Assembly do IA-32 em ambiente Linux

Exercícios de Programação 4

Resolução

Exercício 4.1 (*Localização de elementos de um array*):

Este exercício permite testar o conhecimento que se tem sobre tamanho dos dados e indexação de elementos de *arrays*. De notar que qualquer apontador em IA-32 tem o tamanho de 4 *bytes*.
A norma ANSI C inclui `long double` como tipo de dados, e a maioria das combinações máquina-compilador implementa estes dados com o mesmo formato que `double`, i.e., com 8 *bytes*. O suporte de GCC para o tipo de dados `long double` para a arquitectura IA-32 (e apenas para esta!) utiliza a representação estendida de vírgula flutuante (10 *bytes*). Contudo, para melhorar o desempenho do sistema de memória, a implementação de `long double` em GCC usa 12 *bytes* para armazenar cada valor, mesmo sabendo que o formato usado nos cálculos apenas requer 10 *bytes*.

```
short         S[7];
short        *T[3];
short       **U[6];
long double   V[8];
long double  *W[4];
```

Array	Element Size	Total Size	Start Address	Element ***i***
`S`	2	28	x_S	$x_S + 2$ `i`
`T`	4	12	x_T	$x_T + 4$ `i`
`U`	4	24	x_U	$x_U + 4$ `i`
`V`	12	96	x_V	$x_V + 12$ `i`
`W`	4	16	x_W	$x_W + 4$ `i`

Exercício 4.2 (*Apontadores*):

Este problema é uma variante de um outro mostrado anteriormente para o *array* de inteiros E. É importante compreender a diferença entre um apontador, e o objecto que se está a apontar. Uma vez que o tipo de dados `short` precisa de 2 *bytes*, todos os índices de *arrays* deverão ser afectados por um factor de escala de 2. E em vez de se usar `movl` como anteriormente, agora dever-se-á usar `movw`.

Expression	Type	Value	Assembly Code
`S+1`	`short *`	x_S + 2	`leal 2(%edx),%eax`
`S[3]`	`short`	*Mem*[x_S + 6]	`movw 6(%edx),%ax`
`&S[i]`	`short *`	x_S + 2 i	`leal (%edx,%ecx,2),%eax`
`S[4*i+1]`	`short`	*Mem*[x_S + 8 i + 2]	`movw 2(%edx,%ecx,8),%ax`
`S+i-5`	`short *`	x_S + 2 i - 10	`leal -10(%edx,%ecx,2),%eax`

Exercício 4.3 (*Matriz de 2 dimensões*):

Este problema pede que se façam os cálculos dos endereços com os factores de escala adequados, e que se aplique a fórmula de indexação orientada à linha (*row-major*).

```
1 int mat1[M][N];
2 int mat2[N][M];
3
4 int sum_element(int i, int j)
5 {
6    return mat1[i][j] + mat2[j][i];
7 }
```

O primeiro passo é comentar o código *assembly* para se determinar como são calculadas as referências à memória:

```
1    movl   8(%ebp),%ecx              Obter i (%ecx)
2    movl   12(%ebp),%eax             Obter j
3    leal   0(,%eax,4),%ebx           4*j (%ebx)
4    leal   0(,%ecx,8),%edx           8*i
5    subl   %ecx,%edx                 7*i (%edx)
6    addl   %ebx,%eax                 5*j
7    sall   $2,%eax                   4*(5*j), =20*j (%eax)
8    movl   mat2(%eax,%ecx,4),%eax    mat2[(20*j + 4*i)/4]
9    addl   mat1(%ebx,%edx,4),%eax    + mat1[(4*j + 28*i)/4] (%eax, valor de retorno)
```

Note que para um *array* declarado como ***T*** `D[`***R***`][`***C***`],` `o` elemento do *array* `D[i][j]` `está` localizado na memória em $x_D + L(C*i + j)$, onde ***L*** `é o tamanho do tipo de dados`***T*** em *bytes*.
Daqui podemos ver que a referência ao elemento da matriz `mat1[i][j]` é feita com um deslocamento de `4(7i + j)` *bytes* relativamente à localização de base da matriz `mat1`, enquanto a referência ao elemento da matriz `mat2[j][i]` é feita com um deslocamento de `4(5j + i)` *bytes* relativamente à localização de base da matriz `mat2`. Daqui, aplicando a fórmula, podemos determinar que `mat1` tem 7 colunas enquanto `mat2` tem 5, dando `M=5` e `N=7.`

Exercício 4.4 (*Structures*):

Este problema leva-nos a pensar na organização de uma *structure*, e no código usado para aceder aos seus campos. A declaração da *structure* é uma variante do exemplo apresentado no texto; mostra que *structures* aninhadas são colocadas na memória, embebendo-as no interior das *structures* mais externas.

```
struct prob {
     int *p;
     struct {
          int x;
          int y;
     } s;
     struct prob *next;
};
```

a) A organização da memória alocada a esta *structure* é a seguinte:

Deslocamento	0	4	8	12
Conteúdo	`p`	`s.x`	`s.y`	`next`

b) 16 *bytes*

c) Como sempre, começa-se por comentar o código *assembly*:

```
1    movl   8(%ebp),%eax          Obter sp
2    movl   8(%eax),%edx          Obter sp->s.y
3    movl   %edx,4(%eax)          Copiar para sp->s.x
4    leal   4(%eax),%edx          Obter &(sp->s.x)
5    movl   %edx,(%eax)           Copiar para sp->p
6    movl   %eax,12(%eax)         sp->next = sp
```

A partir daqui, pode-se gerar o seguinte código C:

```
void sp_init(struct prob *sp)
{
     sp->s.x = sp->s.y ;
     sp->p = &(sp->s.x) ;
     sp->next = sp ;
}
```