

Questão 1 (35 pontos) Todas as figuras a serem anexadas na resposta deste item devem ser capturas de tela legíveis. Sejam dados os seguintes arquivos C:

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

```
Arquivo m2.c:
short int y[2]={0x3210,0x7654};
extern short int x[2];
void f(){
y[0] = x[1];
y[1] = x[0];
}
```

a) (5) Gere o executável m12 com "gcc -m32 -o m12 m1.c m2.c". Gere o executável m21 com "gcc -m32 -o m21 m2.c m1.c", agora invertendo a ordem dos arquivos fontes. Compile e execute m12 e m21 numa única janela de shell, capture a tela mostrando as duas compilações e as duas execuções. Anexe a captura e garanta que a figura esteja legível.

b) (5) Justifique, com os argumentos adequados sobre símbolos, a atribuição de memória (tipo da variável, endereço na memória e quantidade de bytes alocada) feita pelo GCC para os vetores x e y. Só mostrar a captura não é suficiente, pois o fundamental é explicar e justificar as atribuições feitas às variáveis x e y.

c) (5) Imprima a tabela de símbolos de m12, indicando o comando utilizado. Capture a saída mostrando as linhas da tabela relativas a x e y. Faça o mesmo para m21. A partir do item anterior, analise e conclua como a ordem de compilação afeta a alocação de memória.

d) (5) Liste, numa mesma tela, o conteúdo das seções .data e .bss do executável m12. Indique o comando usado e anexe a captura da tela. Indique o conteúdo de x[0], x[1], y[0] e y[1]. Justifique agora a saída obtida ao rodar m12 no item (a), explicando clara e detalhadamente.

e) (5) Liste, numa mesma tela, o conteúdo das seções .data e .bss do executável m21. Indique o comando usado e anexe a captura da tela. Indique o conteúdo de x[0], x[1], y[0] e y[1]. Justifique agora a saída obtida ao rodar m21 no item (a), explicando clara e detalhadamente.

f) (5) Compile m1.c com as opções -m32 -fno-PIC -O1 -c para obter m1.o. Descubra como listar o código de montagem de main junto com a informação de realocação. É preciso que a informação de realocação esteja já junto com o código de montagem, indicando onde a realocação será feita. Documente com a captura de tela. Explique e justifique a razão dos tipos de realocação apresentados e, em especial, o conteúdo inicial nos bytes que serão realocados. Como a posição de realocação é conhecida, você deve explicar a escolha do tipo de realocação de cada entrada.

g) (5) Compile m2.c com as opções -m32 -fno-PIC -O1 -c para obter m2.o. Descubra como listar o código de montagem de main junto com a informação de realocação. É preciso que a informação de realocação esteja já junto com o código de montagem, indicando onde a realocação será feita. Documente com a captura de tela. Explique e justifique a razão dos tipos de realocação apresentados e, em especial, o conteúdo inicial nos bytes que serão realocados. Como a posição de realocação é conhecida, você deve explicar a escolha do tipo de realocação de cada entrada.

Questão 2 (20 pontos) A compilação de m1.c com as opções -m32 -O1 -fPIC -S gera o código de montagem abaixo, onde suprimimos as diretivas e linhas não relevantes.

```
.file "m1.c"
.LC0:
.string "y[0] = %.8x, y[1] = %.8x \n"
main:
1 endbr32
2 leal 4(%esp), %ecx
3 andl $-16, %esp
4 pushl -4(%ecx)
5 pushl %ebp
6 movl %esp, %ebp
7 pushl %ebx
8 pushl %ecx
9 call __x86.get_pc_thunk.bx
10 addl $_GLOBAL_OFFSET_TABLE_, %ebx
11 call f@PLT
12 movl y@GOT(%ebx), %eax
13 pushl 4(%eax)
14 pushl (%eax)
15 leal .LC0@GOTOFF(%ebx), %eax
16 pushl %eax
17 pushl $1
18 call __printf_chk@PLT
19 addl $16, %esp
20 leal -8(%ebp), %esp
21 popl %ecx
22 popl %ebx
23 popl %ebp
24 leal -4(%ecx), %esp
25 ret
...
__x86.get_pc_thunk.bx:
26 movl (%esp), %ebx
27 ret
```

O desmonte do arquivo realocável obtido com gcc -m32 -fPIC -O1 -c m1.c, correspondente ao código de montagem acima, junto com a informação de realocação, gera:

```
00000000 <main>:
1 0: f3 0f 1e fb          endbr32
2 4: 8d 4c 24 04          lea    0x4(%esp),%ecx
3 8: 83 e4 f0              and    $0xffffffff0,%esp
4 b: ff 71 fc              pushl  -0x4(%ecx)
5 e: 55                    push   %ebp
6 f: 89 e5                mov    %esp,%ebp
7 11: 53                    push   %ebx
8 12: 51                    push   %ecx
9 13: e8 fc ff ff ff        call   14 <main+0x14>
14: R_386_PC32 __x86.get_pc_thunk.bx
10 18: 81 c3 02 00 00 00      add    $0x2,%ebx
1a: R_386_GOTPC _GLOBAL_OFFSET_TABLE_
11 1e: e8 fc ff ff ff        call   1f <main+0x1f>
1f: R_386_PLT32 f
12 23: 8b 83 00 00 00 00      mov    0x0(%ebx),%eax
25: R_386_GOT32X y
13 29: ff 70 04              pushl  0x4(%eax)
14 2c: ff 30                  pushl  (%eax)
```

```

15 2e: 8d 83 00 00 00 00    lea    0x0(%ebx),%eax
                                30: R_386_GOTOFF .LC0
16 34: 50                    push   %eax
17 35: 6a 01                    push   $0x1
18 37: e8 fc ff ff ff        call   38 <main+0x38>
                                38: R_386_PLT32 __printf_chk
19 3c: 83 c4 10                add     $0x10,%esp
20 3f: 8d 65 f8                lea     -0x8(%ebp),%esp
21 42: 59                        pop     %ecx
22 43: 5b                        pop     %ebx
23 44: 5d                        pop     %ebp
24 45: 8d 61 fc                lea     -0x4(%ecx),%esp
25 48: c3                        ret

```

Desmontagem da seção .data:

```

00000000 <x>:
  0: 10 32 54 76
  4: 98 ba dc fe

```

Desmontagem da seção .rodata.str1.1:

```

00000000 <.LC0>:
  0: 79 5b 30 5d 20
  5: 3d 20 25 2e 38
  a: 78 2c 20 79 5b
  f: 31 5d 20
 12: 3d 20 25 2e 38
 17: 78 20 0a 00

```

Desmontagem da seção .text.__x86.get_pc_thunk.bx:

```

00000000 <__x86.get_pc_thunk.bx>:
  0: 8b 1c 24                mov     (%esp),%ebx
  3: c3                        ret

```

a) (5) Qual objetivo é alcançado pelo gcc ao executar call __x86.get_pc_thunk.bx na linha 9? Explique a existência desta função.

b) (5) O símbolo __GLOBAL_OFFSET_TABLE_ é realocado com o tipo R_386_GOTPC. Explique o valor a ser atribuído ao símbolo, seu significado preciso e porque este valor pode ser determinado pelo ligador. Procure descobrir a razão do valor 0x02 colocado inicialmente na posição a ser realocada.

c) (5) Na linha 11 a função f é chamada. Como f é uma rotina do usuário e não parte de uma biblioteca dinâmica, ela está na seção .text, junto com main. Como explicar o significado de f@PLT, já que estamos gerando código PIC.

d) (5) Os termos y@GOT e .LC0@GOTOFF são constantes, mas de tipos de realocação diferentes. Justifique o significado preciso de cada um deles. Qual a vantagem do código PIC em relação a código compilado com -fno-PIC no trabalho de realocação a ser feito pelo ligador, assumindo que no programa main houvesse várias referências ao vetor y?

Questão 3 (30 pontos) Considerando ainda m1.c e m2.c, quando se gera o executável a partir dos módulos compilados apenas com -O1, sem opção -fno-PIC, desmontando a seção .text temos:

...

Desmontagem da seção .text:

```
000011cd <main>:
1      11cd: f3 0f 1e fb      endbr32
2      11d1: 8d 4c 24 04      lea     0x4(%esp),%ecx
3      11d5: 83 e4 f0          and     $0xffffffff0,%esp
4      11d8: ff 71 fc          pushl   -0x4(%ecx)
5      11db: 55              push    %ebp
6      11dc: 89 e5            mov     %esp,%ebp
7      11de: 53              push    %ebx
8      11df: 51              push    %ecx
9      11e0: e8 eb fe ff ff    call    .... <_x86.get_pc_thunk.bx>
10     11e5: 81 c3 f3 2d 00 00  add     $0x2df3,%ebx
11     11eb: e8 26 00 00 00    call    .... <f>
12     11f0: 8d 83 38 00 00 00  lea     0x38(%ebx),%eax
13     11f6: ff 70 04          pushl   0x4(%eax)
14     11f9: ff 30            pushl   (%eax)
15     11fb: 8d 83 30 e0 ff ff  lea     -0x1fd0(%ebx),%eax
16     1201: 50              push    %eax
17     1202: 6a 01            push    $0x1
18     1204: e8 77 fe ff ff    call    1080 <__printf_chk@plt>
19     1209: 83 c4 10          add     $0x10,%esp
20     120c: 8d 65 f8          lea     -0x8(%ebp),%esp
21     120f: 59              pop     %ecx
22     1210: 5b              pop     %ebx
23     1211: 5d              pop     %ebp
24     1212: 8d 61 fc          lea     -0x4(%ecx),%esp
25     1215: c3              ret
...
```

a) (5) Qual o endereço de carga da tabela GOT? Mostre os cálculos, passo a passo, para obter a resposta. Não apresente apenas o resultado final.

b) (5) Calcule os endereços das rotinas _x86.get_pc_thunk.bx e f. Mostre os cálculos realizados.

c) (5) Qual o endereço de carga da lista de controle de printf? Mostre como calculou.

d) (5) Como explicar a substituição do movl por lea na linha 12? Justifique a razão disso. Esta substituição mostra que o ligador e o carregador alteraram o código gerado pelo compilador e o que de fato é executado não é exatamente o que o código de montagem apresenta.

e) Listando os segmentos de carga do executável compilado com -O1, obtemos:

m: formato do arquivo elf32-i386

Cabeçalho do Programa:

```
PHDR off    0x00000034 vaddr 0x00000034 paddr 0x00000034 align 2**2
      filesz 0x00000180 memsz 0x00000180 flags r--
INTERP off   0x000001b4 vaddr 0x000001b4 paddr 0x000001b4 align 2**0
      filesz 0x00000013 memsz 0x00000013 flags r--
LOAD off    0x00000000 vaddr 0x00000000 paddr 0x00000000 align 2**12
      filesz 0x00000418 memsz 0x00000418 flags r--
LOAD off    0x00001000 vaddr 0x00001000 paddr 0x00001000 align 2**12
      filesz 0x000002e4 memsz 0x000002e4 flags r-x
LOAD off    0x00002000 vaddr 0x00002000 paddr 0x00002000 align 2**12
```

```

        filesz 0x000001cc memsz 0x000001cc flags r--
LOAD off 0x00002ed8 vaddr 0x00003ed8 paddr 0x00003ed8 align 2**12
        filesz 0x0000013a memsz 0x0000013c flags rw-
DYNAMIC off 0x00002ee0 vaddr 0x00003ee0 paddr 0x00003ee0 align 2**2
        filesz 0x000000f8 memsz 0x000000f8 flags rw-
NOTE off 0x000001c8 vaddr 0x000001c8 paddr 0x000001c8 align 2**2
        filesz 0x00000060 memsz 0x00000060 flags r--
0x6474e553 off 0x000001ec vaddr 0x000001ec paddr 0x000001ec align 2**2
        filesz 0x0000001c memsz 0x0000001c flags r--
EH_FRAME off 0x00002024 vaddr 0x00002024 paddr 0x00002024 align 2**2
        filesz 0x0000005c memsz 0x0000005c flags r--
STACK off 0x00000000 vaddr 0x00000000 paddr 0x00000000 align 2**4
        filesz 0x00000000 memsz 0x00000000 flags rw-
RELRO off 0x00002ed8 vaddr 0x00003ed8 paddr 0x00003ed8 align 2**0
        filesz 0x00000128 memsz 0x00000128 flags r--

```

e1) (5) Quais os endereços iniciais e finais dos segmentos de código e de dados? Comprove que os endereços dos vetores x e y e a tabela GOT estão de fato no segmento de dados. Indique como estes segmentos são inicializados em memória, de onde vem a informação e o número de bytes efetivos usados na memória.

e2) (5) Qual segmento armazena .rodata? Explique e justifique, analisando os endereços inicial e final, tamanho em bytes e de onde vem o conteúdo deste segmento.

Questão 4 (10 pontos) Pelo desmonte do executável, vemos que printf é chamada com call 0x1080. Listando o conteúdo da PLT e de GOT temos:

00001030 <.plt>:

PLT[0]

```

1030: ff b3 04 00 00 00    pushl 0x4(%ebx)
1036: ff a3 08 00 00 00    jmp *0x8(%ebx)
103c: 0f 1f 40 00          nopl 0x0(%eax)

```

PLT[1]

```

1040: f3 0f 1e fb          endbr32
1044: 68 00 00 00 00 00    push $0x0
1049: e9 e2 ff ff ff      jmp 1030 <.plt>
104e: 66 90              xchg %ax,%ax

```

PLT[2]

```

1050: f3 0f 1e fb          endbr32
1054: 68 08 00 00 00 00    push $0x8
1059: e9 d2 ff ff ff      jmp 1030 <.plt>
105e: 66 90              xchg %ax,%ax

```

...

00001080 <__printf_chk@plt>:

```

1080: f3 0f 1e fb          endbr32
1084: ff a3 10 00 00 00    jmp *0x10(%ebx)
108a: 66 0f 1f 44 00 00    nopw 0x0(%eax,%eax,1)

```

GOT: (os endereços foram apagados)

```

.... e0 3e 00 00      GOT[0]
.... 00 00 00 00      GOT[1]
.... 00 00 00 00      GOT[2]
.... 40 10 00 00      GOT[3]
.... 50 10 00 00      GOT[4]
.... 00 00 00 00      GOT[5]

```

a) (5) Liste exatamente a sequência de instruções de máquina com seus respectivos endereços de carga que são executadas ao chamar printf_chk pela primeira vez, tomando por base o executável desmontado do acima até que o controle seja passado para o ligador dinâmico. Apresente breve justificativa para explicar como desviou para uma determinada instrução e como o endereço de carga da instrução foi obtido. Inicie

pela execução da instrução no endereço 0x1204 do segmento de código. Ao final, indique qual será tarefa a ser realizada pelo ligador dinâmico.

b) (5) Cada processo que chama printf tem a biblioteca carregada em seu espaço virtual em endereços iguais ou não? Como os vários processos podem executar uma mesma cópia do código de printf na memória, simultaneamente? Como o código de printf é de fato carregado na memória? Explique.