TPC8 e Guião Laboratorial

Resolução/explicação dos exercícios

1.b) Identifique as instruções responsáveis pelo ciclo `for` na função `soma`. Qual seria a diferença se a constante N fosse um argumento da função?

```
.L6:
    addl    (%ecx,%edx,4), %eax
    incl    %edx
    cmpl    $99, %edx
    jle     .L6
```

O compilador implementou uma estrutura de controlo muito simples (e otimizada), cujo C equivalente seria:

```
do {
  /* corpo do for */
  i++;
} while (i<=99);    // 99 = 0x63
```

Em primeiro lugar, o compilador utilizou a constante `$99` diretamente na instrução `cmpl` (nota: esse valor faz parte do código máquina da instrução, i.e., `83 fa 63`, porque é uma constante). Caso o valor fosse um argumento da função teria que ser copiado da pilha (i.é., da memória) para um registo e seria usado esse registo no `cmpl`.

Em segundo lugar, o compilador removeu o teste ***if i>=N*** antes do `do ... while`, porque, ao analisar o código, i inicia a 0, o que é <100, logo esse teste não é necessário. (nota: um ciclo *for* em C tem a semântica: `while(i<N) { ... }`, ou seja, se `i>=N` não executa o ciclo). Caso o N fosse um argumento o seria necessário incluir esse teste (por exemplo, para o caso de N=0).

1.c) Quantos *bytes* ocupa o vetor? Em que zona de memória está alocado o vetor e qual é a instrução no *assembly* gerado que reserva espaço para o vetor? Indique a instrução *assembly* que liberta esse espaço?

O vetor ocupa `100 x sizeof(int)`, ou seja, 400 bytes. Esse vetor é alocado na pilha, uma vez que é uma variável local à função `main`. Analisando o *assembly* da parte inicial da função:

```
pushl   %ebp
movl    %esp, %ebp
pushl   %ebx
subl    $404, %esp
andl    $-16, %esp
subl    $12, %esp
```

Neste código são reservados primeiramente 404 bytes que corresponde a espaço para 101 elementos sendo posteriormente ajustado o endereço da pilha para um endereço múltiplo de 16 (colocando os últimos 4 bits do endereço a 0, pois `$-16 = 0xfffffff0`).

A instrução que liberta o espaço é o `leave` que aparece no final do *assembly* da função. Esta instrução coloca `%esp` igual a `%ebp`, que liberta o espaço reservado na pilha dessa função.

Neste exercício é importante compreender que a reserva e libertação do espaço para o vetor `v` é feita na pilha, de forma automática (ou seja, o compilador gera as instruções para esse efeito).

1.d) Aumente sucessivamente o N 100 vezes (i. é., acrescente dois zeros de cada vez) e execute o programa até a execução gerar um "`Segmentation fault`". Utilize o gdb para identificar o ponto do programa onde ocorreu o erro e identificar a origem do problema.

Quando N>=10000000 (nota: compilar o código com -g para facilitar a depuração com o `gdb`):

```
[xxxxx@sc TPC8]$ gcc -g -O2 vectInt.c
[xxxxx@sc TPC8]$ ./a.out
Segmentation fault
```

Executar o programa dentro do gdb:

```
[xxxxx@sc TPC8]$ gdb a.out
(gdb) run
Starting program: /home/xxxxx/TPC8/a.out

Program received signal SIGSEGV, Segmentation fault.
0x0804838e in main () at vectInt.c:17
17      init(v);
(gdb)
```

Este erro indica que o programa acedeu a uma posição de memória para a qual não tinha permissão. O `gdb` indica ainda que foi na linha 17 do programa C (no seu código pode ser diferente), mais especificamente na chamada à função `init`. Este resultado indica, ainda, que o `%EIP` tinha o valor `0x0804838e`

Para obter mais informação, é necessário ver o *assembly* do `main`.

```
(gdb) disas main
Dump of assembler code for function main:
0x08048378 <main+0>:    push   %ebp
0x08048379 <main+1>:    mov    %esp,%ebp
0x0804837b <main+3>:    push   %ebx
0x0804837c <main+4>:    sub    $0x2625a04,%esp
0x08048382 <main+10>:   and    $0xfffffff0,%esp
0x08048385 <main+13>:   sub    $0xc,%esp
0x08048388 <main+16>:   lea    0xfd9da5f8(%ebp),%ebx
0x0804838e <main+22>:   push   %ebx
0x0804838f <main+23>:   call   0x8048344 <init>
```

A instrução `push %ebx` é a primeira instrução, após reservar o espaço, que tenta escrever na pilha. Ao tentar escrever num endereço fora da zona de memória reservada para o programa, é lançado um erro (`Segmentation Fault`).

1e) Identifique as instruções responsáveis pelo cálculo do endereço de `vec[i]` na função `soma`. Qual seria a diferença se os elementos do vetor fossem do tipo `double`?

No *assembly* apresentado em 1b) temos:

```
add     (%ecx,%edx,4),%eax
```

O registo `%ecx` contém o argumento `vec`, o registo `%edx` tem o valor de i e 4 é o tamanho de um inteiro (32 bits). No caso de um vetor de *double* o 4 seria substituído por 8 (64 bits, norma IEEE precisão dupla).

2.b) Identifique e explique as instruções responsáveis pelo cálculo do endereço de `vec[i].a`. Compare com a resposta à alínea e) da questão 1.

```
add     4(%ecx,%edx,8),%eax
```

O fator de escala passou a ser 8, o deslocamento de 4 é necessário pois o campo `"a"` está deslocado 4 posições em relação ao início de cada estrutura.

2.c) Qual o espaço ocupado por cada elemento do vetor. Qual será o espaço ocupado se o tamanho do campo **s** da estrutura aumentar para 5 caracteres.

Cada elemento ocupa 8 bytes, 4 do campo `"s"` e 4 do campo `"a"`. Com a alteração do campo seria necessário mais um byte para armazenar a estrutura. Contudo esta alteração obriga a um aumento de bytes devido a alinhamento de dados.

2.d) Verifique qual o espaço efetivamente alocado para cada elemento do vetor quando o campo **s** da estrutura tem 5 caracteres. Para isso, modifique no código C para imprimir o tamanho da estrutura (i.é., `printf("Size of struct %d\n",sizeof(struct S));`).

`"Size of struct 12"` -> a estrutura necessita de 9 bytes para os elementos, o resto é para alinhar a estrutura na memória.

2.e) Identifique e explique as instruções responsáveis pelo cálculo do endereço de `vec[i].a` no caso anterior e compare com a resposta na alínea b).

A implementação mais direta para este caso seria utilizar um fator de escala de 12, o que seria:

```
add     8(%ecx,%edx,12),%eax
```

O problema é que este fator de escala não é suportado (só suporta 1, 2, 4, 8 e 16 pois são mais "comuns"). Assim o compilador optou por gerar um código diferente, baseado em apontadores:

```
        movl  8(%ebp), %edx
        xorl  %eax, %eax        # soma =0
        addl  $8, %edx          # %edx aponta para &v[0].a
        movl  $99, %ecx         # i=99
.L14:
        addl  (%edx), %eax      # %eax = *ptr
        addl  $12, %edx         # %ptr++   // passa para elemento seguinte
        decl  %ecx
        jns   .L14              # repete se i>=0
```