

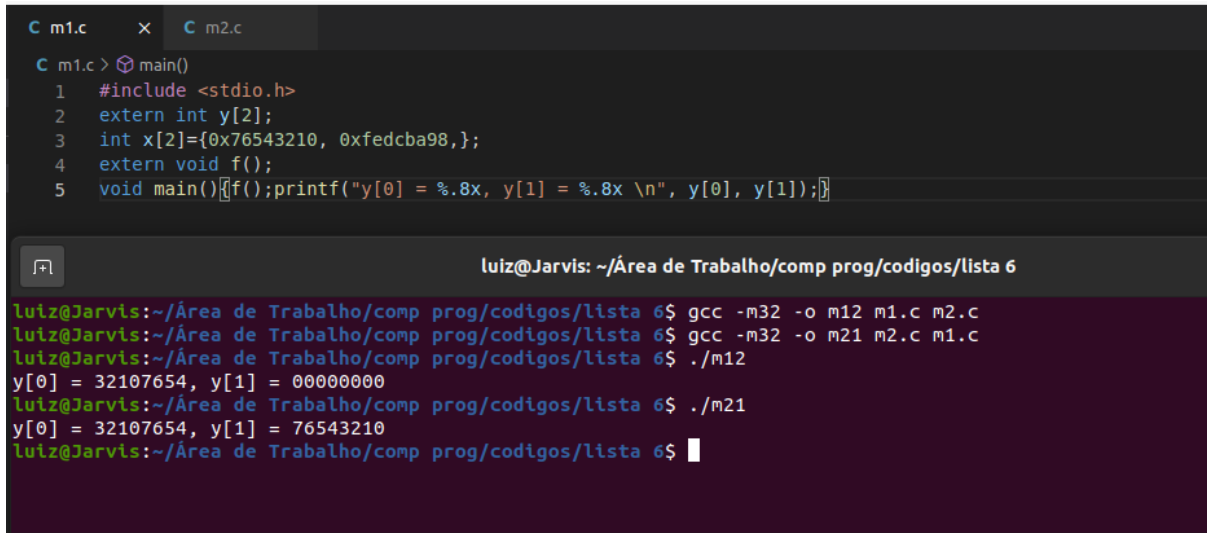
Lista 6 - Comp Prog

Alunos: Luiz Rodrigo Lacé Rodrigues (DRE:11804983)

Livia Barbosa Fonseca (DRE:118039721)

Questão 1)

a)



```
C m1.c x C m2.c
C m1.c > main()
1  #include <stdio.h>
2  extern int y[2];
3  int x[2]={0x76543210, 0xfedcba98,};
4  extern void f();
5  void main(){f();printf("y[0] = %.8x, y[1] = %.8x \n", y[0], y[1]);}

luiz@Jarvis: ~/Área de Trabalho/comp prog/codigos/lista 6
luiz@Jarvis:~/Área de Trabalho/comp prog/codigos/lista 6$ gcc -m32 -o m12 m1.c m2.c
luiz@Jarvis:~/Área de Trabalho/comp prog/codigos/lista 6$ gcc -m32 -o m21 m2.c m1.c
luiz@Jarvis:~/Área de Trabalho/comp prog/codigos/lista 6$ ./m12
y[0] = 32107654, y[1] = 00000000
luiz@Jarvis:~/Área de Trabalho/comp prog/codigos/lista 6$ ./m21
y[0] = 32107654, y[1] = 76543210
luiz@Jarvis:~/Área de Trabalho/comp prog/codigos/lista 6$
```

b)

Podemos perceber que o vetor x vai ser referenciado externamente como sendo um int em m1.c e em m2.c será um vetor de short int. Podemos notar que em m1.c o vetor vai ser declarado como global inteira e será inicializado. Sendo assim, teremos um símbolo forte em m1.c se sobrepondo a um símbolo fraco em m2.c. Por se tratar de uma variável global iniciada, temos que x está localizada na seção .data do arquivo objeto resultante e ocupa 8 bytes de memória, pois temos 2 espaços de memória de 4 bytes cada, por se tratar de um int.

Já o y, é referenciado externamente como um int em m1.c e em m2.c é referenciado como short int. Temos que em m2.c a variável será declarada como global de short inteiro e será inicializada. Podemos observar que em m1.c teremos um símbolo fraco e em m2.c um símbolo forte. Sendo assim, o vetor y, por se tratar de uma variável inicializada, estará localizada na seção .data do arquivo e ocupa 4 bytes, pois teremos 2 posições de 2 bytes cada.

c)

Vamos imprimir a tabela de símbolos de m12 por meio do comando “readelf -s m12”

```
C m1.c x C m2.c
C m1.c > main()
1 #include <stdio.h>
2 extern int y[2];
3 int x[2]={0x76543210, 0xfedcba98,};
4 extern void f();
5 void main(){f();printf("y[0] = %.8x, y[1] = %.8x \n", y[0], y[1]);}

luiz@Jarvis: ~/Área de Trabalho/comp prog/codigos/lista 6
luiz@Jarvis:~/Área de Trabalho/comp prog/codigos/lista 6$ readelf -s m12

Tabela de símbolo ".dynsym" contém 8 entradas:
Número: Tamanho do Valor do Tipo de Vínculo Nome Vis Ndx
0: 00000000 0 NOTYPE LOCAL DEFAULT UND
1: 00000000 0 NOTYPE WEAK DEFAULT UND _ITM_deregisterTMCloneTab
2: 00000000 0 lista6.pdf GLOBAL DEFAULT UND printf@GLIBC_2.0 (2)
3: 00000000 0 FUNC WEAK DEFAULT UND __cxa_finalize@GLIBC_2.1.3 (3)
4: 00000000 0 NOTYPE WEAK DEFAULT UND __gmon_start__
5: 00000000 0 FUNC GLOBAL DEFAULT UND __libc_start_main@GLIBC_2.0 (2)
6: 00000000 0 NOTYPE WEAK DEFAULT UND _ITM_registerTMCloneTable
7: 00002004 4 OBJECT GLOBAL DEFAULT 18 _IO_stdin_used

Tabela de símbolo ".symtab" contém 74 entradas:
Número: Tamanho do Valor do Tipo de Vínculo Nome Vis Ndx
0: 00000000 0 NOTYPE LOCAL DEFAULT UND
1: 000001b4 0 SECTION LOCAL DEFAULT 1
2: 000001c8 0 SECTION LOCAL DEFAULT 2
3: 000001ec 0 SECTION LOCAL DEFAULT 3
4: 00000208 0 SECTION LOCAL DEFAULT 4
5: 00000228 0 SECTION LOCAL DEFAULT 5
6: 00000248 0 SECTION LOCAL DEFAULT 6
7: 000002c8 0 SECTION LOCAL DEFAULT 7
8: 00000366 0 SECTION LOCAL DEFAULT 8
```

```
C m1.c x C m2.c
C m1.c > main()
1 #include <stdio.h>
2 extern int y[2];
3 int x[2]={0x76543210, 0xfedcba98,};
4 extern void f();

luiz@Jarvis: ~/Área de Trabalho/comp prog/codigos/lista 6

46: 00001000 0 FUNC LOCAL DEFAULT 12 _init
47: 000012d0 5 FUNC GLOBAL DEFAULT 16 __libc_csu_fini
48: 00000000 0 NOTYPE WEAK DEFAULT UND _ITM_deregisterTMCloneTab
49: 000010d0 4 FUNC GLOBAL HIDDEN 16 __x86.get_pc_thunk.bx
50: 00004000 0 NOTYPE WEAK DEFAULT 25 data_start
51: 00000000 0 FUNC GLOBAL DEFAULT UND printf@GLIBC_2.0
52: 000012d5 0 FUNC GLOBAL HIDDEN 16 __x86.get_pc_thunk.bp
53: 00004014 0 NOTYPE GLOBAL DEFAULT 25 _edata
54: 00004008 8 OBJECT GLOBAL DEFAULT 25 x
55: 000012dc 0 FUNC GLOBAL HIDDEN 17 _fini
56: 00001220 53 FUNC GLOBAL DEFAULT 16 f
57: 000011c9 0 FUNC GLOBAL HIDDEN 16 __x86.get_pc_thunk.dx
58: 00000000 0 FUNC WEAK DEFAULT UND __cxa_finalize@@GLIBC_2.1
59: 00004000 0 NOTYPE GLOBAL DEFAULT 25 __data_start
60: 00000000 0 NOTYPE WEAK DEFAULT UND __gmon_start__
61: 00004004 0 OBJECT GLOBAL HIDDEN 25 __dso_handle
62: 00002004 4 OBJECT GLOBAL DEFAULT 18 _IO_stdin_used
63: 00000000 0 FUNC GLOBAL DEFAULT UND __libc_start_main@@GLIBC_
64: 00001260 101 FUNC GLOBAL DEFAULT 16 __libc_csu_init
65: 00004018 0 NOTYPE GLOBAL DEFAULT 26 _end
66: 00001090 58 FUNC GLOBAL DEFAULT 16 _start
67: 00002000 4 OBJECT GLOBAL DEFAULT 18 _fp_hw
68: 00004010 4 OBJECT GLOBAL DEFAULT 25 y
69: 00004014 0 NOTYPE GLOBAL DEFAULT 26 __bss_start
70: 000011cd 83 FUNC GLOBAL DEFAULT 16 main
71: 00001255 0 FUNC GLOBAL HIDDEN 16 __x86.get_pc_thunk.ax
```

Podemos observar que a variável x é listada na linha 54 da tabela e está na posição de memória 0x0004008. Já a variável y se encontra na linha 68, sendo localizada na posição 0x00004010.

Agora vamos analisar a tabela de símbolos de m21 por meio do comando “readelf -s m21”

```

C m1.c x C m2.c
C m1.c > main()
1 #include <stdio.h>
2 extern int y[2];
3 int x[2]={0x76543210, 0xfedcba98,};
4 extern void f();

luiz@Jarvis: ~/Área de Trabalho/comp prog/codigos/lista 6
luiz@Jarvis:~/Área de Trabalho/comp prog/codigos/lista 6$ readelf -s m21

Tabela de símbolo ".dynsym" contém 8 entradas:
Número: Tamanho do Valor do Tipo de Vínculo Nome Vis Ndx
0: 00000000 0 NOTYPE LOCAL DEFAULT UND
1: 00000000 0 NOTYPE WEAK DEFAULT UND _ITM_deregisterTMCloneTab
2: 00000000 0 FUNC GLOBAL DEFAULT UND printf@GLIBC_2.0 (2)
3: 00000000 0 FUNC WEAK DEFAULT UND __cxa_finalize@GLIBC_2.1.3 (3)
4: 00000000 0 NOTYPE WEAK DEFAULT UND __gmon_start__
5: 00000000 0 FUNC GLOBAL DEFAULT UND __libc_start_main@GLIBC_2.0 (2)
6: 00000000 0 NOTYPE WEAK DEFAULT UND _ITM_registerTMCloneTable
7: 00002004 4 OBJECT GLOBAL DEFAULT 18 _IO_stdin_used

Tabela de símbolo ".syntab" contém 74 entradas:
Número: Tamanho do Valor do Tipo de Vínculo Nome Vis Ndx
0: 00000000 0 NOTYPE LOCAL DEFAULT UND
1: 000001b4 0 SECTION LOCAL DEFAULT 1
2: 000001c8 0 SECTION LOCAL DEFAULT 2
3: 000001ec 0 SECTION LOCAL DEFAULT 3
4: 00000208 0 SECTION LOCAL DEFAULT 4
5: 00000228 0 SECTION LOCAL DEFAULT 5
6: 00000248 0 SECTION LOCAL DEFAULT 6
7: 000002c8 0 SECTION LOCAL DEFAULT 7
8: 00000366 0 SECTION LOCAL DEFAULT 8
9: 00000378 0 SECTION LOCAL DEFAULT 9
10: 000003a8 0 SECTION LOCAL DEFAULT 10

C m1.c x C m2.c
C m1.c > main()
1 #include <stdio.h>
2 extern int y[2];
3 int x[2]={0x76543210, 0xfedcba98,};
4 extern void f();

luiz@Jarvis: ~/Área de Trabalho/comp prog/codigos/lista 6
49: 000010d0 4 FUNC GLOBAL HIDDEN 16 __x86.get_pc_thunk.bx
50: 00004000 0 NOTYPE WEAK DEFAULT 25 data_start
51: 00000000 0 FUNC GLOBAL DEFAULT UND printf@GLIBC_2.0
52: 000012d5 0 FUNC GLOBAL HIDDEN 16 __x86.get_pc_thunk.bp
53: 00004014 0 NOTYPE GLOBAL DEFAULT 25 __edata
54: 0000400c 8 OBJECT GLOBAL DEFAULT 25 x
55: 000012dc 0 FUNC GLOBAL HIDDEN 17 _fini
56: 000011cd 53 FUNC GLOBAL DEFAULT 16 f
57: 000011c9 0 FUNC GLOBAL HIDDEN 16 __x86.get_pc_thunk.dx
58: 00000000 0 FUNC WEAK DEFAULT UND __cxa_finalize@GLIBC_2.1
59: 00004000 0 NOTYPE GLOBAL DEFAULT 25 __data_start
60: 00000000 0 NOTYPE WEAK DEFAULT UND __gmon_start__
61: 00004004 0 OBJECT GLOBAL HIDDEN 25 __dso_handle
62: 00002004 4 OBJECT GLOBAL DEFAULT 18 _IO_stdin_used
63: 00000000 0 FUNC GLOBAL DEFAULT UND __libc_start_main@GLIBC_
64: 00001260 101 FUNC GLOBAL DEFAULT 16 __libc_csu_init
65: 00004018 0 NOTYPE GLOBAL DEFAULT 26 __end
66: 00001090 58 FUNC GLOBAL DEFAULT 16 __start
67: 00002000 4 OBJECT GLOBAL DEFAULT 18 __fp_hw
68: 00004008 4 OBJECT GLOBAL DEFAULT 25 y
69: 00004014 0 NOTYPE GLOBAL DEFAULT 26 __bss_start
70: 00001206 83 FUNC GLOBAL DEFAULT 16 main
71: 00001202 0 FUNC GLOBAL HIDDEN 16 __x86.get_pc_thunk.ax
72: 00004014 0 OBJECT GLOBAL HIDDEN 25 __TMC_END__
73: 00000000 0 NOTYPE WEAK DEFAULT UND _ITM_registerTMCloneTable
luiz@Jarvis:~/Área de Trabalho/comp prog/codigos/lista 6$

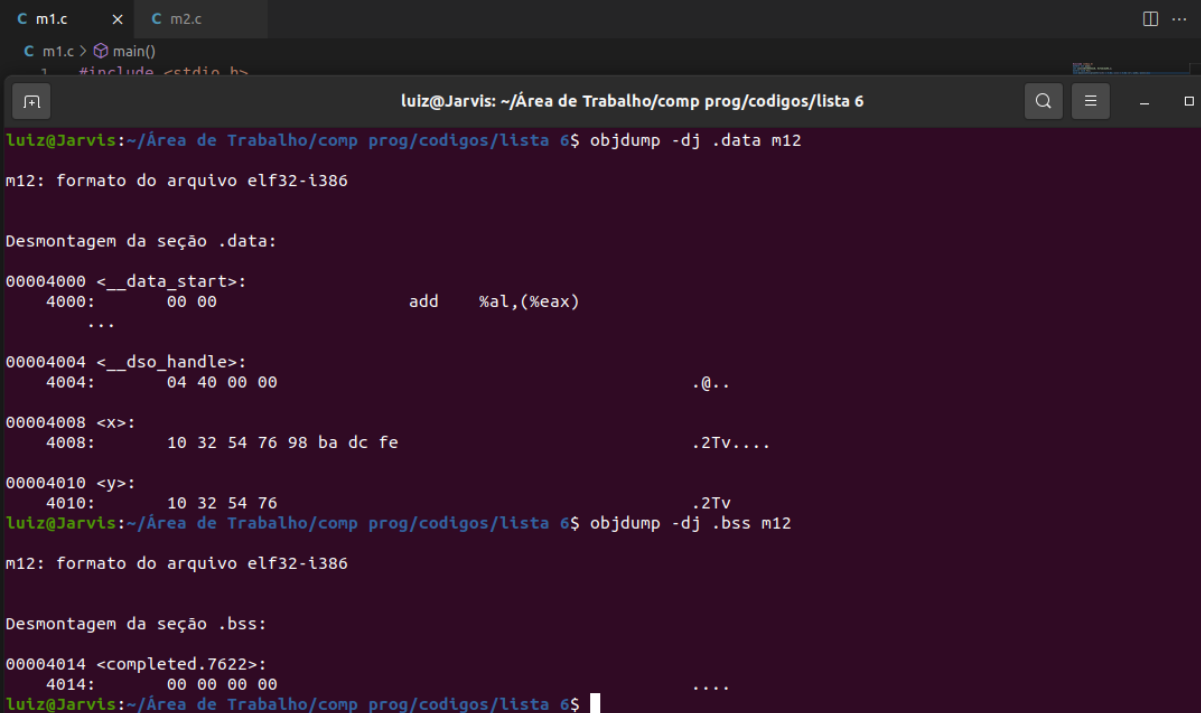
```

No m12, o compilador irá alocar espaço para x antes de y, pois m1.c é compilado antes de m2.c. Sabendo que x é um símbolo forte e y é declarado em outro módulo, tendo em vista, o uso da palavra “extern”. Assim, quando m2.c for compilado, x já está declarado e y virá logo a seguir na memória, pois aqui o y é um símbolo forte.

Já no m21, teremos o oposto. Como y é um símbolo forte em m2.c e x não, y será alocado primeiro na memória e depois x ao compilar m1.c.

d)

Com ajuda do comando “objdump -dj .data m12” vamos listar as seções .data e .bss de m12:



```
C m1.c x C m2.c
C m1.c > main()
1 #include <stdio.h>

luiz@Jarvis: ~/Área de Trabalho/comp prog/codigos/lista 6
luiz@Jarvis:~/Área de Trabalho/comp prog/codigos/lista 6$ objdump -dj .data m12
m12: formato do arquivo elf32-i386

Desmontagem da seção .data:
00004000 <__data_start>:
4000: 00 00 add %al,(%eax)
...
00004004 <__dso_handle>:
4004: 04 40 00 00 .@..
00004008 <x>:
4008: 10 32 54 76 98 ba dc fe .2Tv....
00004010 <y>:
4010: 10 32 54 76 .2Tv
luiz@Jarvis:~/Área de Trabalho/comp prog/codigos/lista 6$ objdump -dj .bss m12
m12: formato do arquivo elf32-i386

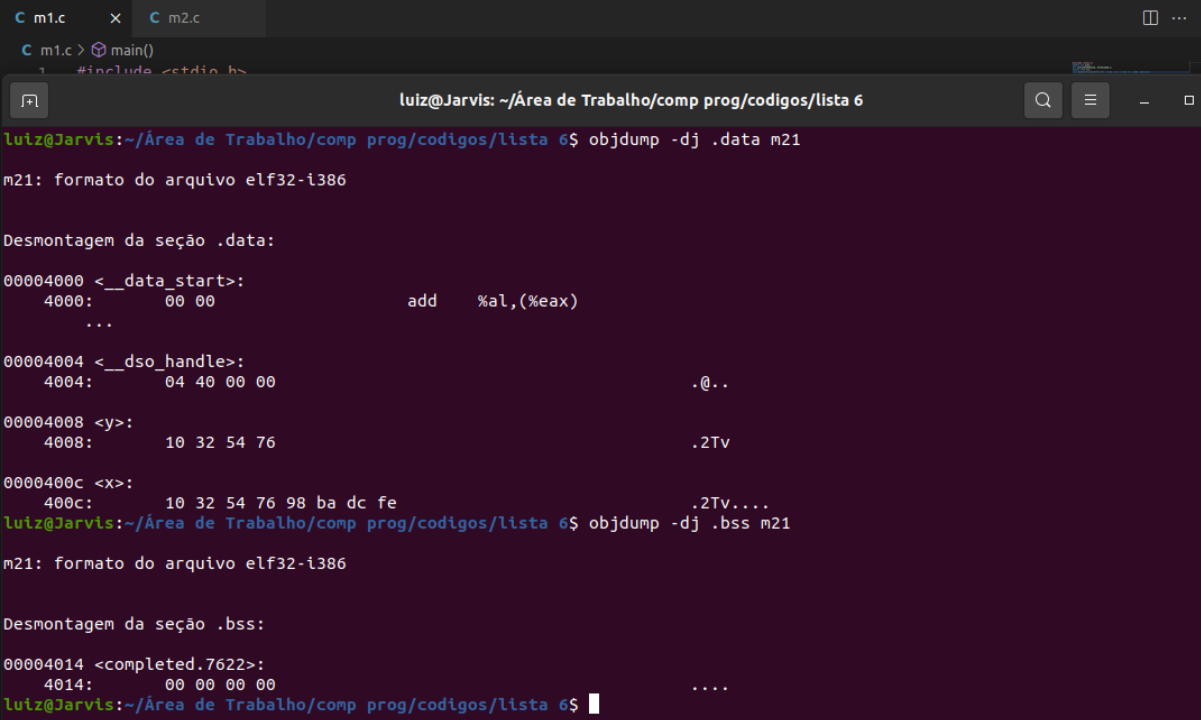
Desmontagem da seção .bss:
00004014 <completed.7622>:
4014: 00 00 00 00 ....
luiz@Jarvis:~/Área de Trabalho/comp prog/codigos/lista 6$
```

Sabendo que a saída desse programa é $y[0] = 32107654$ $y[1] = 00000000$. A função f está declarada no arquivo $m2.c$. Nele, y e y são declarados como short int. Assim, temos x sendo declarado, porém, só sendo inicializado em $m1.c$ com os valores de $x[0] = 0x76543210$ e $x[1] = 0xfedcba98$, como visto na imagem acima temos 10 32 54 76 98 ba dc fe. Assim, ao referenciar $x[1]$ e $x[0]$, podemos observar que vamos utilizar apenas os primeiros 4 bytes do vetor x declarado em $m1.c$. Sendo assim, teremos que $x[0]$ é dado por 10 32 e $x[1]$ é dado por 54 76. Logo, $y[0]$ vai receber 54 76 e $y[1]$ vai receber 10 32 na função f . Porém o `printf`, que é o responsável por printar o conteúdo está no arquivo $m1.c$ e nele x e y são declarados como int. Sabendo que $y[0]$ será dado pelos dois bytes de y modificados na função f e mais outros 2 bytes que estão se seguida na memória, teremos que 2 bytes vão ser 0, pois a variável y vai começar na seção .bss. Essa seção possui espaços preenchidos com zero, para variáveis, e espaços preenchidos com NULL para ponteiros. Como y foi declarado como short int em $m2.c$ e como int em $m1.c$, os 4 bytes restantes serão inicializados com zeros.

Como a impressão formata para `.8x`, teremos os bytes completados com 0. Como $y[1]$ pertence a seção .bss, será impresso o 0. Sabendo que `.8x` que estende com zeros a esquerda se o número possuir menos do que 8 dígitos após a sua conversão para hexa. Logo, teremos “00000000” sendo impresso.

e)

Com ajuda do comando “objdump -dj .data m21” vamos listar as seções .data e .bss de m12:



```
C m1.c x C m2.c
C m1.c > main()
1 #include <stdio.h>

luiz@Jarvis: ~/Área de Trabalho/comp prog/codigos/lista 6
luiz@Jarvis:~/Área de Trabalho/comp prog/codigos/lista 6$ objdump -dj .data m21
m21: formato do arquivo elf32-i386

Desmontagem da seção .data:
00004000 <__data_start>:
4000: 00 00          add    %al,(%eax)
...
00004004 <__dso_handle>:
4004: 04 40 00 00          .@..
00004008 <y>:
4008: 10 32 54 76          .2Tv
0000400c <x>:
400c: 10 32 54 76 98 ba dc fe .2Tv....
luiz@Jarvis:~/Área de Trabalho/comp prog/codigos/lista 6$ objdump -dj .bss m21
m21: formato do arquivo elf32-i386

Desmontagem da seção .bss:
00004014 <completed.7622>:
4014: 00 00 00 00          ....
luiz@Jarvis:~/Área de Trabalho/comp prog/codigos/lista 6$
```

Podemos observar que y será declarado antes de x, assim, todos terão o mesmo conteúdo inicial do item anterior. Assim, teremos a saída do nosso programa será y[0] = 32107654 e y[1] = 76543210.

Pelo mesmo motivo anterior, após a chamada de f, y[0] vai receber 54 76 e y[1] vai receber 10 32. O printf irá interpretar y como inteiro, assim, os 4 primeiros bytes de y terão 10 32 54 76 e a função irá imprimir y[0] = 32107654.

Como agora o compilador irá alocar espaço para y antes de x, teremos x em cima de y. E como x[0] foi iniciado com 0x76543210 e y será considerado um vetor de inteiros no m1.c, o printf irá entender que y[1] será os próximos 4 bytes após y[0]. Como podemos ver com a imagem acima, depois de y teremos os bytes 10 32 54 76. Como o processador é little endian, teremos y[1] = 76543210. Esse valor vai ser o valor impresso na tela.

f) Vamos criar um realocável m1.o utilizando o comando “gcc -m32 -fno-PIC -O1 -c m1.c”. Depois, vamos listar o código de montagem junto com a informação de realocação por meio do comando “objdump -d m1.o -r” do terminal.

```
m1.c - lista 6 - Visual Studio Code

luiz@Jarvis: ~/Área de Trabalho/comp prog/codigos/lista 6
luiz@Jarvis:~/Área de Trabalho/comp prog/codigos/lista 6$ gcc -m32 -fno-PIC -O1 -c m1.c
luiz@Jarvis:~/Área de Trabalho/comp prog/codigos/lista 6$ objdump -d m1.o -r

m1.o: formato do arquivo elf32-i386

Desmontagem da seção .text:

00000000 <main>:
 0: f3 0f 1e fb      endbr32
 4: 8d 4c 24 04      lea 0x4(%esp),%ecx
 8: 83 e4 f0         and $0xffffffff0,%esp
 b: ff 71 fc        pushl -0x4(%ecx)
 e: 55             push %ebp
 f: 89 e5          mov %esp,%ebp
11: 51             push %ecx
12: 83 ec 04       sub $0x4,%esp
15: e8 fc ff ff ff  call 16 <main+0x16>
16: R_386_PC32    f
1a: ff 35 04 00 00 00 pushl 0x4
1c: R_386_32      y
20: ff 35 00 00 00 00 pushl 0x0
22: R_386_32      y
26: 68 00 00 00 00 00 push $0x0
27: R_386_32      .rodata.str1.1
2b: 6a 01         push $0x1
2d: e8 fc ff ff ff  call 2e <main+0x2e>
2e: R_386_PC32    __printf_chk
32: 83 c4 10       add $0x10,%esp
35: 8b 4d fc       mov -0x4(%ebp),%ecx
38: c9            leave
39: 8d 61 fc       lea -0x4(%ecx),%esp
3c: c3            ret
luiz@Jarvis:~/Área de Trabalho/comp prog/codigos/lista 6$
```

Podemos observar que teremos 5 entradas da tabela de realocação. A primeira, será “16: R_386_PC32 f”, com alvo localizado no byte 0x16 da seção .text. Ela faz relação com a realocação da função do arquivo m2.c. O tipo de alocação ‘R_386_PC32’ nos mostra que ele será feito usando o endereço relativo do PC. Sabendo assim, que para obtermos a posição da função temos que somar PC o valor. Analisando o código de montagem, percebemos que o PC naquela instrução vale 0x1A, e o valor inicial colocado é de -4. Esse é o conteúdo inicial porque, por padrão, a conta deve dar igual à posição da própria referência que precisa ser alterada. Dessa forma, temos 0x1A - 0x4 = 0x16.

Em seguida, podemos observar a entrada “1c: R_386_32 y”, está localizada no byte 0x1c da seção .text, e tem relação com a realocação do vetor y, definido em m2.c, cuja segunda posição (y[1]) é passada como argumento de printf. O “R_386_32” nos fala sobre a realocação por endereço absoluto de memória. O conteúdo inicial do código de montagem é 0x4 pois, se somarmos esse valor com a posição de memória absoluta de y, encontramos o endereço de y[1].

A terceira entrada será “22: R_386_32 y”, ela está localizada em 0x22 da seção .text, essa entrada faz referência à realocação do vetor y, definido em m2.c, cuja primeira posição (y[0]) é passada como argumento de printf. “R_386_32” nos informa que a realocação é por endereço absoluto de memória. O conteúdo inicial do código de montagem é 0x0 porque, ao somar com a posição de memória absoluta de y, resultará no endereço de y[0].

A quarta entrada é “27: R_386_32 .rodata.str1.1” está localizada em 0x27 da seção .text. Essa entrada faz referência à realocação da lista de impressão do printf (.rodata.str1.1), localizada na seção de .rodata. “R_386_32” indica a realocação por endereço absoluto de memória. O conteúdo inicial do código de montagem é 0x0 pois, se somarmos o conteúdo com a posição de memória da lista de impressão, vamos obter a mesma referência.

A última será “2e: R_386_PC32 __printf_chk” e estará no byte 0x2b da seção .text. Ela faz referência à realocação da função printf() que é chamada na segunda linha da main(). “R_386_PC32” nos mostra que ele será feito usando o endereço relativo do PC, de forma que se somarmos PC com o valor, teremos a posição da função. No código de montagem, percebemos que o PC naquela instrução vale 0x2b, e o valor inicial é -4. Esse é o conteúdo inicial pois, a conta deve dar igual à posição da própria referência que precisa ser alterada. Nesse caso, teremos $0x2b - 0x4 = 0x27$.

g)

Vamos criar um realocável m2.o utilizando o comando “gcc -m32 -fno-PIC -O1 -c m2.c”. Depois, vamos listar o código de montagem junto com a informação de realocação por meio do comando “objdump -d m2.o -r” do terminal.

```
luiz@Jarvis: ~/Área de Trabalho/comp prog/codigos/lista 6
luiz@Jarvis:~/Área de Trabalho/comp prog/codigos/lista 6$ gcc -m32 -fno-PIC -O1 -c m2.c
luiz@Jarvis:~/Área de Trabalho/comp prog/codigos/lista 6$ objdump -d m2.o -r

m2.o: formato do arquivo elf32-i386

Desmontagem da seção .text:

00000000 <f>:
   0: f3 0f 1e fb          endbr32
   4: 0f b7 05 02 00 00 00 movzwl 0x2,%eax
                        7: R_386_32      x
   b: 66 a3 00 00 00 00    mov    %ax,0x0
                        d: R_386_32      y
  11: 0f b7 05 00 00 00 00 movzwl 0x0,%eax
                        14: R_386_32      x
  18: 66 a3 02 00 00 00    mov    %ax,0x2
                        1a: R_386_32      y
  1e: c3                  ret
luiz@Jarvis:~/Área de Trabalho/comp prog/codigos/lista 6$
```

Podemos observar que teremos 4 entradas da tabela de realocação na imagem anterior. A primeira que encontramos será “7: R_386_32 x”, com alvo localizado no byte 0x7 da seção .text, essa entrada faz referência à realocação do vetor x, definido em m1.c, cuja segunda posição (x[1]) é atribuída para y[0] na primeira linha da função. “R_386_32” nos mostra que indica a realocação por endereço absoluto de memória. O conteúdo inicial do código de montagem é 0x2 porque, ao somar com a posição de memória absoluta de x, resultará no endereço de x[1].

A segunda entrada é “d: R_386_32 y”, localizada no byte 0xd da seção .text. Ela faz referência à realocação do vetor y, variável global de m2.c, cuja primeira posição (y[0]) receberá o valor de x[1] pela primeira linha da função. “R_386_32” indica a realocação por

endereço absoluto de memória. O conteúdo inicial do código de montagem é 0x0 porque, ao somar com a posição de memória absoluta de y, resultará no endereço de y[0].

A terceira entrada é “14: R_386_32 x”, com alvo localizado no byte 0x14, essa entrada faz referência à realocação do vetor x, que está m1.c, cuja primeira posição (x[0]) é atribuída para y[1] na segunda linha da função. “R_386_32” nos mostra a realocação por endereço absoluto de memória. O conteúdo inicial do código de montagem é 0x0 porque, ao somar com a posição de memória absoluta de x, resultará no endereço de x[0].

A quarta entrada é “1a: R_386_32 y”, com alvo localizado no byte 0x1a. Ela faz referência à realocação do vetor y, variável global de m2.c, cuja segunda posição (y[1]) receberá o valor de x[0] pela segunda linha da função. “R_386_32” nos informa a realocação por endereço absoluto de memória. O conteúdo inicial do código de montagem é 0x2 porque, ao somar com a posição de memória absoluta de y, resultará no endereço de y[1].

Questão 2)

a)

Podemos observar que ao executar o call transferimos o controle para outra parte do código. Assim, a função (`__x86.get_pc_thunk.bx`) realiza a alteração do conteúdo de `%ebx`, fazendo com que ele tenha o valor do Program Counter. Dessa forma, quando ele retorna para executar a linha 10, teremos o valor da posição de memória da instrução 10 do código de montagem.

b)

Ele representa o valor de referência para a realocação relativa à constante de GOT, dada por `_GLOBAL_OFFSET_TABLE_`, que se encontra em `%ebx`. Quando ocorre a atribuição de endereços, que se dá depois da ligação, o compilador fornece um endereço constante. Assim, quando é aplicado um deslocamento de 2 bytes a esse endereço, para encontrar o ponto de realocação em relação ao início da instrução, obtemos o ponto de entrada para os símbolos da tabela. E quando somamos 2 bytes ao endereço resolvido pelo compilador, conseguimos acessar o endereço de GOT em `%ebx`.

c)

A função `f` é a do próprio usuário, ela é declarada em m1.c. Ela se encontra no arquivo m2.c que ainda precisará ser ligado. Sendo assim, precisamos realocar a referência da função para que ela possa ser chamada. Por estarmos utilizando o PIC, iremos ligar a função de forma semelhante de quando ligamos a “função de biblioteca dinâmica”, por meio de ligação tardia, com o PLT. Dessa forma, `f@PLT` é uma constante que referencia a entrada de PLT responsável por ligar a função.

d)

“`y@GOT`” e “`.LC0@GOTOFF`” são constantes que quando são somadas à posição de memória inicial da tabela GOT nos mostram referências da variável global `y` e da lista de impressão `.LC0`.

Analisando o código de montagem, após a linha 10, %ebx tem a posição de memória de início da tabela GOT e a linha 12 faz %eax receber y@GOT + %ebx. Assim, na linha 13 podemos pegar o valor de y[1], que é um dos parâmetros da função printf() que se encontra na linha 18. Podemos observar que a lista de impressão .LC0 é passada como parâmetro para a printf() na linha 15. Aqui temos a referência dela sendo .LC0@GOTOFF + %ebx.

Uma das vantagens de ter um código PIC sobre um -fno-PIC em relação a realocação é que o PIC não precisará realocar cada uma das referências quando estamos ligando, o que não acontece no "-fno-PIC". No código PIC, basta que ele altere os valores de GOT, o que fica menos custoso.

Questão 3)

a)

Sabendo que a função que se encontra na linha 9 coloca o valor da instrução que o Program Counter aponta na chamada dela o registrador %ebx.

Analisando a linha 10, podemos observar que %ebx guarda 0x11E5. Nessa linha, somamos 0x2DF3 ao %ebx, logo, %ebx irá guardar 0x3FD8 que é o valor da tabela GOT, como explicamos no exercício anterior.

b)

Vamos calcular os endereços das rotinas __x86.get_pc_thunk.bx e f. O cálculo se dá somando o Program Counter com o valor, de forma que essa soma seja o endereço de __x86.get_pc_thunk.bx. E podemos perceber que, nesse momento na qual a função é chamada, o Program Counter aponta para 0x11E5.

Sabendo que a instrução realiza um deslocamento de EB FE FF FF em little endian em relação à 0x11E5, o PC irá apontar para a soma (0x11E5 + 0xFFFFFEEB). Já que 0xFFFFFEEB = -277, o endereço da f é 0x10D0.

Para calcularmos o endereço de f iremos usar o mesmo procedimento. Nesse momento, o Program Counter aponta para 0x11F0 e o deslocamento é de 0x26. Quando somados, teremos 0x1216.

c)

Vamos calcular o endereço de carga da lista de controle de printf. Na linha 15, observamos que ocorre a operação para recuperar o controle de printf, pois o segundo argumento de printf é essa lista e ela é empilhada na linha 16 por meio do registrador %eax.

Dos exercícios anteriores, sabemos que %ebx é 0x3FD8. Quando realizamos a soma de -0x1FD0 com 0x3FD8 (por meio do comando na linha 15), encontramos 0x2008.

d)

Nesse caso, não precisamos realizar o endereçamento indireto utilizando o gcc, visto que y está na seção .data. Assim, vamos substituir %eax + valor para obter o endereço de y, pelo endereço de y, sabendo que %eax é GOT[0]=0x000011e5 + 0x00002df3 = 0x00003fd8. Realizando essa mudança vamos evitar um acesso à memória, que aconteceria no

processo indireto indicado por GOT. Sabendo que quando existe um acesso à GOT temos um processo ineficiente.

Sendo assim, a instrução “lea” vai passar o endereço de y diretamente para o registrador. Dessa forma, ele é mais eficiente, pois não temos acesso à memória, o que é mais vantajoso e ganhamos eficiência.

e1)

O segmento de dados que é indicado por LOAD e flags rw-, está alinhado em fronteira de 4k. Esse segmento possui tamanho total de 0x13c bytes. Seu endereço inicial se encontra em 0x00003ed8 e o final em 0x00004014. Os primeiros 0x13a bytes são lidos e carregados na memória, já os que não foram inicializados, não ocupam espaço em disco e são inicializados com zero.

O segmento de código, que é indicado por LOAD e flags r-x, está alinhado em fronteira de 4k. Esse segmento possui tamanho total de 0x2e4 bytes. Seu endereço inicial se encontra em 0x00001000 e o final em 0x000012e3. Todos os bytes são lidos e carregados na memória.

Sabendo que os vetores x e y estão dentro da seção .data, para mostrarmos que os endereços dos vetores x e y e a tabela GOT estão no segmento de dados, precisamos verificar os endereços de .data e .bss. Se esses endereços estiverem dentro do segmento de dados, x, y e GOT também estarão.

Agora vamos analisar se x, y e GOT se encontram dentro do segmento de dados. Sabendo que os primeiros 0x13a do segmento de dados são lidos do disco e o endereço final de .data é 0x00004011 (0x00003ed8+ 139). Dessa forma, podemos observar que o endereço final da seção .bss é igual o end final do segmento de dados. Assim, a seção .bss equivale aos 3 bytes que são inicializados com 0 e não estão na memória.

e2)

O segmento de Read-Only Memory (somente leitura) armazena o “.rodata”. Ele vai guardar o formato de string para printf, porém, como só teremos dados para leitura, teremos apenas constantes estáticas.

Podemos observar que nesse exercício, o conteúdo de .rodata é a string “y[0] = %.8x, y[1] = %.8x \n”. Sabendo que o segmento de dados tem um tamanho de 0x2e4 bytes e todos os bytes são lidos e carregados na memória, assim, o tamanho reservado para as variáveis estáticas em .rodata.

Questão 4)

a)

Vamos começar analisando a tabela GOT:

```
//Aqui temos o end da seção .dynamic
e0 3e 00 00 GOT[0]
//Aqui temos a informação de identificação para ligador
00 00 00 00 GOT[1]
//Aqui temos um ponto de entrada para o ligador dinâmico
00 00 00 00 GOT[2]
//Aqui temos o endereço de pushl de PLT[1] - a printf
40 10 00 00 GOT[3]
//Aqui temos o endereço de pushl em PLT[2] - addvec
50 10 00 00 GOT[4]
00 00 00 00 GOT[5]
```

Podemos observar com o código acima que GOT[2] vai ser carregado no início do processo por meio do carregador dinâmico com o endereço do ligador dinâmico. O ligador dinâmico acaba se ligando quando a função “printf” é executada pela primeira vez. Sendo assim, cuja função principal será carregar na entrada de GOT, o endereço de printf, para que a função printf seja executada a partir da segunda chamada. No nosso caso, teremos o GOT[3] realizando esse processo.

Podemos observar abaixo a sequência de instruções executadas:

```
1080: f3 0f 1e fb endbr32
//Teremos um desvio para PLT[2]
1084: ff a3 10 00 00 00 jmp *0x10(%ebx)
//Nesse momento não teremos nenhuma operação acontecendo
108a: 66 0f 1f 44 00 00 nopw 0x0(%eax,%eax,1)
1050: f3 0f 1e fb endbr32
//Aqui pegamos o ID para addvec
1054: 68 08 00 00 00 push $0x8
//Realizamos um jmp para PLT[0]
1059: e9 d2 ff ff ff jmp 1030 <.plt>
105e: 66 90 xchg %ax,%ax
//Aqui estamos carregando conteúdo de GOT[1] na pilha
1030: ff b3 04 00 00 00 pushl 0x4(%ebx)
//Realizamos um jmp para *GOT[2] (ligador) preenchimento
1036: ff a3 08 00 00 00 jmp *0x8(%ebx)
-- Nesse momento o GOT ligador dinâmico vai reescrever seu --
    conteúdo com MEM[addvec] e vai desviar para addvec
//Nesse momento não teremos nenhuma operação acontecendo
103c: 0f 1f 40 00 nopl 0x0(%eax)
```

b)

Quando carregamos uma biblioteca dinâmica ela é compartilhada por todos os processos que chamam a biblioteca. Quando a biblioteca já está carregada, ela é ligada sem que haja necessidade de carregar outro código. Porém, quando a biblioteca não está carregada em memória, e é chamada pela primeira vez, ela é carregada na memória pelo ligador dinâmico. Quando usamos as bibliotecas compartilhadas, temos a geração de executáveis que ocupam menor espaço no disco.

A printf é uma biblioteca dinâmica compartilhada e tem um único código carregado na memória física. Dessa forma, temos um melhor uso da memória física, tendo um melhor desempenho para alocar páginas na memória física.

Sabendo que as modificações e atualizações na biblioteca ficam transparentes para o programa, visto que a biblioteca é ligada na carga do processo pelo carregador dinâmico antes do início da execução do processo ou na ligação tardia pelo ligador dinâmico em tempo de execução. Dessa forma, podemos realizar a manutenção dos códigos em paralelo.