Jahzeel Miranda Pérez

**Problema de práctica 7.1**

Este problema se refiere a los módulos m.o y swap.o de la figura 7.5. Para cada símbolo que se define o se referencia en swap.o, indique si tendrá una entrada en la tabla de símbolos .symtab en el módulo swap.o. Si es así, indique el módulo que define el símbolo (swap.o o m.o), el tipo de símbolo (local, global o externo) y la sección (.text, .data, .bss o COMMON) a la que se asigna en el módulo.

|  |  |
| --- | --- |
| (a) m.c | (b) swap.c |
| *code/link/m.c*  1 void swap();  2  3 int buf[2] = {1, 2};  4  5 int main()  6 {  7 swap();  8 return 0;  9 }  *code/link/m.c* | *code/link/swap.c*  1 extern int buf[];  2  3 int \*bufp0 = &buf[0];  4 int \*bufp1;  5  6 void swap()  7 {  8 int temp;  9  10 bufp1 = &buf[1];  11 temp = \*bufp0;  12 \*bufp0 = \*bufp1;  13 \*bufp1 = temp;  14 }  *code/link/swap.c* |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Symbol | .symtab entry? | Symbol type | Module where defined | Section |
| buf | Si | extern | m.o | .data |
| bufp0 | Si | global | swap.o | .data |
| bufp1 | Si | global | swap.o | COMMON |
| swap | Si | global | swap.o | .text |
| temp | no |  |  |  |

El objetivo del problema es comprender la relación entre los símbolos del enlazador y las variables y funciones en C, y determinar si cada símbolo definido o referenciado en "swap.o" tendrá una entrada en la tabla de símbolos ".symtab" de "swap.o", y en caso afirmativo, especificar el módulo que define el símbolo, el tipo de símbolo y la sección a la que se asigna.

En primer lugar, se identifican los símbolos definidos en "swap.o", que son "bufp0", "bufp1" y "swap". Se determina que todos estos símbolos son globales, ya que se definen fuera de cualquier función y, por lo tanto, se les asigna una entrada en la tabla de símbolos ".symtab" de "swap.o". Además, se especifica la sección a la que se asigna cada símbolo, que es ".data" para "bufp0", COMMON para "bufp1" y ".text" para "swap".

Luego, se determina que la variable "temp" es una variable local que no se accede desde otros módulos, por lo que no se necesita una entrada de símbolo en la tabla de símbolos ".symtab" de "swap.o".

Finalmente, se identifica un símbolo externo referenciado en "swap.o", que es "buf". Se determina que este símbolo se define en el módulo "m.o" y se asigna a la sección ".data". Como se trata de un símbolo externo, se le asigna una entrada en la tabla de símbolos ".symtab" de "swap.o".

**Problema de práctica 7.2**

En este problema, sea REF(x.i)→DEF(x.k) el hecho de que el enlazador asociará una referencia arbitraria al símbolo x en el módulo i con la definición de x en el módulo k.

Para cada ejemplo que sigue, use esta notación para indicar cómo el enlazador resolvería las referencias al símbolo definido múltiples veces en cada módulo. Si hay un error en tiempo de enlace (regla 1), escriba "error". Si el enlazador elige arbitrariamente una de las definiciones (regla 3), escriba "desconocido".

|  |  |
| --- | --- |
| 1. /\* Módulo 1 \*/   int main()  {  } | /\* Módulo 2\* /  int main;  int p2()  {  } |

(a) REF(main.1)→DEF(\_\_\_\_\_ .\_\_\_\_\_\_\_ )

(b) REF(main.2)→DEF(\_\_\_\_\_ .\_\_\_\_\_\_\_ )

|  |  |
| --- | --- |
| B. / \*Módulo 1 \*/  void main()  {  } | /\* Módulo 2 \*/  int main = 1;  int p2()  {  } |

(a) REF(main.1)→DEF(\_\_\_\_\_ .\_\_\_\_\_\_\_ )

(b) REF(main.2)→DEF(\_\_\_\_\_ .\_\_\_\_\_\_\_ )

|  |  |
| --- | --- |
| C. /\* Módulo 1 \*/  int x;  void main()  {  } | / \*Módulo 2 \*/  double x = 1.0;  int p2()  {  } |

(a) REF(x.1)→DEF(\_\_\_\_\_ .\_\_\_\_\_\_\_ )

(b) REF(x.2)→DEF(\_\_\_\_\_ .\_\_\_\_\_\_\_ )

Respuestas:

En el A, hay dos módulos que definen una función main, uno en el Módulo 1 y otro en el Módulo 2. Según la regla 2, el enlazador elegirá el símbolo fuerte definido en el Módulo 1 sobre el símbolo débil definido en el Módulo 2. Por lo tanto, la respuesta es:

(a) REF(main.1)→DEF(main.1)

(b) REF(main.2)→DEF(main.1)

En el B, tanto el Módulo 1 como el Módulo 2 definen un símbolo fuerte main. Según la regla 1, esto es un error en tiempo de enlace. Por lo tanto, la respuesta es que hay un error.

En el C, hay dos módulos que definen una variable x, una como entero en el Módulo 1 y otra como doble en el Módulo 2. Según la regla 2, el enlazador elegirá el símbolo fuerte definido en el Módulo 2 sobre el símbolo débil definido en el Módulo 1. Por lo tanto, la respuesta es:

(a) REF(x.1)→DEF(x.2)

(b) REF(x.2)→DEF(x.2)

**Práctica del problema 7.3**

Sea "a" y "b" módulos de objeto o bibliotecas estáticas en el directorio actual, y sea "a→b" que "a" depende de "b", en el sentido de que "b" define un símbolo que se referencia en "a". Para cada uno de los siguientes escenarios, muestre la línea de comando mínima (es decir, una con la menor cantidad de argumentos de archivo de objeto y biblioteca) que permitirá al enlazador estático resolver todas las referencias de símbolos.

A. p.o→libx.a

B. p.o→libx.a→liby.a

C. p.o→libx.a→liby.a y liby.a→libx.a→p.o

Respuesta:

En el escenario A, solo hay un archivo objeto "p.o" que depende de la biblioteca estática "libx.a". Por lo que la línea de comando mínima es "gcc p.o libx.a". Esto significa que "p.o" se enlaza con "libx.a" para resolver todas las referencias de símbolos.

1. linux> gcc p.o libx.a

En el escenario B, hay un archivo objeto "p.o" que depende de la biblioteca estática "libx.a", que a su vez depende de la biblioteca estática "liby.a". Por lo que la línea de comando mínima es "gcc p.o libx.a liby.a". Esto significa que "p.o" se enlaza con "libx.a" y "liby.a" para resolver todas las referencias de símbolos.

1. linux> gcc p.o libx.a liby.a

En el escenario C, hay un archivo objeto "p.o" que depende de la biblioteca estática "libx.a", que a su vez depende de la biblioteca estática "liby.a", que a su vez depende de "libx.a" y "p.o". Por lo que la línea de comando mínima es "gcc p.o libx.a liby.a libx.a". Esto significa que "p.o" se enlaza con "libx.a" y "liby.a", y luego "libx.a" se incluye nuevamente para resolver cualquier referencia de símbolo en "p.o" que se define en "libx.a".

1. linux> gcc p.o libx.a liby.a libx.a

**Práctica del problema 5.4**

Cuando usamos gcc para compilar combine3 con la opción de línea de comando -O2, obtenemos un código con un rendimiento de CPE sustancialmente mejor que con -O1:

Entero Punto flotante

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Función | Página | Método | + | \* | + | \* |
| combine3 | 549 | Compiled-O1 | 7.17 | 9.02 | 9.02 | 11.03 |
| combine3 | 549 | Compiled-O2 | 1.60 | 3.01 | 3.01 | 5.01 |
| combine4 | 551 | Accumulate in temporary | 1.27 | 3.01 | 3.01 | 5.01 |

Logramos un rendimiento comparable al de combine4, excepto para el caso de la suma de enteros, pero incluso esta mejora significativamente. Al examinar el código de ensamblado generado por el compilador, encontramos una variante interesante para el bucle interno:

Bucle interno de combine3. data\_t = double, OP = \*. Compilado -O2

dest en %rbx, data+i en %rdx, data+length en %rax

Producto acumulado en %xmm0

1 .L22: loop:

2 vmulsd (%rdx), %xmm0, %xmm0 Multiplicar producto por data[i]

3 addq $8, %rdx Incremento data+i

4 cmpq %rax, %rdx Comparar con data+length

5 vmovsd %xmm0, (%rbx) Almacenar el producto en dest

6 jne .L22 Si es !=, ir al bucle

Podemos comparar esto con la versión creada con nivel de optimización 1:

Bucle interno de combine3. data\_t = double, OP = \*. Compilado -O1

dest en %rbx, data+i en %rdx, data+length en %rax

1 .L17: loop:

2 vmovsd (%rbx), %xmm0 Leer producto de dest

3 vmulsd (%rdx), %xmm0, %xmm0 Multiplicar producto por data[i]

4 vmovsd %xmm0, (%rbx) Almacenar el producto en dest 5 addq $8, %rdx Incremento data+i

6 cmpq %rax, %rdx Comparar con data+length

7 jne .L17 Si es !=, ir al bucle

Vemos que, además de algunas instrucciones reordenadas, la única diferencia es que la versión más optimizada no contiene la vmovsd que implementa la lectura desde la ubicación designada por dest (línea 2).

A. ¿Cómo difiere el papel del registro %xmm0 en estos dos bucles?

B. ¿La versión más optimizada implementará fielmente el código C de combine3, incluso cuando haya alias de memoria entre dest y el vector de datos?

C. Explique por qué esta optimización preserva el comportamiento deseado, o dé un ejemplo donde produciría resultados diferentes al código menos optimizado.

Respuesta:

A: En la versión de código menos optimizada (-O1), el registro %xmm0 se utiliza solo como un valor temporal que se establece y utiliza en cada iteración del bucle. Sin embargo, en la versión más optimizada (-O2), se utiliza como una variable acumuladora, similar a como se utiliza la variable "acc" en la función combine4. La diferencia es que en cada iteración del bucle, se actualiza la ubicación de memoria "dest" mediante la segunda instrucción "vmovsd" en la versión optimizada.

B: La respuesta es sí, la versión más optimizada implementará fielmente el código C de combine3 incluso si hay alias de memoria entre "dest" y el vector de datos.

C: La optimización se puede realizar sin cambiar el comportamiento del programa, ya que el valor leído desde "dest" al comienzo de cada iteración será el mismo valor que se escribió en el registro %xmm0 al final de la iteración anterior. Por lo tanto, la instrucción que combina los valores puede utilizar el valor ya almacenado en %xmm0. Esto preserva el comportamiento deseado del programa y no produce resultados diferentes a la versión menos optimizada.

**Problema de práctica 5.5**

Supongamos que deseamos escribir una función para evaluar un polinomio, donde un polinomio de grado n se define como un conjunto de coeficientes a0, a1, a2, ..., an. Para un valor x, evaluamos el polinomio calculando

a0 + a1x + a2x^2 + ... + anx^n (5.2)

Esta evaluación se puede implementar mediante la siguiente función, que tiene como argumentos una matriz de coeficientes a, un valor x y el grado del polinomio (el valor n en la ecuación 5.2). En esta función, calculamos tanto los términos sucesivos de la ecuación como las sucesivas potencias de x dentro de un solo bucle:

1 double poly(double a[], double x, long degree)

2 {

3 long i;

4 double result = a[0];

5 double xpwr = x; /\* Equals x^i at start of loop \*/

6 for (i = 1; i <= degree; i++) {

7 result += a[i] \* xpwr;

8 xpwr = x \* xpwr;

9 }

10 return result;

11 }

A. ¿Cuántas adiciones y cuántas multiplicaciones realiza este código para un grado n?

B. En nuestra máquina de referencia, con las operaciones aritméticas que tienen las latencias mostradas en la Figura 5.12, medimos el CPE para esta función en 5.00. Explique cómo surge este CPE basado en las dependencias de datos formadas entre iteraciones debido a las operaciones que implementan las líneas 7-8 de la función.

Respuesta:

A. La función poly realiza 2n multiplicaciones y n adiciones para evaluar un polinomio de grado n. Cada término del polinomio requiere una multiplicación por la potencia correspondiente de x, así que se realizan n multiplicaciones para calcular todas las potencias de x y otras n multiplicaciones para multiplicar cada coeficiente por su respectiva potencia de x. Además, se realizan n adiciones para sumar todos los términos del polinomio.

B. El cálculo que limita el rendimiento de la función poly es la expresión repetida xpwr = x \* xpwr, que implica una multiplicación en punto flotante con una latencia de 5 ciclos de reloj. Cada iteración del bucle necesita el resultado de la multiplicación anterior (xpwr) para calcular la siguiente multiplicación, lo que crea una dependencia de datos entre iteraciones. Por lo tanto, el cálculo para una iteración no puede comenzar hasta que se complete el de la iteración anterior, lo que impone una latencia mínima de 5 ciclos de reloj para cada iteración. Además, la actualización de result solo requiere una suma en punto flotante con una latencia de 3 ciclos de reloj entre iteraciones sucesivas. En consecuencia, el tiempo de ejecución promedio de la función poly está limitado por el tiempo de espera de las multiplicaciones en punto flotante, y el CPE medido para la función es de 5.00, lo que indica que la función tarda en promedio 5 ciclos de reloj por elemento.