#97
    blt store_content   @ if ascii number is lower than 97, the character is not a number or a letter, so we store it

    sub r0,r0,#32               @ if it is a lower case letter, make it capital
    strb r0, [r1], #1           @ store it and check next character
    b start_conversion
```

Σε οποιαδήποτε άλλη περίπτωση, δεν αλλάζουμε τον χαρακτήρα. Στο τέλος κάθε μετασχηματισμού, αποθηκεύουμε τον μετασχηματισμένο χαρακτήρα στη θέση του αρχικού (στο string inp), και συνεχίζουμε στον επόμενο.

```
store_content:
    strb r0, [r1], #1           @ store character and check next
    b start_conversion
```

5) Έχοντας τη μετασχηματισμένη συμβολοσειρά, προχωράμε στην εμφάνιση της στο stdout μέσω write system call, και επιπλέον ελέγχουμε αν η αρχική συμβολοσειρά που έδωσε ο χρήστης ήταν παραπάνω απο 32 χαρακτήρες. Αν ήταν, διαβάζουμε όλους τους έξτρα χαρακτήρες ώστε να αδειάσουμε τον buffer, και συνεχίζουμε στην επόμενη εκτέλεση του κώδικα.

```
print:

    cmp r3,#33          @ check if string had 32 characters
    bne continue        @ if not, continue
    mov r0,#10          @ if yes, add the \n character to the end of the string and continue
    strb r0,[r1],#1
continue:
    mov r0, #1                  @ stdout
    ldr r1,=result_mes          @ "result is" message
    mov r2,#result_mes_len      @ length
    mov r7, #4                  @ write syscall
    swi 0


    mov r0, #1      @ stdout
    ldr r1, =inp    @ converted input message to write to output
    mov r2, r3      @ length
    mov r7, #4      @ write syscall
    swi 0

    cmp r3,#33          @ lastly, if input is more than 32 characters, read until buffe is empty
    beq bigger_than_32
    b start
```

```
bigger_than_32:
    mov r0, #0      @ stdin
    ldr r1, =inp    @ buffer that stores input/converted string
    mov r2, #32     @ read 32 characters
    mov r7, #3      @ read syscall
    swi 0

    cmp r0, #32         @ check what we cleared
    bne start           @ if we cleared less than 32 bytes, everything is cleared
    ldrb r3, [r1, #31]  @ else if we cleared 32 bytes, check if the last was \n
    cmp r3, #10         @ if so we are set
    bne bigger_than_32  @ else clear more

    b start

exit:
    pop {ip,lr}
    mov r7, #1      @ exit syscall
    swi 0

.data
    out_mes: .ascii "Input a string of up to 32 chars long: "
    mes_len = . - out_mes @ .- indicates the length of  out_mes
    inp: .ascii "\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0" @  33 bytes with null character for input string
    result_mes: .ascii "Result is: "
    result_mes_len = . - result_mes
```

Κάνουμε compile τον κώδικα με την παρακάτω εντολή :

gcc -Wall ask1.s -o ask1.out

Και τρέχουμε με :

./ask1.out

Ο κώδικας βρίσκεται και στο zip file (ask1.s), και τα σχόλια δίπλα σε κάθε γραμμή δίνουν μία ακόμα πιο αναλυτική εικόνα του πως λειτουργεί ο κώδικας.

Ερώτημα 2ο : Επικοινωνία των guest και host μηχανημάτων μέσω σειριακής θύρας.

Σκοπός της άσκησης είναι να δημιουργηθούν 2 προγράμματα, ένα σε C στο host μηχάνημα και ένα σε assembly του ARM στο guest μηχάνημα τα οποία θα επικοινωνούν μέσω εικονικής σειριακής θύρας. Το πρόγραμμα στο host μηχάνημα θα δέχεται ως είσοδο έναν string μεγέθους έως 64 χαρακτήρων. Το string αυτό θα αποστέλλεται μέσω σειριακής θύρας στο guest μηχάνημα και αυτό με την σειρά του θα απαντάει ποιος είναι ο χαρακτήρας του string με την μεγαλύτερη συχνότητα εμφάνισης και πόσες φορές εμφανίστηκε. Ο κενός χαρακτήρας (Ascii no 32) εξαιρείται από την καταμέτρηση. Στην περίπτωση που δύο ή παραπάνω χαρακτήρες έχουν μέγιστη συχνότητα εμφάνισης, το πρόγραμμα να επιστρέφει στον host, τον χαρακτήρα με τον μικρότερο ascii κωδικό. Ακολουθεί η υλοποίηση της άσκησης για τον host και τον guest αντίστοιχα.

Host:

1) Αρχικά διαβάζουμε μέσω του read() system call, από το stdin, τη συμβολοσειρά εισόδου από το χρήστη, και ελέγχουμε αν τηρείται ο

περιορισμός για το μέγεθός της (64 χαρακτήρες). Αν δεν τηρείται, τότε εμφανίζεται το αντίστοιχο error, διαφορετικά συνεχίζουμε.

```c
#include <sys/types.h>

#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <termios.h>
#include <stdio.h>

int main(int argc, char **argv)
{
    int fd;
    char in[66], out[66];
    struct termios config;
    ssize_t r;
    char device[] = "/dev/pts/2";

    printf("Please give a string to send to guest:\n");
    r = read(0, in, 66);
    if (r<0){
        perror("read");
        exit(1);
    }
    if (r == 66) {
        fprintf(stderr, "String must have at most 64
characters! Try again.\n");
        fflush(0);
        exit(1);
    }
```

2) Αφού έχουμε διαβάσει τη συμβολοσειρά εισόδου από το χρήστη με επιτυχία, ανοίγουμε τη σειριακή θύρα μέσω του open() system call, με flags O_RDWR και O_NOCTTY, ώστε να μπορούμε να εκτελούμε reads/writes στη θύρα.
Τονίζουμε πως στο open() system call χρησιμοποιούμε το όνομα του αρχείου που δημιουργήθηκε στη δική μας περίπτωση (`"/dev/pts/2"`). Για να τρέξετε το πρόγραμμα στο δικό σας σύστημα, πρέπει αφού βρείτε το όνομα του αντίστοιχου αρχείου (μπορεί να είναι π.χ. `"/dev/pts/7"`), να αλλάξετε τη μεταβλητή *device* στον κώδικα του host στο όνομα αυτό.

Στη συνέχεια, πρέπει να ορίσουμε τις ρυθμίσεις για το configuration, ώστε να έχουμε τα επιθυμητά χαρακτηριστικά για το terminal και την ανάγνωση των γραμμών. Ενδεικτικά, χρησιμοποιούμε Baudrate B9600, καθώς και canonical configuration mode, το οποίο αποθηκεύει τα δεδομένα μας σε ένα

buffer και στέλνει το περιεχόμενό του (τη συμβολοσειρά) μετά από line feed (χαρακτήρα νέας γραμμής \n). Περισσότερες πληροφορίες για τις ρυθμίσεις του configuration και τη μέθοδο που ακολουθήσαμε παρέχονται στο εξής link : https://en.wikibooks.org/wiki/Serial_Programming/termios
Αφού επιλέξουμε τις ρυθμίσεις που θέλουμε, τις θέτουμε στη σειριακή θύρα μέσω της συνάρτησης tcsetattr().

```
    fd = open(device, O_RDWR | O_NOCTTY);


    if (fd == -1) {
        printf("Error opening file\n");
        return 1;
    }

// Get the current configuration of the serial interface
    if(tcgetattr(fd, &config) < 0) {
        printf("Configuration error\n");
        return 1;
    }

    config.c_lflag = 0;
    config.c_lflag |= ICANON; // canonical mode
    config.c_cflag |= (CLOCAL | CREAD | CS8);
    config.c_cflag &= ~CSTOPB;
    config.c_cc[VMIN] = 1; // One input byte is enough to
return from read()
    config.c_cc[VTIME] = 0; // Inter-character timer off


// Communication speed (simple version, using the predefined
// constants)
    if(cfsetispeed(&config, B9600) < 0 || cfsetospeed(&config,
B9600) < 0) {
        printf("Baudrate error\n");
        return 1;
    }

// Finally, apply the configuration
    if(tcsetattr(fd, TCSANOW, &config) < 0) {
        printf("Error with settings\n");
        return 1;
    }
```

3) Τέλος, είμαστε έτοιμοι να στείλουμε το μήνυμα στον guest (αφού έχουμε ρυθμίσει και σε αυτόν τις αντίστοιχες ρυθμίσεις), μέσω write system call. Στη συνέχεια, περιμένουμε μέχρι να διαβάσουμε τα αποτελέσματα από τον guest μέσω read system call και τα τυπώνουμε.

```
 tcflush(fd, TCIOFLUSH); // flush buffer

    printf("Sending message...\n");
    write(fd, in, 66); // send string
    printf("Getting results...\n");
    read(fd, out, 66); // get results

//print them, don't forget to subtract output[2] by 48,
char>int
    if(out[2]-'0'!=0)
      printf("The most frequent character is %c and it
appeared %d times.\n", out[0], out[2]-'0');
    else
      printf("Nothing was written\n");

    close(fd);
    return 0;
}
```

4) Σημειώνουμε πως σε κάθε επανάληψη, είναι αναγκαίο να αδειάζουμε τον buffer που αναφέραμε παραπάνω, για να αποβάλουμε έξτρα χαρακτήρες που μπορεί να βρίσκονται εκεί. Αυτό πραγματοποιείται μέσω της συνάρτησης tcflush().

Guest:

Όπως αναφέρθηκε παραπάνω, ο σκοπός του guest είναι να διαβάσει τη συμβολοσειρά που λαμβάνει από τον host, να υπολογίσει τον χαρακτήρα με την μεγαλύτερη συχνότητα εμφάνισης (ο κενός χαρακτήρας αγνοείται), και να στείλει τη συχνότητα εμφάνισης καθώς και τον εκάστοτε χαρακτήρα πίσω στον host.

Συνεπώς, ακολουθούμε την εξής διαδικασία :

1) Ανοίγουμε τη σειριακή θύρα από τη μεριά του guest, η αποία ορίζεται ως η "/dev/ttyAMA0" και ορίζουμε τις ρυθμίσεις του configuration (όπως κάναμε και στον host). Οι ρυθμίσεις δίνονται στο τέλος του κώδικα στα data με όνομα "options". Στη συνέχεια, αφού έχουμε κάνει τις επιθυμητές ρυθμίσεις, διαβάζουμε το μήνυμα από τον host με read system call σε assembly και το αποθηκεύουμε στον πίνακα *"input"* των 64 bytes (όσοι και οι χαρακτήρες της συμβολοσειράς εισόδου) που έχουμε αρχικοποιήσει με κενούς χαρακτήρες.

```
.text
.global main
.extern tcsetattr

main:

open:
    ldr r0, =serial_port    @ serial_port name
    ldr r1, =#258           @ O_RDWR | O_NOCTTY
    mov r7, #5              @ open syscall
    swi 0

    mov r6, r0              @ save fd from r0, will be needed to read/write

setup:
    mov r0, r6              @ call tcsetattr to set the settings for our port
    ldr r2, =options
    mov r1, #0
    bl tcsetattr
```

```
read:
    mov r0, r6              @ read from fd stored in r6
    ldr r1, =input          @ our 64 byte input array
    mov r2, #64
    mov r7, #3              @ read syscall
    swi 0
```

2) Στο σημείο αυτό, καλούμαστε να κάνουμε τους απαραίτητους υπολογισμούς, ώστε να υπολογίσουμε τη μεγαλύτερη συνότητα εμφάνισης ανάμεσα στους χαρακτήρες της συμβολοσειράς. Η υλοποίηση γίνεται ως εξής :
   - Ορίζουμε έναν πίνακα με 255 θέσεις (freq_array), αρχικοποιημένες όλες στην τιμή 0. Κάθε θέση θα αντιπροσωπεύει έναν κωδικό ascii με τον αριθμό της θέσης να αντιπροσωπεύει την συχνότητα εμφάνισης του χαρακτήρα που αντιστοιχεί στον κωδικό αυτό. Ο πίνακας αυτός θα περιέχει τις συχνότητες εμφάνισεις για κάθε ascii χαρακτήρα. Αν, π.χ. στη θέση 97 του πίνακα έχουμε την τιμή 10, αυτό σημαίνει πως ο χαρακτήρας ‘a’ (με ascii code 97) εμφανίζεται 10 φορές στη συμβολοσειρά. Έτσι, στην αρχή, όλες οι συχνότητες είναι 0.
   - Στη συνέχεια σκανάρουμε τους χαρακτήρες της συμβολοσειράς εισόδου έναν-έναν. Κάθε φορά που εντοπίζουμε έναν χαρακτήρα, αυξάνουμε την τιμή του πίνακα στη θέση που αντιστοιχεί στον κωδικό ascii του χαρακτήρα κατά 1, έως ότου βρούμε τον χαρακτήρα \n, όπου και σταματάμε γιατί εκεί τελειώνει η συμβολοσειρά. Στο τέλος αυτής της διαδικασίας, η συχνότητα εμφάνισης κάθε χαρακτήρα θα είναι αποθηκευμένη στον πίνακα.

```
    ldr r0, =input          @ r0 has the input string loaded, and will be used to check all characters
    ldr r1,=freq_array      @ r1 has the frequency array loaded, and will be used to update the frequencies
calculate_freq:
    ldrb r2, [r0], #1       @ get  ascii code of character, and increase address by 1 for next iteration
    cmp r2, #10             @ if character is \n, we have scanned all of the input characters
    beq freq_calculated     @ if so, check which character is the most frequent
    ldrb r3, [r1,r2]        @ else load the frequency of ascii character in r2, from frequency array r1
    add r3,r3,#1            @ increase its frequency by 1
    strb r3, [r1,r2]        @ store the increased value back
    b calculate_freq        @ repeat for next iteration
```

- Τώρα το μόνο που έχουμε να κάνουμε είναι να διασχύσουμε τον πίνακα freq_array, κρατώντας σε κάθε επανάληψη τη μέγιστη συχνότητα και τον αντίστοιχο χαρακτήρα έως στο σημείο αυτό.
  Τονίζουμε πως η διάσχυση του πίνακα ξεκίνησε από τη θέση #33, καθώς όλοι οι χαρακτήρες με κωδικό ascii μικρότερο από αυτή την τιμή δεν είναι printable (και ο #32 να είναι το space που αγνοείτε), άρα δε χρειάζετε να τους ελέγξουμε. Η τιμή της μέγιστης συχνότητας κρατείτε στον καταχωρητή r0 ενώ ο αντίστοιχος χαρακτήρας στον καταχωρητή r4.

```
freq_calculated:
    mov r0, #0          @ r0 will store max, current_max = 0
    ldr r1,=freq_array  @ r1 has the frequencies again
    mov r2,#33          @ r2 is the offset used to parse the frequency array, we begin by character with ascci code 33
                        @ because all the previous characters are not printable

find_max:
    ldrb r3, [r1,r2]    @ get frequency with offset r2 (initially 33 = character '!')
    cmp r0,r3           @ compare the frequency of the character with the max frequency until now
    movlt r0,r3         @ if we found new max, update r0  with the max frequency
    movlt r4, r2        @ and store the ascii code of the most frequent char in r4
    add r2,r2,#1        @ increase r2 for next iteration
    cmp r2,#256         @ if we've checked all 256 ascii characters
    beq all_calculated  @ we can exit the function
    b find_max          @ else repeat for next iteration
```

3) Στη συνέχεια αποθηκεύουμε το περιεχόμενο των καταχωρητών αυτών στον πίνακα εξόδου (output), στέλνουμε το αποτέλεσμα στο host μέσω write system call, και κλείνουμε το device και το πρόγραμμα μέσω close και exit system calls αντίστοιχα.

```
all_calculated:
  ldr r1,=output
  strb r4, [r1]         @ store ascii code of most frequent character in output[0]
  strb r0, [r1,#2]      @ store the value of frequency (+48 because we initialized with freq_array with 0) in output[2]
                        @ we subtract by 48 to get the number at the host


write:
    mov r0, r6
    ldr r1, =output
    ldr r2, =output_len
    mov r7, #4          @ write system call
    swi 0

close:
    mov r0, r6
    mov r7, #6          @ close system call
    swi 0

exit:
    mov r0, #0
    mov r7, #1          @ exit system call
    swi 0
```

```
.data
    options: .word 0x00000000 @ c_iflag
             .word 0x00000000 @ c_oflag
             .word 0x000008bd @ c_cflag
             .word 0x00000002 @ c_lflag
             .byte 0x00       @ c_line
             .word 0x00000000 @ c_cc[0-3]
             .word 0x00010000 @ c_cc[4-7]
             .word 0x00000000 @ c_cc[8-11]
             .word 0x00000000 @ c_cc[12-15]
             .word 0x00000000 @ c_cc[16-19]
             .word 0x00000000 @ c_cc[20-23]
             .word 0x00000000 @ c_cc[24-27]
             .word 0x00000000 @ c_cc[28-31]
             .byte 0x00       @ padding
             .hword 0x0000    @ padding
             .word 0x0000000d @ c_ispeed
             .word 0x0000000d @ c_ospeed

    serial_port: .asciz "/dev/ttyAMA0"

    input: .ascii "\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0
    len = . - input
    freq_array: .ascii "0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
    output: .asciz "0000\n"
    output_len = . - output
```

Κανουμε compile τους κώδικες με :

gcc -Wall guest.s -o guest.out

gcc -Wall host.c -o host.out

Τρέχουμε πρώτα τον κώδικα του guest με :

./guest.out

Και στη συνέχεια του Host με :

sudo ./host.out

Ο κώδικας βρίσκεται και στο zip file (ask2), και τα σχόλια δίπλα σε κάθε γραμμή δίνουν μία ακόμα πιο αναλυτική εικόνα του πως λειτουργεί ο κώδικας.

Ερώτημα 3ο : Σύνδεση κώδικα C με κώδικα assembly του επεξεργαστή ARM.

Σκοπός της παρούσας άσκησης είναι να συνδυαστεί κώδικας γραμμένος σε C με συναρτήσεις γραμμένες σε assembly του επεξεργαστή ARM. Το πρόγραμμα string_manipulation.c που μας δίνεται έτοιμο ανοίγει ένα αρχείο με 512 γραμμές, κάθε γραμμή του οποίου περιέχει μια τυχαία κατασκευασμένη συμβολοσειρά, μεγέθους από 8 έως 64 χαρακτήρες. Κατά την εκτέλεση του προγράμματος κατασκευάζονται 3 αρχεία εξόδου:

1. Το πρώτο περιέχει το μήκος της κάθε γραμμής του αρχείου εισόδου.
2. Το δεύτερο περιέχει τις συμβολοσειρές του αρχείου εισόδου ενωμένες (concatenated) ανά 2.
3. Το τρίτο περιέχει τις συμβολοσειρές του αρχείου εισόδου ταξινομημένες σε αύξουσα αλφαβητική σειρά

Για να επιτευχθούν οι παραπάνω στόχοι γίνεται χρήση των συναρτήσεων strlen, strcpy, strcat και strcmp από την βιβλιοθήκη string.h. Στόχος της άσκησης είναι να αντικαταστήσουμε τις παραπάνω συναρτήσεις με δικές μας γραμμένες σε ARM assembly.

Αφού υλοποιήσουμε τις 4 συναρτήσεις σε assembly σε ξεχωριστά αρχεία, όπως παρουσιάζεται παρακάτω, πρέπει να δηλώσουμε τις συναρτήσεις αυτές ως extern στον κώδικα C. Απαραίτητο είναι η συνάρτηση να έχει δηλωθεί στο κώδικα assembly με το directive .global για να μπορεί να είναι ορατή από τον linker κατά την σύνδεση των δύο αρχείων. Στη συνέχεια, κάνουμε compile (-c flag του gcc) το αρχείο string_manipulation.c και το ίδιο για κάθε ένα αρχείο .s των συναρτήσεων. Συνδέουμε (link) τα object αρχεία που παρήχθησαν από τα παραπάνω βήματα, για την παραγωγή του τελικού εκτελέσιμου αρχείου string_manipulation.out.

Στην παρούσα άσκηση δεν χρειάστηκε να αλλάξουμε τίποτα στο αρχείο string_manipulation.c καθώς ήταν εντελώς έτοιμο. Επομένως χρειάστηκε να υλοποιήσουμε τις 4 συναρτήσεις ακριβώς όπως προδιαγράφεται στα man pages της κάθε συνάρτησης καθώς και το makefile έτσι ώστε να πραγματοποιεί όσα αναφέρθηκαν στην προηγούμενη παράγραφο.

Υλοποίηση συναρτήσεων:

strlen.s

```
.text
.align 4
.global strlen
.type strlen, %function

start:
    @r0 is the address of the string

    sub r0, r0, #1 @now ro is the address of the previous byte of the
string
    mov r2, #0 @initialize counter

loop:
    ldrb r1, [r0,#1]! @auto-indexing addressing, r1=mem[r0+1],r0=r0+1
```

```
    cmp r1, #0 @if r1 is the null character you should exit
    beq exit
    add r2, r2, #1 @if it isnt null then add to the counter
    b loop @loop until null

exit:
    mov r0, r2
    bx lr
```

Η συγκεκριμένη συνάρτηση δέχεται ως είσοδο τη διεύθυνση μιας συμβολοσειράς και επιστρέφει το μήκος της. Εφόσον η συνάρτηση έχει ένα όρισμα, το όρισμα αυτό ,δηλαδή η διεύθυνση της συμβολοσειράς, θα περνάει στον καταχωρητή r0 όπως και σε κάθε συνάρτηση σε arm assembly. Η έξοδος επίσης θα δοθεί από τον r0. Στην υλοποίηση χρησιμοποιήσαμε τον καταχωρητή r2 ως μετρητή (αρχικοποιημένο στο 0) και τον καταχωρητή r1 ως καταχωρητή προσωρινής φόρτωσης χαρακτήρα από την συμβολοσειρά. Σε κάθε επανάληψή φορτώνουμε από τη θέση μνήμης στην οποία δείχνει ο r0 έναν χαρακτήρα (1 byte) και αυξάνουμε τον r0 κατά 1, προκειμένου να δείχνει στην επόμενη θέση μνήμης. Αν ο χαρακτήρας που διαβάσαμε είναι το τερματικό σύμβολο, η συνάρτηση επιστρέφει. Διαφορετικά, η τιμή του μετρητή αυξάνεται κατά 1 και συνεχίζουμε με την επόμενη επανάληψη, ώστε να διαβάσουμε τον επόμενο χαρακτήρα της συμβολοσειράς.

Τα σχόλια στον κώδικα είναι πολύ κατατοπιστικά.

strcpy.s

```
.text
.align 4
.global strcpy
.type strcpy, %function

start:
    mov r2, r0 @save the address of the destination string
    sub r0, r0, #1 @for the inital store
    sub r1, r1, #1 @for the initial load

loop:
    ldrb r3, [r1,#1]! @auto-indexing addressing, r3=mem[r1+1],r1=r1+1
    strb r3, [r0,#1]! @auto-indexing addressing, mem[r0+1]=r3,r0=r0+1,
store even if null
    cmp r3, #0 @ if null exit
    bne loop

exit:
    mov r0, r2 @return the destination address
    bx lr
```

Η συγκεκριμένη συνάρτηση δέχεται ως είσοδο στον καταχωρητή r0 τη διεύθυνση μιας συμβολοσειράς s1 και στον r1 τη διεύθυνση μιας συμβολοσειράς s2. Η συνάρτηση πρέπει να αντιγράψει την s2 στην s1. Αρχικά αποθηκεύουμε την διεύθυνση του destination string δηλαδή το r0 καθώς και μειώνουμε τον r0 και r1 ώστε στην πρώτη επανάληψη (εξαιτίας του auto-indexing addressing) να ξεκινήσει η φόρτωση στους προσωρινούς καταχωρητές από την σωστή διεύθυνση. Σε κάθε επανάληψή, διαβάζουμε ένα byte (έναν χαρακτήρα) από τη διεύθυνση που περιέχει ο r1, την οποία και αυξάνουμε στη συνέχεια κατά 1, για να προετοιμαστούμε για την επόμενη επανάληψη. Ο χαρακτήρας που διαβάστηκε αποθηκεύεται στη θέση μνήμης, η διεύθυνση της οποίας βρίσκεται στον r0. Στη συνέχεια, αυξάνουμε το διεύθυνση που περιέχει ο r0, για να προετοιμαστούμε για την επόμενη επανάληψη. Αν ο χαρακτήρας που διαβάστηκε ήταν ο τερματικός, η επανάληψη σταματάει. Διαφορετικά, συνεχίζουμε με την αντιγραφή του επόμενου χαρακτήρα. Τέλος αφού ολοκληρωθεί ο βρόχος φορτώνουμε στον r0 την διεύθυνση του string που έγινε η αντιγραφή, την οποία είχαμε αποθηκεύσει εμείς στην αρχή του προγράμματος.

Τα σχόλια στον κώδικα είναι πολύ κατατοπιστικά.

strcat.s

```
.text
.align 4
.global strcat
.type strcat, %function

start:
    mov r2, r0 @save the destination address
    sub r0, r0, #1 @for the inital load
    sub r1, r1, #1 @for the initial load

end_of_dst:
    ldrb r3, [r0, #1]! @auto-indexing addressing, r3=mem[r0+1],r0=r0+1,
load destination char
    cmp r3, #0
    bne end_of_dst @you didnt find the end so loop

loop:
    ldrb r3, [r1, #1]! @auto-indexing addressing, r3=mem[r1+1],r1=r0+1,
load src char
    strb r3, [r0], #1 @post-index addressing, mem[r0]=r3, r0 = r0+1,
store the src char to the dest string end
    cmp r3, #0
    bne loop @you didnt find the end of src string so loop

exit:
    mov r0, r2 @return dest address
    bx lr
```

Η συγκεκριμένη συνάρτηση δέχεται ως είσοδο στον καταχωρητή r0 τη διεύθυνση μιας συμβολοσειράς s1 και στον r1 τη διεύθυνση μιας συμβολοσειράς s2. Η συνάρτηση πρέπει να

αντιγράψει τη δεύτερη στο τέλος της πρώτης. Όπως και για την strcpy, αποθηκεύουμε την destination address ώστε να την δώσουμε ως έξοδο στο τέλος της συνάρτηση. Αρχικά, διατρέχουμε τη συμβολοσειρά s1 μέχρι να φτάσουμε στον τερματικό χαρακτήρα με μέθοδο παρόμοια με τις άλλες συναρτήσεις δηλαδή με διάβασμα χαρακτήρα- χαρακτήρα. Όταν φτάσουμε στο τέλος της συμβολοσειράς στον r0 υπάρχει το τερματικό σύμβολο το οποίο αργότερα θα αντικαταστήσουμε με τον πρώτο χαρακτήρα της δεύτερης συμβολοσειράς. Ύστερα, ακολουθούμε διαδικασία όμοια με αυτή της strcpy, διαβάζοντας έναν έναν τους χαρακτήρες της s2 και αποθηκεύοντας τους στη διεύθυνση που δείχνει ο r0, η οποία αυξάνεται κατά 1 σε κάθε επανάληψη. Μόλις αντιληφθούμε ότι έχει αντιγραφεί και ο τερματικός χαρακτήρας της δεύτερης συμβολοσειράς βγαίνουμε από το βρόχο. Τέλος αφού ολοκληρωθεί ο βρόχος φορτώνουμε στον r0 την διεύθυνση του string που έγινε η προσκόλληση, την οποία είχαμε αποθηκεύσει εμείς στην αρχή του προγράμματος.

Τα σχόλια στον κώδικα είναι πολύ κατατοπιστικά.

strcmp.s

```
.text
.align 4
.global strcmp
.type strcmp, %function

start:
    push {r4}
    sub r0, r0, #1 @for the inital load
    sub r1, r1, #1 @for the initial load

loop:
    ldrb r2, [r0,#1]! @auto-indexing addressing, r2=mem[r0+1],r0=r0+1
    ldrb r3, [r1,#1]! @auto-indexing addressing, r3=mem[r1+1],r1=r1+1
    cmp r2,r3
    beq equal
    blt less
    mov r4, #1 @if the value isnt the same or less then it is greater so
exit with 1
    b exit

equal:
    cmp r2, #0 @since they are equal then r3 = #0 so exit with 0
    bne loop
    mov r4, #0
    b exit

less:
    mov r4, #0xffffffff @exit with -1 since the value of the first arg is
less than the value of the second
    b exit

exit:
```

```
    mov r0, r4
    pop {r4}
    bx lr
```

Η συγκεκριμένη συνάρτηση δέχεται ως είσοδο στον καταχωρητή r0 τη διεύθυνση μιας συμβολοσειράς s1 και στον r1 τη διεύθυνση μιας συμβολοσειράς s2. Η συνάρτηση πρέπει να συγκρίνει λεξικογραφικά τις δύο συμβολοσειρές, επιστρέφοντας τελικά, μέσω του r0, το αποτέλεσμα της σύγκρισης. Σε κάθε επανάληψη διαβάζουμε ένα byte (έναν χαρακτήρα) από τη διεύθυνση που περιέχει ο r0 και ένα από τη διεύθυνση που περιέχει ο r1, τις οποίες και αυξάνουμε στη συνέχεια κατά 1, για να προετοιμαστούμε για την επόμενη επανάληψη. Συγκρίνουμε τους δύο χαρακτήρες και διακρίνουμε 3 περιπτώσεις. Αν είναι ίσοι τότε ελέγχουμε αν είναι οι τερματικοί χαρακτήρες και αν ισχύει αυτό τότε το πρόγραμμα τερματίζει με έξοδο 0 που σημαίνει ότι οι συμβολοσειρές είναι ίσες. Αν δεν είναι οι τερματικοί χαρακτήρες τότε επαναλαμβάνουμε την διαδικασία για τους επόμενους. Αν ο χαρακτήρας από την πρώτη συμβολοσειρά είναι μικρότερος από τον χαρακτήρα της δεύτερης τότε έχουμε s1 < s2 και επομένως επιστρέφουμε αρνητική τιμή (συγκεκριμένα -1). Αν από την άλλη προκύψει ότι ο χαρακτήρας που διαβάστηκε από το s1 είναι μεγαλύτερος αυτού απ' το s2, τότε s1 > s2 και επομένως επιστρέφουμε θετική τιμή (συγκεκριμένα 1). Τελικά επιστρέφουμε μέσω του r0 το αποτέλεσμα: 0 αν είναι ίσες , -1 αν s1 < s2 και 1 αν s1 > s2.

Τα σχόλια στον κώδικα είναι πολύ κατατοπιστικά.

Αφού εκτελέσαμε το αρχείο string_manipulation.out που προέκυψε από το Makefile με τα δύο αρχεία εισόδου που μας δόθηκαν ελέγξαμε την εγκυρότητα των αποτελεσμάτων με αυτά που προκύπτουν με χρήση των ίδιων συναρτήσεων αλλά υλοποιημένων μέσα στην βιβλιοθήκη <string.h>.