O'Hallaron: CS:APP, 2ª Ed.

Signatura: ESIIT/C.1 BRY com

Capítulo 3: Representación de Programas a nivel de máquina

Problemas Prácticos T2.4:

3.50-3.53, pp.318, 323-25

3.35-3.39, pp.267-68, 270, 272, 277

3.50. Una función C `incrprob` tiene argumentos `q`, `t`, y `x` de diferentes tamaños, y cada uno puede que sea con signo o sin signo. La función tiene el siguiente cuerpo:

```
*t += x;
*q += *t;
```

Compila al siguiente código x86_64:

```
1       incrprob:
2         addl    (%rdx), %edi
3         movl    %edi, (%rdx)
4         movslq  %edi, %rdi
5         addq    %rdi, (%rsi)
6         ret
```

Determinar los ocho prototipos de función válidos para `incrprob` determinando el orden y tipos posibles de los tres parámetros.

3.51. Para el programa C:

```
long int local_array(int i)
{
    long int a[4] = {2L, 3L, 5L, 7L};
    int idx = i & 3;
    return a[idx];
}
```

GCC genera el siguiente código:

```
  implementación x86_64 de local_array
  Argumento: i en %edi
1       local_array:
2         movq    $2, -40(%rsp)
3         movq    $3, -32(%rsp)
4         movq    $5, -24(%rsp)
5         movq    $7, -16(%rsp)
6         andl    $3, %edi
7         movq    -40(%rsp,%rdi,8), %rax
8         ret
```

A. Dibujar un diagrama indicando las posiciones de pila usadas por esta función y sus desplazamientos relativos al puntero de pila.
B. Anotar el código ensamblador para describir el efecto de cada instrucción.
C. ¿Qué característica interesante ilustra este ejemplo sobre la disciplina de pila en x86-64?

3.52. Para el programa factorial recursivo

```
long int rfact(long int x)
{
    if (x <= 0)
        return 1;
    else {
       long int xm1 = x-1;
        return x * rfact(xm1);
    }
}
```

GCC genera el siguiente código:

```
  implementación x86_64 de función factorial recursiva rfact
  Argumento: x en %rdi
1     rfact:
2       pushq  %rbx
3       movq   %rdi, %rbx
4       movl   $1, %eax
5       testq  %rdi, %rdi
6       jle    .L11
7       leaq   -1(%rdi), %rdi
8       call   rfact
9       imulq  %rbx, %rax
10    .L11:
11      popq   %rbx
12      ret
```

A. ¿Qué valor almacena la función en `%rbx`?
B. ¿Cuál es el propósito de las instrucciones `pushq` (línea 2) y `popq` (línea 11)?
C. Anotar el código ensamblador para describir el efecto de cada instrucción.
D. ¿En qué se diferencia el manejo del marco de pila que hace esta función del de otras que hemos visto?

3.53. Para cada una de las siguientes declaraciones de estructuras, determinar el desplazamiento de cada campo, el tamaño total de la estructura, y sus requisitos de alineamiento bajo x86-64.

A. `struct P1 { int i; char c; long j; char d; };`
B. `struct P2 { long i; char c; char d; int j; };`
C. `struct P3 { short w[3]; char c[3]; };`
D. `struct P4 { short w[3]; char *c[3]; };`
E. `struct P5 { struct P1 a[2]; struct P2 *p };`

3.35. Considerar las siguientes declaraciones:

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

Rellenar la siguiente tabla describiendo el tamaño de elemento, el tamaño total, y la dirección del elemento *i* para cada uno de estos arrays:

Array	Tam.elemento	Tam.total	Dirección inicio	Elemento *i*
S			x_S	
T			x_T	
U			x_U	
V			x_V	
W			x_W	

3.36. Suponer que la dirección del array de enteros `short S` (x_S) y el índice entero *i* están almacenados en los registros `%edx` y `%ecx`, respectivamente. Para cada una de las siguientes expresiones, indicar su tipo, una fórmula para su valor, y una implementación en código ensamblador. El resultado debería almacenarse en el registro `%eax` si es un puntero y en el elemento de registro `%ax` si es un entero `short`.

Expresión	Tipo	Valor	Código ensamblador
S+1			
S[3]			
&S[i]			
S[4*i+1]			
S+i-5			

3.37. Considerar el siguiente código fuente, donde *M* y *N* son constantes declaradas con `#define`:

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

Al compilar este programa, GCC genera el siguiente código ensamblador:

```
  i en %ebp+8, j en %ebp+12
1        movl    8(%ebp), %ecx
2        movl    12(%ebp), %edx
3        leal    0(,%ecx,8), %eax
4        subl    %ecx, %eax
```

```
5         addl    %edx, %eax
6         leal    (%edx,%edx,4), %edx
7         addl    %ecx, %edx
8         movl    mat1(,%eax,4), %eax
10        addl    mat2(,%edx,4), %eax
```

Usar las habilidades personales de ingeniería inversa para determinar los valores de *M* y *N* basándose en este código ensamblador.

3.38. El siguiente código C ajusta los elementos diagonales de uno de nuestros arrays de tamaño fijo a `val`:

```
1     /* Ajustar todos los elementos diagonales a val */
2     void fix_set_diag(fix_matrix A, int val){
3         int i;
4         for (i = 0; i < N; i++)
6             A[i][i] = val;
11    }
```

Cuando se compila, GCC genera el siguiente código ensamblador:

```
  A en %ebp+8, val en %ebp+12
1         movl    8(%ebp), %ecx
2         movl    12(%ebp), %edx
3         movl    $0, %eax
4       .L14:
5         movl    %edx, (%ecx,%eax)
6         addl    $68, %eax
7         cmpl    $1088, %eax
8         jne     .L14
```

Crear un programa en lenguaje C `fix_set_diag_opt` que use optimizaciones similares a las que hay en el código ensamblador, al estilo del código en la Figura 3.28(b) (mostrada abajo). Usar expresiones que involucren al parámetro *N* mejor que constantes enteras, de manera que el código funcione correctamente si N se redefine.

```
1     /* Calcular prod(i,k) de matrices tam.fijo */
2     int fix_prod_ele_opt(fix_m A,fix_m B, int i, int k) {
3         int *Arow = &A[i][0];
4         int *Bptr = &B[0][k];
5         int result = 0;
6         int j;
7         for (j = 0; j != N; j++) {
8             result += Arow[j] * *Bptr;
9             Bptr += N;
10        }
11        return result;
12    }
```

3.39. Considerar la siguiente declaración de estructura:

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

Esta declaración ilustra que una estructura puede anidarse dentro de otra, igual a como los arrays pueden anidarse dentro de estructuras y dentro de arrays.

El siguiente procedimiento (con algunas expresiones omitidas) opera sobre esta estructura:

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

A. ¿Cuáles son los desplazamientos (en bytes) de los siguientes campos?

```
   p:  _________
 s.x:  _________
 s.y:  _________
next:  _________
```

B. ¿Cuántos bytes en total requiere la estructura?

C. El compilador genera el siguiente código ensamblador para el cuerpo de `sp_init`:

```
   sp en %ebp+8
1      movl    8(%ebp), %eax
2      movl    8(%eax), %edx
3      movl    %edx, 4(%eax)
4      leal    4(%eax), %edx
5      movl    %edx, (%eax)
6      movl    %eax, 12(%eax)
```

En base a esta información, rellenar las expresiones que faltan en el código para `sp_init`.