CC4301 Arquitectura de Computadores
Auxiliar 7

Prof. Aux.: Gaspar Pizarro V.

12 de octubre de 2012

1. P1 Control 2 Año 2005

```
1  .globl Q
2  Q:
3      pushl %ebp
4      movl %esp, %ebp
5      pushl %esi
6      pushl %ebx
7      movl 8(%ebp), %edx # 1er param
8      movl 12(%ebp), %ecx # 2do param
9      movl 16(%ebp), %ebx # 3er param
10     movl (%edx,%ecx,4), %esi
11     movl (%edx,%ebx,4), %eax
12     movl %eax, (%edx,%ecx,4)
13     movl %esi, (%edx,%ebx,4)
14     popl %ebx
15     popl %esi
16     popl %ebp
17     ret
18
19 .globl P
20 P:
21     pushl %ebp
22     movl %esp, %ebp
23     pushl %edi
24     pushl %esi
25     pushl %ebx
26     subl $12, %esp
27     movl 8(%ebp), %edi
28     movl 12(%ebp), %ebx
29     movl 16(%ebp), %esi
30     # si %ebx>=%esi goto .L10
31     cmpl %esi, %ebx
32     jge .L10
33 .L8:
34     # si (%edi,%ebx,4)>=0 goto .L6
35     cmpl $0, (%edi,%ebx,4)
36     jns .L6
37     addl $1, %ebx
38     jmp .L3
39 .L6:
40     subl $4, %esp
41     pushl %esi # 3er. arg
42     pushl %ebx # 2do. arg
43     pushl %edi # 1er. arg
44     call Q
45     subl $1, %esi
46     addl $16, %esp
47 .L3:
48     cmpl %esi, %ebx # si %ebx<%esi goto .L8
49     jl .L8
50 .L10:
51     movl %ebx, %eax # valor de ret.
52     leal -12(%ebp), %esp # %esp= %ebp-12
53     popl %ebx
54     popl %esi
55     popl %edi
56     popl %ebp
57     ret
```

Solución

```
void Q(int *a, int b, int c) {
    int t1 = a[b];
    int t2 = a[c];
    a[b] = t2;
    a[c] = t1;
}
```

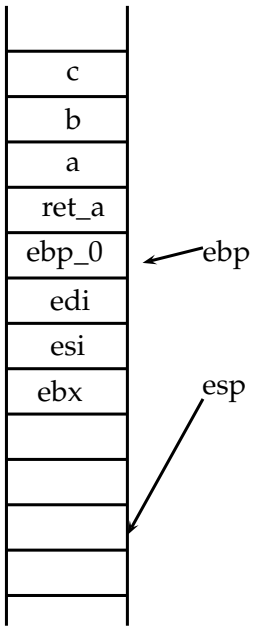
Se ve que en las lineas 11 y 12 que se carga %eax con un valor en memoria, y en la instrucción siguiente que se escribe un valor en memoria con el contenido de %eax. Esto significa que %eax es solo un registro de paso para hacer el swap, entonces puede ser omitido del código C, y así hacer `a[b] = a[c]`.

```
int P(int *a, int b, int c) {
    while (b>=c) {
        if (a[b]>=0) {
            Q(a,b,c);
            c--
        }
        else b++;
    }
    return b;
}
```

En la etiqueta .L8 se tiene que se hace la pega, y en las lineas 48 y 49 se hace una comparación que puede saltar a .L8. Esto sugiere que esto es un ciclo do-while, sin embargo, hay que ver qué se hace antes de entrar a .L8. En las lineas 39 y 31 se ve que se comparan registros y se puede saltar a .L10, que es el retorno. Viendo la condición, se tiene que es la misma que se verifica en las lineas 48 y 49. Entonces el ciclo puede no hacerse nunca, y por lo tanto es un ciclo while.

En la llamada a función de la línea 44 está precedida de un decremento de %esp, y el apilamiento de los argumentos (puestos en orden inverso). Entonces es importante que despues de llamar a la función hay que desapilar los argumentos. Lo que se hace en este caso simplemente es aumentar el %esp en lo que tuvo que bajar antes de ejecutar Q (3 args y el extraño decremento de la linea 40 = 16 bytes.)

En la linea 52 se hace una forma pomposa de decrementar el %esp para compensar el incremento realizado en la línea 26. Considérese el layout antes de la linea 51:



Hay espacios vacíos porque se decrementó %esp en la linea 26. Entonces si se carga %esp con %ebp +12, %esp apuntará al %ebx apilado, que es los mismo que aumentar en 12 el %esp (es solo coincidencia que sean 12 para arriba de %esp o 12 para abajo de %ebp. En otro caso se pudo haber hecho un decremento distinto de %esp y la línea 52 no hubiera cambiado).

2. Recursión

Considérese la siguiente estructura:

```
struct node {
    int value;
    node *left;
    node *right;
}
```

Decompile los siguientes programas en assembler x86 de 32 bits, sabiendo sabiendo que todos los punteros apuntan a `struct node`.

2.1.

```
1  .globl f
2  f:
3      pushl %ebp
4      movl %esp, %ebp
5      pushl %esi
6      pushl %ebx
7      movl 8(%ebp), %ebx
8      movl 12(%ebp), %esi
9      addl %esi, (%ebx)
10     movl 4(%ebx), %eax
11     testl %eax, %eax
12     je .L2
13     pushl %esi
14     pushl %eax
15     call f
16 .L2:
17     movl 8(%ebx), %eax
18     testl %eax, %eax
19     je .L4
20     pushl %esi
21     pushl %eax
22     #movl %esi, 4(%esp)
23     #movl %eax, (%esp)
24     call f
25 .L4:
26     popl %ebx
27     popl %esi
28     movl %ebp, %esp
29     popl %ebp
30     ret
```

Solución

```
void f(struct node *a, int b) {
    a->value = a->value+b;
    if (a->left!=0) f(a->left, b);
    if (a->right!=0) f(a->right, b);
}
```

2.2.

```
31 .globl f
32 f:
33     pushl %ebp
34     movl %esp, %ebp
35     pushl %ebx
36     subl $20, %esp
37     movl 8(%ebp), %eax
38     movl 12(%ebp), %ebx
39     testl %eax, %eax
40     je .L5
41     movl (%eax), %ecx
42     movl (%ebx), %edx
43     cmpl %edx, %ecx
44     jne .L3
45     movl 4(%ebx), %edx
46     movl %edx, 4(%eax)
47     movl 8(%ebx), %edx
48     movl %edx, 8(%eax)
49     jmp .L5
50 .L3:
51     cmpl %edx, %ecx
52     jle .L4
53     movl %ebx, 4(%esp)
54     movl 4(%eax), %eax
55     movl %eax, (%esp)
56     call f
57     jmp .L5
58 .L4:
59     cmpl %edx, %ecx
60     jge .L5
61     movl %ebx, 4(%esp)
62     movl 8(%eax), %eax
63     movl %eax, (%esp)
64     call f
65 .L5:
66     addl $20, %esp
67     popl %ebx
68     popl %ebp
69     ret
```

Solución

```
void f(struct node *a, struct node *b) {
    if (a==0) return;
    if (a->value==b->value) {
        a->left = b->left;
        a->right = b->right;
    }
    else if (a->value>b->value) f(a->left, b);
    else if (a->value<b->value) f(a->right, b);
}
```

El primer condicional corresponde a la lineas 39 y 40. El segundo condicional corresponde a las lineas 45-49, ya que la condición de salto en la linea 44 es que los valores a comparar sean desiguales. El segundo condicional corresponde a las lineas 50-57, y el tercer condicional corresponde a las lineas 58-64. Es interesante (y raro) que en las lineas 59 y 60 se compara a->value con b->value, de la misma forma que pasa si es que no se salta en la linea 52, lo que sugiere que estas lineas podrían no estar.

El código que se muestra es lo que generó el código del ejercicio. Entonces un código sacado a mano puede tener diferencias considerables de estructura. Por ejemplo, en vez de hacer un else-if, tiene sentido hacer ifs y en cada caso poner un return. De hecho, haciéndolo a mano, este es el código que obtuve:

```
void f(struct node *a, struct node *b) {
    if (a==0) return;
    if (a->value==b->value) {
        a->left = b->left;
        a->right = b->right;
    }
    else {
        if (a->value>b->value) {
            f(a->left, b);
            return
        }
        else f(a->right, b);
    }
}
```

El cual es un código que hace lo mismo, solo que de forma mucho mas “bruta”. De todas formas es aceptable.