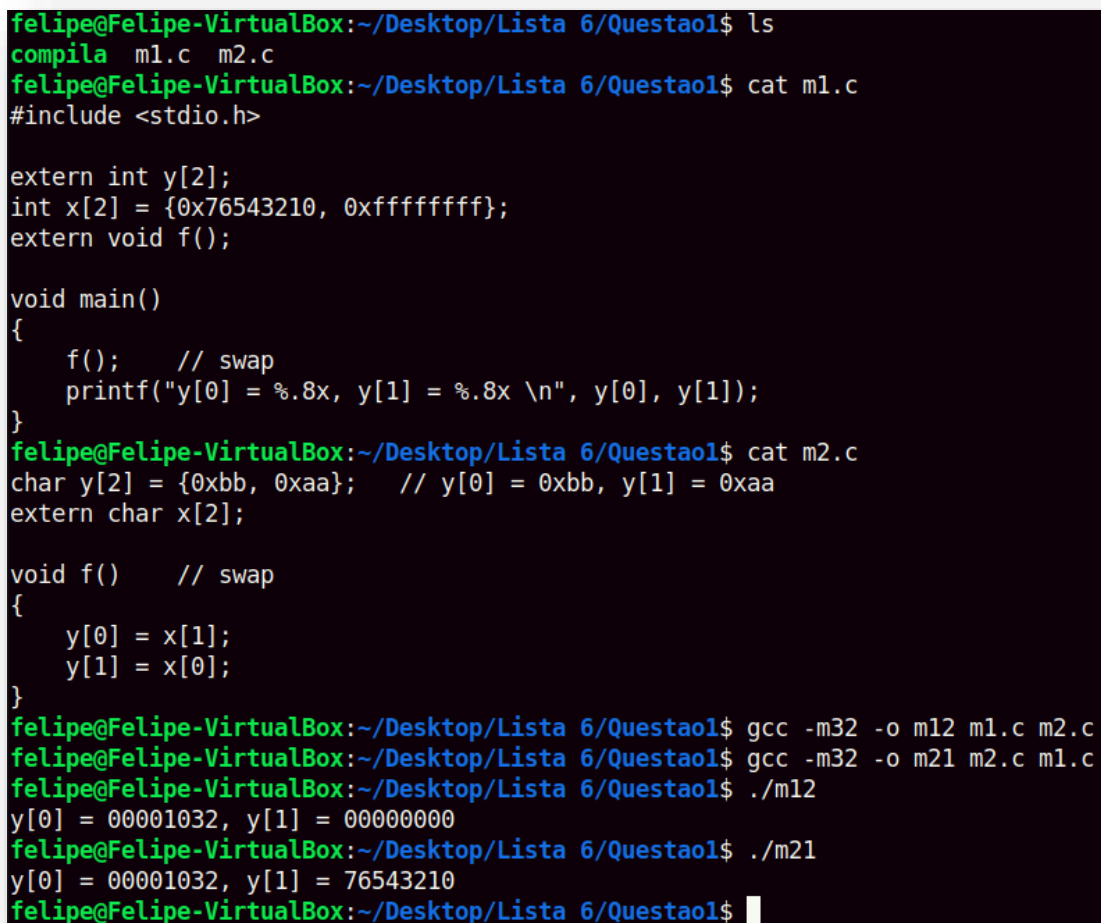


## Lista 6 – Felipe Melo – Thalles Nonato

DRE Felipe: 119093752

DRE Thalles: 119058809

**Questão 1) a) 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:**



```
felipe@Felipe-VirtualBox:~/Desktop/Lista 6/Questao1$ ls
compila m1.c m2.c
felipe@Felipe-VirtualBox:~/Desktop/Lista 6/Questao1$ cat m1.c
#include <stdio.h>

extern int y[2];
int x[2] = {0x76543210, 0xffffffff};
extern void f();

void main()
{
    f();    // swap
    printf("y[0] = %.8x, y[1] = %.8x \n", y[0], y[1]);
}
felipe@Felipe-VirtualBox:~/Desktop/Lista 6/Questao1$ cat m2.c
char y[2] = {0xbb, 0xaa};    // y[0] = 0xbb, y[1] = 0xaa
extern char x[2];

void f()    // swap
{
    y[0] = x[1];
    y[1] = x[0];
}
felipe@Felipe-VirtualBox:~/Desktop/Lista 6/Questao1$ gcc -m32 -o m12 m1.c m2.c
felipe@Felipe-VirtualBox:~/Desktop/Lista 6/Questao1$ gcc -m32 -o m21 m2.c m1.c
felipe@Felipe-VirtualBox:~/Desktop/Lista 6/Questao1$ ./m12
y[0] = 00001032, y[1] = 00000000
felipe@Felipe-VirtualBox:~/Desktop/Lista 6/Questao1$ ./m21
y[0] = 00001032, y[1] = 76543210
felipe@Felipe-VirtualBox:~/Desktop/Lista 6/Questao1$
```

**Questão 1) b) Justifique, com argumentos adequados, 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:**

Para descobrir o tipo das variáveis x e y, basta imprimir a tabela de símbolos (como é pedido no item (c)) e olhar na coluna “Type”. Nela verificamos que o tipo de x e de y é Object. Isso pois x e y são objetos na memória.

Ainda na tabela de símbolos, vemos que o número da seção na qual está x e y é 25, e, utilizando o comando `readelf -S m12` e `readelf -S m21` (como é pedido nos itens (d) e (e)), verificamos que 25 é o número da seção .data, a qual é responsável por armazenar as variáveis globais inicializadas com valor diferente de zero.

Para checar a quantidade de bytes alocada, podemos verificar na tabela de símbolos, na coluna “Size” que a quantidade de bytes alocada para x é 8, e para y é 2. Isso ocorre pois x é um array de inteiros de tamanho 2, com cada posição contendo 4 bytes e y é um array de caracteres de tamanho 2, com cada posição contendo 1 byte.

**Questão 1) c) Imprima a tabela de símbolos de m12 e capture as linhas que mencionam x e y numa mesma tela, bem como o comando utilizado para listar a tabela de símbolos. Faça o mesmo para m21. Analise e conclua como a ordem de compilação afeta a alocação de memória:**

```
felipe@Felipe-VirtualBox:~/Desktop/Lista 6/Questao1$ readelf -s m12
```

Symbol table '.dynsym' contains 8 entries:

Num:	Value	Size	Type	Bind	Vis	Ndx	Name
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

Symbol table '.symtab' contains 74 entries:

Num:	Value	Size	Type	Bind	Vis	Ndx	Name
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	
51:	00000000	0	FUNC	GLOBAL	DEFAULT	UND	printf@GLIBC_2.0
52:	000012d5	0	FUNC	GLOBAL	HIDDEN	16	__x86.get_pc_thunk.bp
53:	00004012	0	NOTYPE	GLOBAL	DEFAULT	25	edata
54:	00004008	8	OBJECT	GLOBAL	DEFAULT	25	x
55:	000012dc	0	FUNC	GLOBAL	HIDDEN	17	fini
56:	00001220	51	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:	00004014	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	2	OBJECT	GLOBAL	DEFAULT	25	y
69:	00004012	0	NOTYPE	GLOBAL	DEFAULT	26	__bss_start
70:	000011cd	83	FUNC	GLOBAL	DEFAULT	16	main
71:	00001253	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

```
felipe@Felipe-VirtualBox:~/Desktop/Lista 6/Questao1$
```

```
felipe@Felipe-VirtualBox:~/Desktop/Lista 6/Questao1$ readelf -s m21
```

Symbol table '.dynsym' contains 8 entries:

Num:	Value	Size	Type	Bind	Vis	Ndx	Name
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

Symbol table '.symtab' contains 74 entries:

Num:	Value	Size	Type	Bind	Vis	Ndx	Name
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	
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	51	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	2	OBJECT	GLOBAL	DEFAULT	25	y
69:	00004014	0	NOTYPE	GLOBAL	DEFAULT	26	__bss_start
70:	00001204	83	FUNC	GLOBAL	DEFAULT	16	main
71:	00001200	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

```
felipe@Felipe-VirtualBox:~/Desktop/Lista 6/Questao1$
```

A ordem de compilação afeta a alocação de memória, pois vemos que como geramos m12 com m1.c antes de m2.c, o endereço de memória de x (0x00004008) vem antes do de y (0x00004010). O mesmo vale para m21, uma vez que o endereço de y (0x00004008) vem antes do de x (0x0000400C).

**Questão 1) d) Liste, numa mesma tela, as seções .data e .bss do executável m12. Indique o comando usado e anexa a captura de 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 detalhadamente:**

```
felipe@Felipe-VirtualBox:~/Desktop/Lista 6/Questao1$ readelf -S m12
There are 31 section headers, starting at offset 0x3858:

Section Headers:
[Nr] Name                Type              Addr             Off              Size             ES Flg Lk Inf Al
[ 0]                      NULL              00000000          000000           000000           00  0  0  0  0
[ 1] .interp                PROGBITS           000001b4          0001b4           000013           00  A  0  0  1
[ 2] .note.gnu.build-id     NOTE               000001c8          0001c8           000024           00  A  0  0  4
[ 3] .note.gnu.property     NOTE               000001ec          0001ec           00001c           00  A  0  0  4
[ 4] .note.ABI-tag          NOTE               00000208          000208           000020           00  A  0  0  4
[ 5] .gnu.hash              GNU_HASH           00000228          000228           000020           04  A  6  0  4
[ 6] .dynsym                DYNSYM             00000248          000248           000080           10  A  7  1  4
[ 7] .dynstr                STRTAB             000002c8          0002c8           00009d           00  A  0  0  1
[ 8] .gnu.version            VERSYM             00000366          000366           000010           02  A  6  0  2
[ 9] .gnu.version_r          VERNEED            00000378          000378           000030           00  A  7  1  4
[10] .rel.dyn                REL                000003a8          0003a8           000040           08  A  6  0  4
[11] .rel.plt                REL                000003e8          0003e8           000010           08  AI 6 24 4
[12] .init                   PROGBITS           00001000          001000           000024           00  AX 0  0  4
[13] .plt                    PROGBITS           00001030          001030           000030           04  AX 0  0 16
[14] .plt.got                PROGBITS           00001060          001060           000010           10  AX 0  0 16
[15] .plt.sec                PROGBITS           00001070          001070           000020           10  AX 0  0 16
[16] .text                   PROGBITS           00001090          001090           000249           00  AX 0  0 16
[17] .fini                   PROGBITS           000012dc          0012dc           000018           00  AX 0  0  4
[18] .rodata                 PROGBITS           00002000          002000           000023           00  A  0  0  4
[19] .eh_frame_hdr           PROGBITS           00002024          002024           00005c           00  A  0  0  4
[20] .eh_frame               PROGBITS           00002080          002080           000158           00  A  0  0  4
[21] .init_array              INIT_ARRAY         00003ed8          002ed8           000004           04  WA 0  0  4
[22] .fini_array              FINI_ARRAY         00003edc          002edc           000004           04  WA 0  0  4
[23] .dynamic                 DYNAMIC            00003ee0          002ee0           0000f8           08  WA 7  0  4
[24] .got                     PROGBITS           00003fd8          002fd8           000028           04  WA 0  0  4
[25] .data                    PROGBITS           00004000          003000           000012           00  WA 0  0  4
[26] .bss                     NOBITS             00004012          003012           000002           00  WA 0  0  1
[27] .comment                 PROGBITS           00000000          003012           00002a           01  MS 0  0  1
[28] .symtab                  SYMTAB             00000000          00303c           0004a0           10  29 47 4
[29] .strtab                  STRTAB             00000000          0034dc           000264           00  0  0  1
[30] .shstrtab                STRTAB             00000000          003740           000118           00  0  0  1

Key to Flags:
W (write), A (alloc), X (execute), M (merge), S (strings), I (info),
L (link order), O (extra OS processing required), G (group), T (TLS),
C (compressed), x (unknown), o (OS specific), E (exclude),
p (processor specific)
felipe@Felipe-VirtualBox:~/Desktop/Lista 6/Questao1$
```

y foi declarado como vetor de caracteres, com valor igual a 0xBBAA. Esse valor foi salvo na seção .data, já que se trata de uma variável global inicializada e y começa apontando para a posição dessa seção que está esse valor. Em f(), tanto y como x estão sendo lidos como caracteres. Então, quando em m2.c vemos y[0] = x[1], ao invés de passar os 4 bytes de x[1] para y[0], é passado apenas o primeiro byte (0x10 tratando-se de Little Endian) de x[0]. Já para y[1], é passado o segundo byte (0x32) de x[0]. Na hora de imprimir os valores, y é considerado um inteiro e os 4 primeiros bytes são lidos para y[0]. No entanto, apenas 2 bytes foram de fato preenchidos, então y[0] será 0x00001032, ao passo que y[1] será 0x00000000.



**Questão 1) e) Liste, numa mesma tela, as seções .data e .bss do executável m21. Indique o comando usado e anexa a captura de 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 detalhadamente:**

```
felipe@Felipe-VirtualBox:~/Desktop/Lista 6/Questao1$ readelf -S m21
There are 31 section headers, starting at offset 0x385c:

Section Headers:
[Nr] Name                Type              Addr             Off             Size            ES Flg Lk Inf Al
[ 0]                      NULL              00000000         000000         000000         00  A  0  0  0
[ 1] .interp                PROGBITS          000001b4         0001b4         000013         00  A  0  0  1
[ 2] .note.gnu.build-id     NOTE              000001c8         0001c8         000024         00  A  0  0  4
[ 3] .note.gnu.property     NOTE              000001ec         0001ec         00001c         00  A  0  0  4
[ 4] .note.ABI-tag          NOTE              00000208         000208         000020         00  A  0  0  4
[ 5] .gnu.hash              GNU_HASH          00000228         000228         000020         04  A  6  0  4
[ 6] .dynsym                DYNSYM            00000248         000248         000080         10  A  7  1  4
[ 7] .dynstr                STRTAB            000002c8         0002c8         00009d         00  A  0  0  1
[ 8] .gnu.version            VERSYM            00000366         000366         000010         02  A  6  0  2
[ 9] .gnu.version_r          VERNEED           00000378         000378         000030         00  A  7  1  4
[10] .rel.dyn               REL               000003a8         0003a8         000040         08  A  6  0  4
[11] .rel.plt               REL               000003e8         0003e8         000010         08  AI 6 24 4
[12] .init                  PROGBITS          00001000         001000         000024         00  AX 0  0  4
[13] .plt                   PROGBITS          00001030         001030         000030         04  AX 0  0 16
[14] .plt.got               PROGBITS          00001060         001060         000010         10  AX 0  0 16
[15] .plt.sec               PROGBITS          00001070         001070         000020         10  AX 0  0 16
[16] .text                  PROGBITS          00001090         001090         000249         00  AX 0  0 16
[17] .fini                  PROGBITS          000012dc         0012dc         000018         00  AX 0  0  4
[18] .rodata                PROGBITS          00002000         002000         000023         00  A  0  0  4
[19] .eh_frame_hdr          PROGBITS          00002024         002024         00005c         00  A  0  0  4
[20] .eh_frame              PROGBITS          00002080         002080         000158         00  A  0  0  4
[21] .init_array             INIT_ARRAY        00003ed8         002ed8         000004         04  WA 0  0  4
[22] .fini_array             FINI_ARRAY        00003edc         002edc         000004         04  WA 0  0  4
[23] .dynamic                DYNAMIC           00003ee0         002ee0         0000f8         08  WA 7  0  4
[24] .got                   PROGBITS          00003fd8         002fd8         000028         04  WA 0  0  4
[25] .data                  PROGBITS          00004000         003000         000014         00  WA 0  0  4
[26] .bss                   NOBITS            00004014         003014         000004         00  WA 0  0  1
[27] .comment               PROGBITS          00000000         003014         00002a         01  MS 0  0  1
[28] .symtab                SYMTAB            00000000         003040         0004a0         10  29 47 4
[29] .strtab                STRTAB            00000000         0034e0         000264         00  0  0  1
[30] .shstrtab              STRTAB            00000000         003744         000118         00  0  0  1

Key to Flags:
W (write), A (alloc), X (execute), M (merge), S (strings), I (info),
L (link order), O (extra OS processing required), G (group), T (TLS),
C (compressed), x (unknown), o (OS specific), E (exclude),
p (processor specific)
felipe@Felipe-VirtualBox:~/Desktop/Lista 6/Questao1$
```

Em m21, ocorre um processo diferente, pois dessa vez m2.c vem antes de m1.c. Agora, o valor salvo para y em m2.c fica antes do inicializado em x. No segundo caso, como m2 inicia primeiro e y é declarado como char, y é colocado na memória inicialmente. Como o compilador, preferencialmente, não coloca conteúdo dividido na memória, os 4 bytes são completados com 0000 (padding), e depois os outros valores de x são alocados, já que após isso há um dado maior que 2 bytes. Portanto, teremos 0xBBAA0000. Ao entrar na função f, como são trocas entre variáveis do tipo char (dados de 1 byte), são selecionados os dois primeiros bytes do inteiro x[0] (para y[0] e y[1]). Dessa forma, y[0] = 32, y[1] = x[0]. Como a impressão é de um inteiro, é impresso, 0x00001032 e a posição seguinte a ele na memória (0x76543210).

**Questão 1) f) Compile m1.c com as opções -m32 -fno-PIC -O1 -c para obter m1.o. Descubra como listar o código de montagem junto com a informação de realocação. É preciso que a informação de realocação esteja inserida no código de montagem. Documente com a captura de tela. Explique e justifique os tipos de realocação assinalados, a posição dos bytes a serem realocados e a razão do conteúdo inicial nesses bytes. Como a entrada de realocação já está dada, interessa que seja explicada a razão de cada entrada:**

```
felipe@Felipe-VirtualBox:~/Desktop/Lista 6/Questao1$ ls
compila m12 m1.c m21 m2.c teste teste.c
felipe@Felipe-VirtualBox:~/Desktop/Lista 6/Questao1$ gcc -m32 -fno-PIC -O1 -c m1.c
felipe@Felipe-VirtualBox:~/Desktop/Lista 6/Questao1$ objdump -r m1.o

m1.o:      file format elf32-i386

RELOCATION RECORDS FOR [.text]:
OFFSET    TYPE              VALUE
00000016  R_386_PC32               f
0000001c  R_386_32                 y
00000022  R_386_32                 y
00000027  R_386_32                 .rodata.str1.1
0000002e  R_386_PC32               __printf_chk

RELOCATION RECORDS FOR [.eh_frame]:
OFFSET    TYPE              VALUE
00000020  R_386_PC32               .text

felipe@Felipe-VirtualBox:~/Desktop/Lista 6/Questao1$
```

A imagem acima é referente à tabela de realocação de m1.o. Como podemos notar, é realizada a realocação da função f e de y (duas vezes, por se tratar de um array de duas posições), da seção .rodata (onde está a lista de saída de \_\_printf\_chk), que é uma chamada a uma biblioteca do sistema. Abaixo podemos ver o código de montagem com as marcações onde foram realizadas essas realocações:

```

felipe@Felipe-VirtualBox:~/Desktop/Lista 6/Questao1$ ls
compila m12 m1.c m21 m2.c m2.o teste teste.c
felipe@Felipe-VirtualBox:~/Desktop/Lista 6/Questao1$ rm m2.o
felipe@Felipe-VirtualBox:~/Desktop/Lista 6/Questao1$
felipe@Felipe-VirtualBox:~/Desktop/Lista 6/Questao1$ ls
compila m12 m1.c m21 m2.c teste teste.c
felipe@Felipe-VirtualBox:~/Desktop/Lista 6/Questao1$ gcc -m32 -fno-PIC -O1 -c m1.c
felipe@Felipe-VirtualBox:~/Desktop/Lista 6/Questao1$ objdump -r -d m1.o

m1.o:          file format elf32-i386


Disassembly of section .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      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    $0x10,%esp
35:  8b 4d fc            mov    -0x4(%ebp),%ecx
38:  c9                  leave
39:  8d 61 fc            lea    -0x4(%ecx),%esp
3c:  c3                  ret
felipe@Felipe-VirtualBox:~/Desktop/Lista 6/Questao1$

```



Como podemos verificar, a função `f` foi realocada logo após ser chamada e após isso, as duas variáveis `y` foram realocadas de maneira estática (feita em tempo de compilação), já que se trata de um tipo de dado com tamanho pré definido (nesse caso, o compilador aloca de forma automática o espaço de memória necessário). Em seguida foi realocado o conteúdo de `.rodata` logo antes de `__printf_chk`.

**Questão 1) g) Compile `m2.c` com as opções `-m32 -fno-PIC -O1 -c` para obter `m2.o`. Descubra como listar o código de montagem junto com a informação de realocação. É preciso que a informação de realocação esteja inserida no código de montagem. Documente com a captura de tela. Explique e justifique os tipos de realocação assinalados, a posição dos bytes a serem realocados e a razão do conteúdo inicial nesses bytes. Como a entrada de realocação já está dada, interessa que seja explicada a razão de cada entrada:**

Abaixo, podemos ver a tabela de realocação de m2.o:

```
felipe@Felipe-VirtualBox:~/Desktop/Lista 6/Questao1$ ls
compila m12 m1.c m1.o m21 m2.c teste teste.c
felipe@Felipe-VirtualBox:~/Desktop/Lista 6/Questao1$ gcc -m32 -fno-PIC -O1 -c m2.c
felipe@Felipe-VirtualBox:~/Desktop/Lista 6/Questao1$ objdump -r m2.o

m2.o:      file format elf32-i386

RELOCATION RECORDS FOR [.text]:
OFFSET    TYPE              VALUE
00000007  R_386_32                 x
0000000c  R_386_32                 y
00000013  R_386_32                 x
00000018  R_386_32                 y

RELOCATION RECORDS FOR [.eh_frame]:
OFFSET    TYPE              VALUE
00000020  R_386_PC32               .text

felipe@Felipe-VirtualBox:~/Desktop/Lista 6/Questao1$
```

Como podemos verificar, as variáveis x e y foram realocadas, visto que são apenas atribuídos valores a elas. Em seguida, serão resgatados na main. Abaixo podemos ver o código de montagem com o momento em que essas realocações foram feitas:

```
felipe@Felipe-VirtualBox:~/Desktop/Lista 6/Questao1$ objdump -r -d m2.o

m2.o:      file format elf32-i386

Disassembly of section .text:

00000000 <f>:
 0:  f3 0f 1e fb          endbr32
 4:  0f b6 05 01 00 00 00  movzbl 0x1,%eax
                        7: R_386_32      x
 b:  a2 00 00 00 00      mov     %al,0x0
                        c: R_386_32      y
10:  0f b6 05 00 00 00 00  movzbl 0x0,%eax
                        13: R_386_32      x
17:  a2 01 00 00 00      mov     %al,0x1
                        18: R_386_32      y
1c:  c3                   ret

felipe@Felipe-VirtualBox:~/Desktop/Lista 6/Questao1$
```



**Questão 2) a) A linha 9 executa um call e transfere o controle para outra parte do código. Ao retornar para executar a linha 10, qual o conteúdo do registrador alterado pela função?**

Há um call para `__x86.get_pc_thunk.bx`. Essa função salva em `%ebx` o endereço de instrução salvo em `%eax`. Uma vez que `%ebx` aponta para esse endereço, ele calcula a distância para a tabela GOT da seção `.data` com a instrução na linha 10 `addl $_GLOBAL_OFFSET_TABLE_, %ebx`. Com isso, `%ebx` passa a apontar para o início da tabela GOT.



**Questão 2) b) A linha 10 do código de montagem soma a constante `_GLOBAL_OFFSET_TABLE_` com `ebx`. No código desmontado, a linha 10 faz a soma de `0x02` com `ebx`. Seria `0x02` o valor dessa constante? Explique o significado da constante e justifique porque o seu valor pode ser determinado em tempo de ligação. Pesquise o significado de `0x02`, se este não for o valor da constante:**

`_GLOBAL_OFFSET_TABLE_` representa a distância em bytes entre o endereço da instrução `addl` e o início da tabela GOT. Como ela foi somada a `%ebx` e depois apontada por `%eax`, esse último registrador passa a apontar para o início da tabela GOT na seção `.data`. Desse modo, o valor dessa constante é 2.



**Questão 2) c) 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?**

`f@PLT` é o endereço de carga da função `f`, carregada na seção `.text` com `main`. Já que o código deve ser compilado de forma a ser independente de posição, o GCC cria um código de montagem modo que se assume que o endereço da carga da função seja referenciado com relação à tabela PLT, que se localiza na seção `.text`. Uma vez que haverá um ligador para definir a posição de `f` na seção, o valor é atualizado no executável sem acessar diretamente a função `f`. Se ela fosse parte de uma biblioteca dinâmica, o acesso seria realizado de forma indireta, via PLT.





**Questão 2) d) Os termos `y@GOT` e `.LC0@GOTOFF` representam constantes? Justifique o significado deles usando o código de montagem como base. Qual a vantagem do código PIC em relação a código compilado com `-fno-PIC` no trabalho de realocação a ser feito pelo ligador?**

Sim, `y@GOT` e `.LC0@GOTFF` representam constantes. Enquanto `y@GOT` é a distância da entrada da tabela GOT alocada para `y`, `.LC0@GOTFF` representa a distância do endereço de memória da lista de controle de `printf` em relação ao início de GOT. A linha 12 indica que buscamos na GOT o endereço de `y` e, portanto, `y@GOT` é o deslocamento do início da tabela GOT até a entrada de `y` na GOT.



**Questão 2) e) Justifique o conteúdo da seção `.rodata`. Mostre a equivalência da memória com ASCII de forma precisa, caractere a caractere, byte a byte:**

```
felipe@Felipe-VirtualBox:~/Desktop/Lista 6/Questao1$ objdump -s -j .rodata m1
m1:      file format elf32-i386

Contents of section .rodata:
 2000 03000000 01000200 795b305d 203d2025  ....y[0] = %
 2010 2e38782c 20795b31 5d203d20 252e3878  .8x, y[1] = %.8x
 2020 200a00                ..
felipe@Felipe-VirtualBox:~/Desktop/Lista 6/Questao1$
```



<code>0x795B305D</code> = “ <code>y[0]</code> ”	<code>0x203D2025</code> = “ <code>= %</code> ”	<code>0x2E38782C</code> = “ <code>.8x,</code> ”	<code>0x20795B31</code> = “ <code>y[1]</code> ”
<code>0x5D203D20</code> = “ <code>] =</code> ”	<code>0x252E3878</code> = “ <code>%.8x</code> ”	<code>0x200A00</code> = “ ”	

Equivalência: `y[0] = %.8x, y[1] = %.8x`

### Questão 3) a) Qual o endereço de carga da tabela GOT? Mostre os cálculos, passo a passo, para obter a resposta:

\_\_x86.get\_pc\_thunk.bx retorna em %ebx o endereço 0x11E5. Efetuando  $0x11E5 + 0x2DF3$ , obtemos o endereço inicial da tabela GOT, que é 0x3FD8.



### Questão 3) b) Calcule os endereços das rotinas \_\_x86.get\_pc\_thunk.bx e f. Mostre os cálculos realizados:

O cálculo é relativo ao PC. Para calcular o endereço da rotina \_\_x86.get\_pc\_thunk.bx efetuamos o seguinte cálculo:  $0x11E5 + 0xFFFFFEEB = 0x10D0$ .



Já o da rotina f é  $0x11F0 + 0x26 = 0x1216$ .

### Questão 3) c) Qual o endereço de carga da lista de controle de printf?

Na linha 18, vemos a instrução `call 1080 <__printf_chk@plt>`. Na linha 15, `lea -0x1FD0(%ebx), %eax` mostra que o endereço da lista de controle está em %eax, pois é parâmetro 1 do printf. Além disso, como %ebx está com o endereço inicial de GOT, o comando está realizando um `.LC0@GOTOFF`, de certa forma, efetuando  $0x3FD8 - 0x1FD0 = 0x2008$ .



### Questão 3) d) Como pode ser justificada a diferença entre os códigos de montagem pelo assembler e a desmontagem do executável, onde no primeiro temos a instrução `movl y@GOT(%ebx), %eax` e no segundo temos a instrução `lea 0x38(%ebx), %eax`? Justifique a alteração de instrução. Isso mostra que o que é executado pode não ser exatamente o código de montagem gerado pelo compilador. Apenas a desmontagem do executável mostra a realidade da execução:

y está no executável, em .data, então, sabendo disso, podemos trocar a soma de %eax com o valor para obter o endereço de y por meio dele mesmo, evitando acesso à memória (utilizando y@GOT). Fazendo a instrução `lea`, o endereço de y é carregado diretamente para %eax, o que aumenta a eficiência.



**Questão 3) e<sub>1</sub>) Quais os endereços iniciais e finais dos segmentos de código e dados? Comprove que os endereços dos vetores x e y e a tabela GOT estão de fato no segmento de dados. Indique como esses segmentos são inicializados em memória, de onde vem a informação e o número de bytes efetivos usados na memória:**

O segmento de código r-x tem início no endereço 0x1000 e fim no endereço 0x12E3. O segmento rw- tem início no endereço 0x3ED8 e fim no endereço 0x4013.



**Questão 3) e<sub>2</sub>) Qual segmento armazena .rodata? Explique e justifique, analisando os endereços inicial e final, tamanho em bytes e de onde vem o conteúdo desse segmento:**

O segmento que armazena .rodata é r--, que tem origem no endereço 0x2000 e fim no endereço 0x21CC. Como no item (c) foi calculado o endereço no qual a lista de `printf` se encontra (0x2008), esse é o segmento que armazena .rodata. a distância de 0x2000 a 0x21CC é 0x1CC. Isso equivale a 1.5 byte, mas é arredondado para 2 bytes.



**Questão 4) a) Quando chamamos `printf` (no caso `printf_chk`) pela primeira vez, queremos entender como o controle é passado para `printf`, quem passa esse controle, que modificações são feitas nas tabelas GOT e PLT e quem as faz. Lembre-se que a desmontagem é apenas do arquivo objeto ainda em disco e não em execução. Seja claro se alguma informação tem que ser alterada na PLT ou em GOT antes do início da execução em main. Usando a informação acima, reproduza o passo a passo nesta primeira chamada com detalhes, explicando e listando cada instrução executada até que o controle seja de fato passado a `printf`:**

No início do processo, GOT[2] é carregado com o endereço do ligador dinâmico e então, quando `printf` é chamado, o esse ligador é executado, carrega em GOT[3] o endereço de `printf`. Dessa forma, a partir da segunda chamada, `printf` passa a ser executada diretamente.



**Questão 4) b) printf é uma biblioteca dinâmica do sistema e pode estar ou não carregada em memória quando é chamada pela primeira vez pelo executável acima. Se ela já estiver carregada e sendo usada por outro processo, o código de printf terá que ser carregado nos mesmos endereços no espaço virtual de cada um dos processos ou poderá estar carregado em endereços virtuais diferentes? Haverá mais de uma cópia de printf carregada na memória física ou não? Explique:**

O carregador dinâmico carregará a biblioteca na memória caso ela já não esteja, que no caso é quando ela é chamada pela primeira vez. Pela biblioteca ser dinâmica, ela não precisa ser carregada de novo toda vez que for chamada, pois ela já é ligada pelo ligador dinâmico. A biblioteca `printf` será carregada no espaço virtual dos arquivos, mas não é necessário que haja correlação do endereço de carga entre eles. Cada processo apontará para uma mesma biblioteca sem que a necessidade de criar múltiplos processos físicos. Essa gerência de memória virtual permite que uma mesma cópia do processo de `printf` seja acessada por vários outros processos. Em outras palavras, a biblioteca dinâmica `printf` terá apenas uma cópia carregada na memória física.

