Lab 5 (All Sections) Prelab: Linking

Name:

Yerania Hernandez

Sign the following statement:

On my honor, as an Aggie, I have neither given nor received unauthorized aid on this academic work

Yerania Hernandez

1 Objective

The main objective of this lab is to understand how a compiler and linker work. Before proceeding with this lab, you should be familiar with relocation entries and compiler operation.

2 Introduction

In this lab you will experiment with linking two source files together by using a `gcc` compiler.

3 Questions

1. Consider the following three machine instructions, which appear in memory starting at the address `0x00400000`.

address (in hex)	contents (in hex)
0x00400000	0x12080002
0x00400004	0x3C11FFFF
0x00400008	0x08100004

   (a) "Disassemble" the instructions; that is, give an assembly language program segment that would be translated into the given machine language.

   (b) You may use numeric rather than symbolic register names.

   (c) Handle branches and jumps specially; where you would normally have a label, provide instead a hexadecimal byte address.

```
beq  $s0, $t0, 0x00400008
lui $s1, 0x0000FFFF
j 0x00400010
```

   For each of the instructions, indicate whether

   (a) It MUST have contributed an entry to the relocation table,

   (b) It MAY have contributed an entry to the relocation table, or

   (c) It COULD NOT have contributed an entry to the relocation table.

   Briefly explain your answers.

0x00400000: It could not have contributed an entry since it is a branch and it uses PC-relative addressing.
0x00400004: It must have contributed to an entry to the relocation table since it is a data reference and it will need the address.
0x00400008: It must have contributed an entry to the relocation table because it is an absolute address.

2. During which phase of the process from coding to execution do each of the following things happen? Place the most appropriate letter to the answer next to each statement. Some letters may never be used; others may be used more than once.

- G The stack pointer decreases;
- I jr `$ra`;
- F Jump statements are resolved;
- H a jal instruction is executed;
- E Symbol and relocation tables are created;
- B Machine code is copied from disk into memory;
- E Pseudo-instructions are replaced.

a) Never
b) during loading
c) while writing higher-level code
d) during the compilation
e) during assembly
f) during linking
g) when malloc is called
h) when a function is called
i) when a function returns
j) when registers are spilled

3. Consider the following fragment of MIPS.

```
1        Loop: lui $at, 0xABCD      # a
2        ori $a0,$at, 0xFEDC # b
3        jal add_link          # c
4        bne $a0,$v0, Loop     # d
```

Which of the following instructions may need to be edited during link phase? Explain.

#a & #b) lui and ori are both data references and therefore will need to be relocated.
#c) The jal will need to be edited during link phase in order to reflect a new starting location in the code and its data segment.

4. Consider the following assembly language program segment:

```
1        funca:
2        la    $t0, n
3        lw    $s0, 0($t0)
4        addi  $s2,$zero,1
5        sub   $s1,$s1,$s1
6        loop:
7        sll       $t0,$s2,31
8        slt       $t1,$t0,$zero
9        bne       $t1,$zero,skip
10       add       $s1,$s1,$s2
11       skip:
12       addi  $s2,$s2,1
```

```
13          beq         $s2 , $s0 , done
14          j       loop
15          done:
16          addi  $v0 ,  $s1 ,  0
17          jr      $ra
18
19          n:      .word           11
```

(a) List instructions that will contribute an entry to the relocation table.

```
la $t0, n
j loop
```

(b) Substitute all instructions listed in Question 4a to obtain *Position Independent Code* (PIC). Position Independent Code does not have any entries in the relocation table, i.e., it does not have any absolute address.

Hint: substitute all absolute addresses by relative addresses; store all constants directly in the program instead of the data segment.

```
funca :
li $s0, 11
addi $s2, $zero, 1
sub $s1, $s1, $s1

loop :
sll $t0, $s2, 31
slt $t1, $t0, $zero
bne $t1, $zero, skip
add $s1, $s1, $s2

skip :
addi $s2, $s2, 1
beq $s2, $s0, done
beq $0, $0, loop

done :
addi $v0, $s1, 0
jr $ra
```

5. Assume that a user has many programs; some of them share libraries with each other. The user sometimes runs more than one of these programs at the same time. The users disk is always slower than the users main memory (RAM). For this user compare the performance of "dynamic linking" with "static linking":

   (a) With respect to the time programs take to start up

   (b) With respect to required disk space.

   (c) With respect to the amount of system memory in use at runtime.

a)Dynamic linking with respect to start up takes longer than static since dynamic linking requires time overhead to load each individual file.
b) Dynamic linking requires less disk space since the execution code is being stored in individual files while static linking stores the entire library. Also, executing two programs requires less memory, especially if they are sharing a library.
c) Dynamic linking needs more system memory at runtime consider the external libraries need to be linked to the program