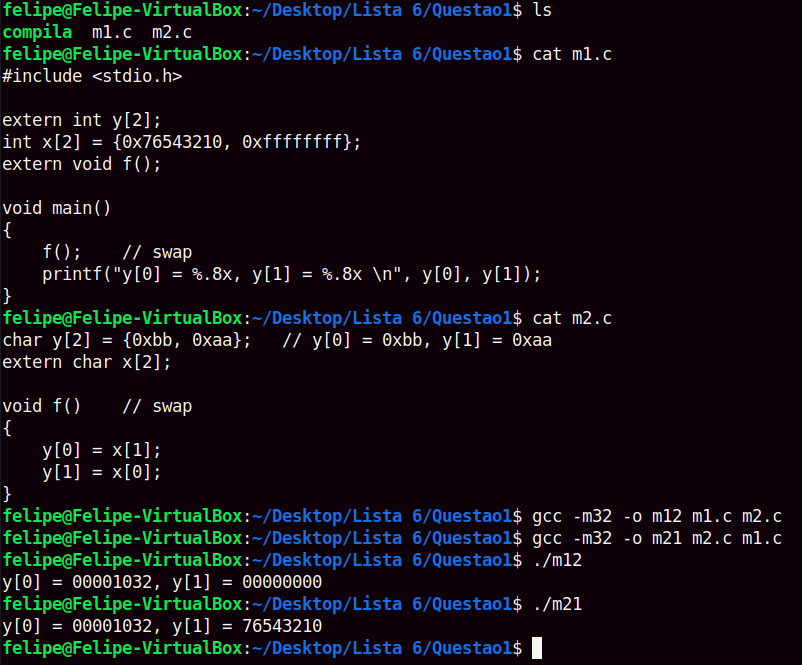
Lista 6 – Felipe Melo – Thalles Nonato

DRE Felipe: 119093752

DRE Thalles: 119058809

**Questão 1) a) Gere o executável m12 com “gcc m-32 -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 excute 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:**



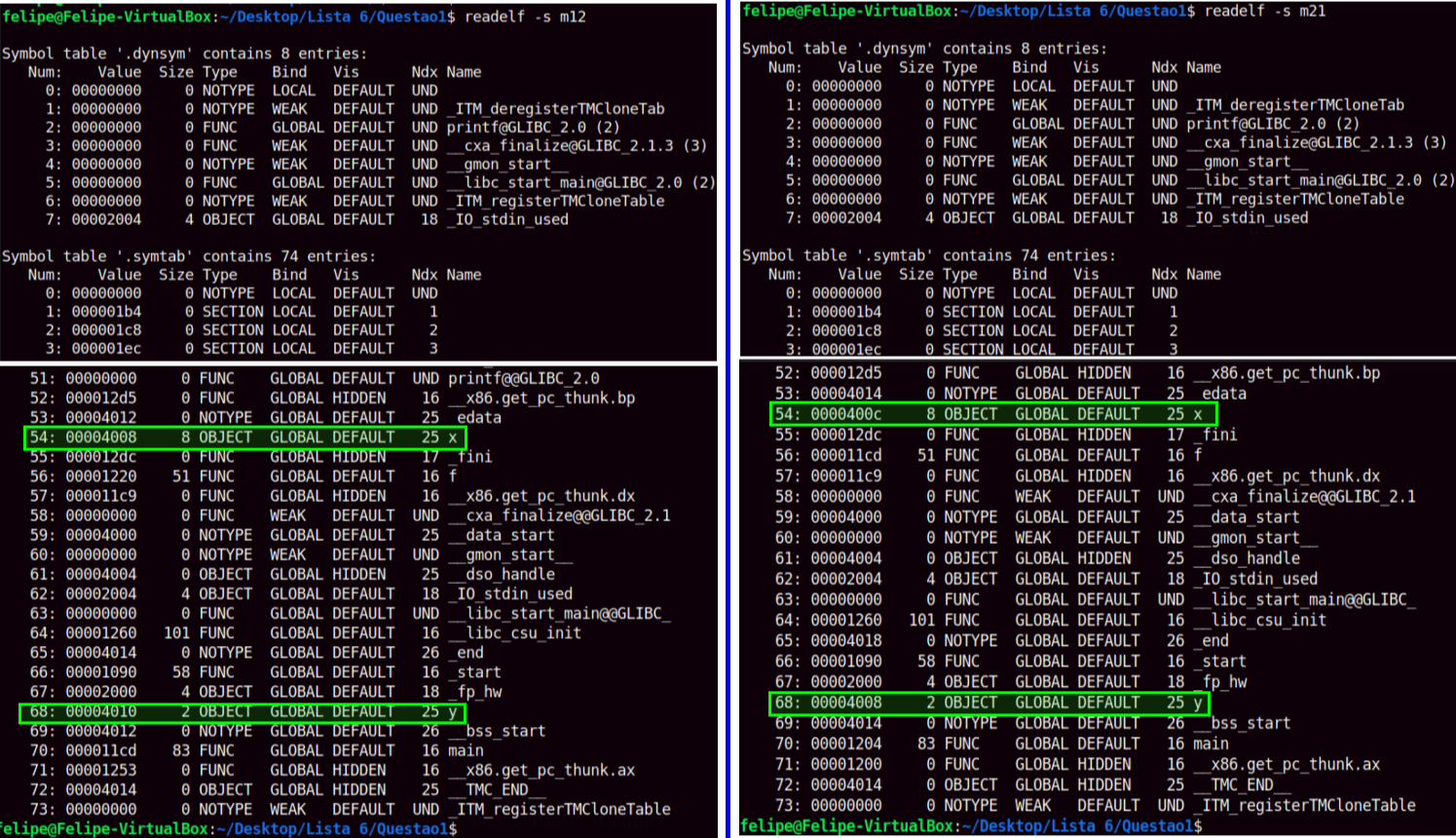
**Questão 1) b) Justifique, com argumentos adequados, a atribuição de memória (tipo da variável, endereço na memória e quantidade de bytes alocada) feita pelo GCC para os vetores x e y. Só mostrar a captura não é suficiente, pois o fundamental é explicar e justificar as atribuições feitas às variáveis 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.

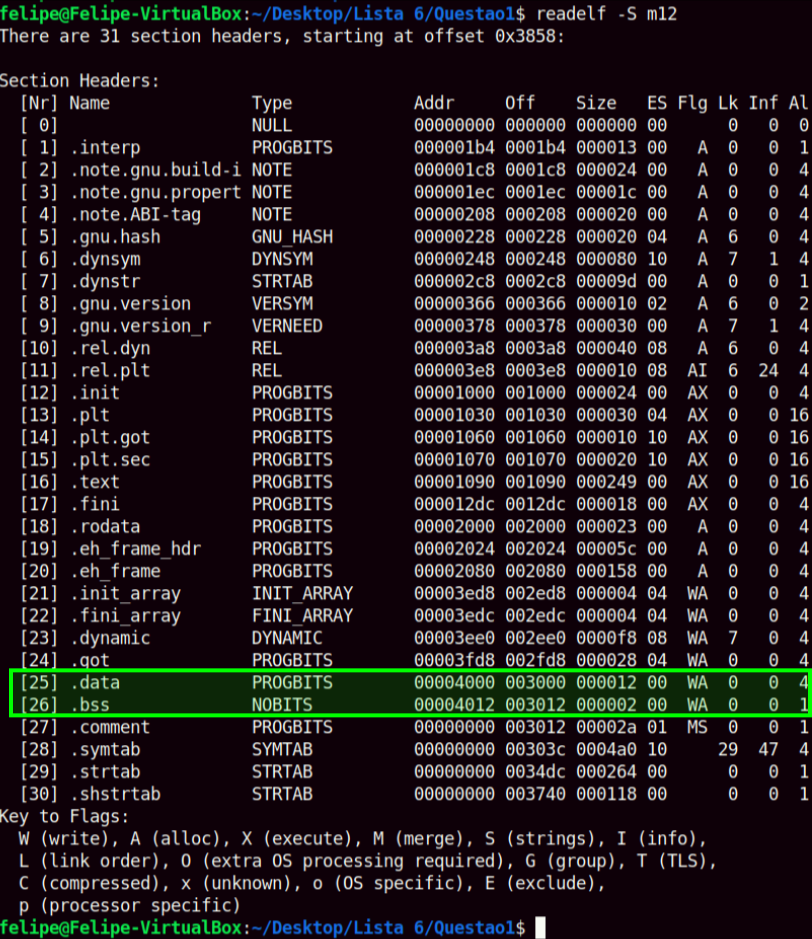
Para checar a quantidade de byes alocada, podemos verificar na tabela de símbolos, na coluna “Size” que a quantidade de bytes alocada para x é 8, e para y é 2. Isso ocorre pois x é um array de inteiros de tamanho 2, com cada posição contendo 4 bytes e y é um array de caracteres de tamanho 2, com cada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



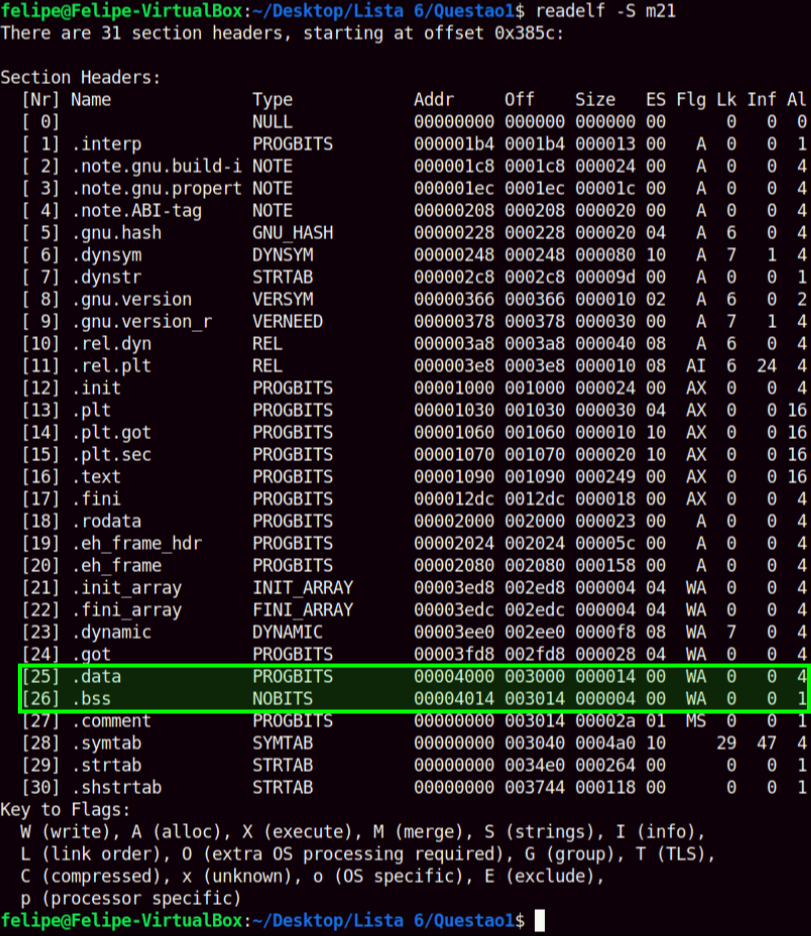
A ordem de compilação afeta a alocação de memória, pois vemos que como geramos m12 com m1.c antes de m2.c, o endereço de memória de x (0x00004008) vem antes do de y (0x00004010). O mesmo vale para m21, uma vez que o endereço de y (0x00004008) vem antes do de x (0x0000400C).

**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:**



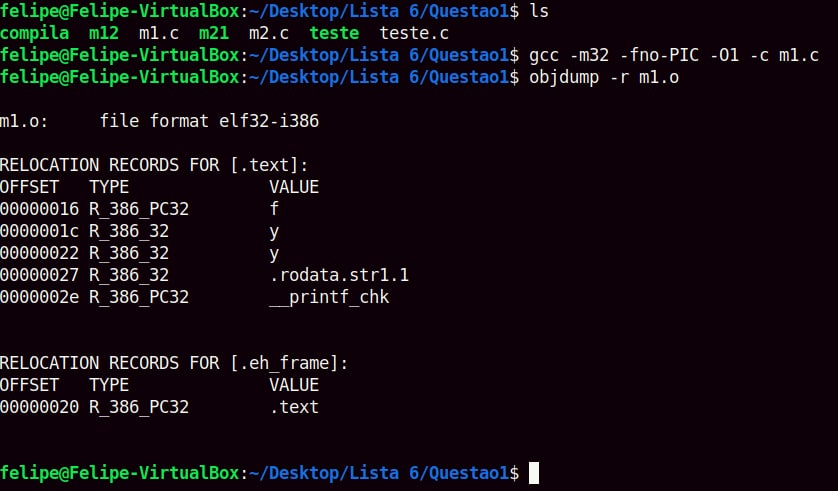
y foi declarado como vetor de caracteres, com valor igual a 0xBBAA. Esse valor foi salvo na seção .data, já que se trata de uma variável global inicializada e y começa apontando para a posição dessa seção que está esse valor. Em f(), tanto y como x estão sendo lidos como caracteres. Então, quando em 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:**

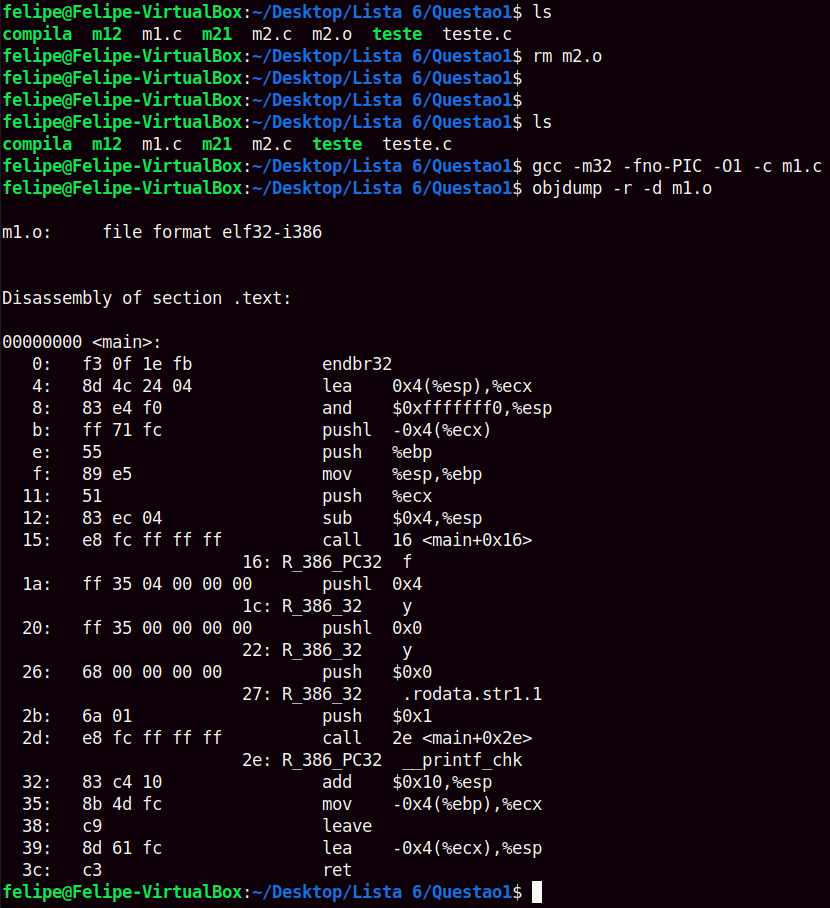


Em m21, ocorre um processo diferente, pois dessa vez m2.c vem antes de m1.c. Agora, o valor salvo para y em m2.c fica antes do inicializado em x. No segundo caso, como m2 inicia primeiro e y é declarado como char, y é colocado na memória inicialmente. Como o compilador, preferencialmente, não coloca 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 haverá 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:**



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:



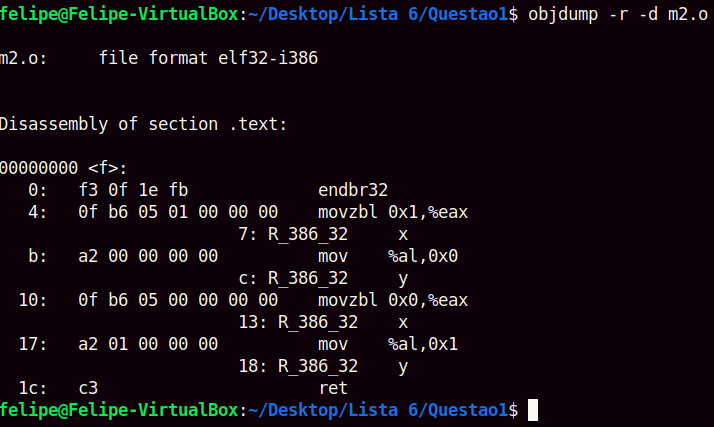
Como podemos verificar, a função f foi realocada logo após ser chamada e após isso, as duas variáveis y foram realocadas de maneira estática (feita em tempo de compilação), já que se trata de um tipo de dado com tamanho pré definido (nesse caso, o compilador aloca de forma automática o espaço de memória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



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:



**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.

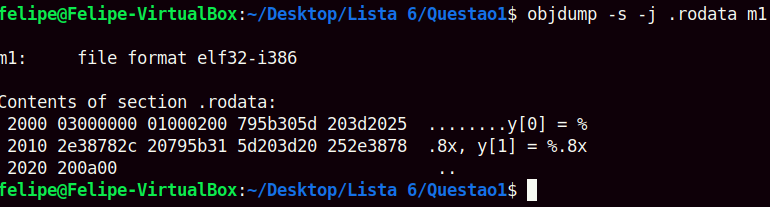
**Questão 2) c) Na linha 11 a função f é chamada. Como f é uma rotina do usuário e não parte de uma biblioteca dinâmica, ela está na seção .text, junto com main. Como explicar o significado de f@PLT, já que estamos gerando código PIC?**

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:**



0x795B305D = “y[0]” 0x203D2025 = “ = %” 0x2E38782C = “.8x,” 0x20795B31 = “y[1”

0x5D203D20 = “] = ” 0x252E3878 = “%.8x” 0x200A00 = “ ”

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) e1) 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) e2) 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.