GUÍA TEÓRICA PARA EL DESARROLLO DEL LABORATORIO DE ORGANIZACIÓN Y ARQUITECTURA DE COMPUTADORAS

SESIÓN N°:	2

TEMA:	PROGRAMACIÓN EN ENSAMBLADOR DE 64 BITS Y RENDIMIENTO

INDICACIONES GENERALES

- Revisar la presente guía teórica antes del desarrollo del laboratorio.
- Complementar el estudio de la guía con el material del curso y la bibliografía indicada para un mejor desempeño.
- Realizar consultas puntuales sobre el contenido de la guía al profesor del curso.
- Revisar y desarrollar los ejemplos por su cuenta.
- Comparar su solución con la expuesta en la presente guía.

OBJETIVOS

- Codificar programas de baja complejidad utilizando ensamblador de 64 *bits*.
- Entender la utilidad del *stack* para el uso de *calling conventions*.
- Estudiar dos aspectos de rendimiento de una computadora: optimización y análisis de rendimiento.
- Medir el tiempo de ejecución de un programa simple.
- Optimizar un programa simple y analizar su mejora.

ÍNDICE

PROGRAMACIÓN EN ENSAMBLADOR DE 64 BITS

La programación en lenguaje ensamblador de 64 bits sigue el formato convencional de 32 bits, con la gran diferencia del uso de registros extendidos así como del uso de más direcciones de memoria. Una de las razones fundamentales para aprender a codificar en 64 bits es su adaptabilidad que tiene para interactuar con programas codificados en C o C++ [1]. Esta interacción se da por el uso de los *calling conventions*.

CALLING CONVENTIONS

Los *calling conventions* son un conjunto de reglas que simplifica la definición y el uso de subrutinas. Por ejemplo, dado un conjunto de reglas del *calling conventions*, un programador no necesita examinar la definición de una subrutina para determinar cómo se deben pasar los parámetros a esa subrutina. Además, es posible hacer que los compiladores de lenguaje de alto nivel sigan dichas reglas y, de esta forma, permitir que las rutinas de lenguaje ensamblador y las rutinas de lenguaje de alto nivel se llamen entre sí [2]. Si fuese el caso en el cual la función presenta más de seis parámetros enteros y/o más de ocho parámetros flotantes (Tabla 1), se pasan al *stack* en orden inverso. Es decir, la función myFun(uno, dos, tres, cuatro, cinco, seis, siete, ocho, nueve) implicaría un orden de insercción de nueve, ocho y luego siete, solo si consideramos parámetros enteros. El mismo principio aplicaría para los parámetros flotantes.

Tabla 1: *Calling conventions* según su tipo de dato

Posición según su tipo	Tipo de dato	
	Entero	Flotante
Primero	rdi	xmm0
Segundo	rsi	xmm1
Tercero	rdx	xmm2
Cuarto	rcx	xmm3
Quinto	r8	xmm4
Sexto	r9	xmm5
Séptimo	-	xmm6
Octavo	-	xmm7

En la Lista 1 se muestra un ejemplo representativo sobre la codificación en lenguaje ensamblador y la declaración desde C.

Lista 1: Ejemplo de *calling conventions*

```
1 extern void myfunction(char a, short b, float c, double *d,
      double e)
2 //a en rdi, b en rsi, c en xmm0, d en rdx, e en xmm1
```

Ejemplo de *calling conventions*

A continuación, en la Lista 2, se muestra un programa principal que hace uso del *calling conventions* para llamar dos funciones que calculan la norma a dos de un vector de

valores en punto flotante simple. Se detallan las dos funciones en lenguaje ensamblador y en lenguaje en C en la Lista 3 y la Lista 4, respectivamente.
Para un vector $\vec{x} = (x_1, x_2, x_3, ..., x_n)$ se define la norma-2 en la ecuación 1.

$$\|\vec{x}\|_{norma-2} = \sqrt{x_1^2 + x_2^2 + x_3^2 + ... + x_n^2} \quad (1)$$

Lista 2: Programa principal con uso de *calling conventions*

```
1  ;Nombre del program: float_norm_two
2  ;Para compilar ejecutar el siguiente comando:
3  ;gcc asmFloatNormTwo.o float_norm_two.c -o float_norm_two -lm
4
5  #include <stdio.h>
6  #include <stdlib.h>
7  #include <math.h>
8
9  extern void asmFloatNormTwo(float *v1, int N, float *n2);
10 void cFloatNormTwo(float *v1, int N, float *n2);
11
12 int main(){
13     float *v1, n2C, n2Asm;
14     int N = 1024;
15     v1 = malloc(N* sizeof(float));
16     int i = 0;
17     for (i = 0; i < N; i++){
18         v1[i] = (float)i;
19     }
20     asmFloatNormTwo(v1, N, &n2Asm);
21     cFloatNormTwo(v1, N, &n2C);
22     printf("%f \n %f \n",n2Asm,n2C);
23     return 0;
24 }
```

Lista 3: Cálculo de la norma dos de un vector en ensamblador

```
1  ;Nombre del program: asmFloatNormTwo
2  ;Para ensamblar ejecutar los siguientes comandos:
3  ;nasm -f elf64 -o asmFloatNormTwo.o asmFloatNormTwo.asm
4  ;ld -m elf64 asmFloatNormTwo.o -o asmFloatNormTwo
5  ;./asmFloatNormTwo
6
7  global asmFloatNormTwo
8  section .text
9
10 asmFloatNormTwo:
11     xorpd xmm0, xmm0    ;xmm0 <- 0
12     xorpd xmm1, xmm1    ;xmm1 <- 0
13     cmp rdx, 0          ;si N == 0
14     je done             ;terminar el programa
15
```

```
16 next:
17     movss xmm0, [rdi]   ;xmm0 <- [rdi]
18     mulss xmm0, xmm0    ;xmm0 <- xmm0 * xmm0
19     addss xmm1, xmm0    ;xmm1 <- xmm1 + xmm0
20     add rdi, 4          ;siguiente elemento
21     sub rsi, 1          ;disminuir contador
22     jnz next
23
24 done:
25     sqrtss xmm1, xmm1
26     movss [rdx], xmm1
27     ret
```

Lista 4: Cálculo de la norma dos de un vector en C

```
1 void cFloatNormTwo(float *v1, int N, float *n2){
2     int i = 0;
3     float sum = 0;
4     for (i = 0; i < N; i++){
5         sum += v1[i] * v1[i];
6     }
7     n2[0] = sqrt1(sum);
8 }
```

OPTIMIZACIÓN Y ANÁLISIS DE RENDIMIENTO

El rendimiento del CPU es el foco principal de los esfuerzos de optimización del sistema; sin embargo, no existe una única forma de mejorar el rendimiento pues este se ve afectado por diversos factores. Las técnicas de optimización potencial de CPU incluyen unidades de coma flotante integradas, unidades de ejecución en paralelo, instrucciones especializadas, canalización de instrucciones, predicción de ramificaciones y optimización de código [3]. Algunas de estas técnicas se describen a continuación:

Loop unrolling (expansión de bucle)

Los bucles son utilizados ampliamente en los programas y son excelentes candidatos para la optimización. El *loop unrolling* consiste en extender un bucle de modo que cada nueva iteración contenga varias de las iteraciones originales como se observa en la Lista 5.

Lista 5: Técnica de *loop unrolling*

```
25 for (i = 1; i <= 30; i++)
26     a[i] = a[i] + b[i] * c;
27
28 ;Aplicando dos veces la tecnica de loop unrolling obtenemos:
29 for (i = 1; i <= 30; i += 3){
30     a[i] = a[i] + b[i] * c;
31     a[i+1] = a[i+1] + b[i+1] * c;
32     a[i+2] = a[i+2] + b[i+2] * c;
33 }
```

A simple vista parece ser una mala forma de programar; no obstante, esta técnica reduce la sobrecarga del bucle y permite que las operaciones de diferentes iteraciones del bucle se ejecuten en paralelo. Adicionalmente, permite una mejor programación de instrucciones debido a una menor dependencia de datos y un mejor uso del registro. Claramente se observa que la cantidad de código aumenta, por lo que esta no es una técnica que deba emplearse para cada ciclo del programa. Se recomienda utilizarlo en secciones de código que representan una parte significativa del tiempo de ejecución.

Loop fusion (Fusión de bucle)

Otra técnica útil de optimización de bucle es el *loop fusion* la cual combina bucles que usan los mismos elementos de datos, como se observa en la Lista 6. Esto da como resultado un mayor rendimiento de caché, un mayor paralelismo de nivel de instrucción y una sobrecarga de bucle reducida.

Lista 6: Técnica de loop fusion

```
25 for (i = 1; i < N; i++)
26     C[i] = A[i] + B[i];
27
28 for (i = 1; i < N; i++)
29     D[i] = E[i] + C[i];
30
31 ;Aplicando la tecnica de loop fusion obtenemos:
32 for (i = 0; i < N; i++){
33     C[i] = A[i] + B[i];
34     D[i] = E[i] + C[i];
35 }
```

Look-up tables (Tablas de búsqueda)

El uso de *look-up tables* representa una mejora de un código pues es una matriz inicializada que contiene información precalculada. Por lo general, se utilizan para evitar realizar cálculos complejos que ralentizan el código. El ejemplo de la Lista 7 presenta el cálculo del factorial de un número.

Lista 7: Técnica de look-up tables

```
1  long factorial(int i){
2      if (i == 0)
3          return 1;
4      else
5          return (i * factorial(i - 1));
6  }
7
8  ;Aplicando look-up tables
9  static long factorial_table[] = {1, 1, 2, 6, 24, 120, 720, 5040,
       40320, 362880, 3628800, 39916800, 479001600, 6227020800};
10 long factorial(int i){
11     return factorial_table[i];
12 }
```

Consejos de optimización de un programa

- Usar constantes y variables locales siempre que sea posible.
- Usar matrices en lugar de punteros cuando pueda.
- Minimizar las conversiones de punto flotante a entero.
- Utilizar el tipo de dato adecuado (*float*, *double*, *int*).
- Considerar usar la multiplicación en lugar de la división.
- Eliminar todas las ramas innecesarias.
- Crear declaraciones condicionales (*if*, *switch*, *case*) con los casos más probables primero.
- No descartar un algoritmo basado en su rendimiento original, una comparación justa solo ocurre cuando todos los algoritmos están complemente optimizados.

El análisis del rendimiento de la computadora es una ciencia cuantitativa. Por ello, se deben utilizar herramientas matemáticas y estadísticas dado que brindan muchas formas de evaluar el rendimiento general de un sistema. Entonces, ¿cuánto tiempo tarda el sistema en ejecutar un programa? De hecho, hay tantas formas de cuantificar el rendimiento del sistema, que seleccionar la estadística correcta se convierte en un desafío. A continuación analizaremos un método de medición y de comparación de rendimiento.

Tiempo de ejecución de CPU

Es la medida de rendimiento más básica y se define como el tiempo que toma un programa en ejecutarse en el procesador analizado. En C existe más de una forma de medir el tiempo. Una de ellas es través de la función *clock()* de la biblioteca *time.h*. Esta función permite contar el número de ciclos de CPU que ha transcurrido hasta el punto donde se invoca. En primer lugar, se debe llamar a la biblioteca *time.h*. Luego, se debe definir las variables de tiempo. Después, se almacena en alguna de las variables declaradas el tiempo de ejecución mediante la función *clock()* de forma que delimiten el código del cual se requiere conocer su tiempo de ejecución. Finalmente, se calcula el tiempo de ejecución en microsegundos (us) por medio de la constante *CLOCKS_PER_SEC*. La Lista 8 muestra los pasos descritos.

Lista 8: Medida del tiempo de ejecución de un programa en C

```
4  #include <time.h>
5  clock_t start_t, end_t, total_t;
6  start_t = clock();
7  // codigo principal
8  end_t = clock();
9  total_t = end_t - start_t;
10 printf("%.0f us\n",1000000*((float)total_t)/CLOCKS_PER_SEC);
```

Speedup (factor de mejora)

El *speedup* que se puede obtener al mejorar una parte de una computadora se puede calcular mediante el uso de una función particular. Dicho factor es utilizado para medir la mejora de un sistema respecto a una versión inicial o básica del mismo [4]. Se define en la ecuación 2.

$$S_p = \frac{T_b}{T_m} \tag{2}$$

Donde:
S_p: Speedup
T_b: Tiempo de ejecución del sistema sin mejora
T_m: Tiempo de ejecución del sistema mejorado

Ejemplo de rendimiento

Escribir un programa en C que permita el cálculo de la traza de una matriz cuadrada $\mathbf{M}$ de n x n elementos e indique su diagonal secundaria. A partir de ello, realizar lo siguiente:

- Calcular el tiempo de ejecución de CPU
- Realizar una mejora al código realizado
- Determinar el *speedup*

Se analizará la solución para el caso cuando $n = 4096$. El programa fue ejecutado y comprobado en un computador Intel Core i5-3337U CPU con frecuencia 1.80GHz con memoria RAM de 4GB y sistema operativo Ubuntu 14.04. Para conocer los datos en Ubuntu de su computador puede escribir los comandos **cat /proc/cpuinfo** y **cat /proc/meminfo** en su terminal. La función dentro del programa que realiza las operaciones requeridas se muestra en la Lista 9.

Lista 9: Función que determina la traza y la diagonal secundaria de una matriz cuadrada

```
25 float trace(float *dsec, float *arr, int numel) {
26         int i, j, m = 0;
27         for(i = 0; i < numel; ++i){
28                 m += arr[i * numel + i];
29         }
30         for(j = 0; j < numel; ++j){
31                 dsec[j] = arr[(j + 1) * numel - j];
32         }
33         return m;
34 }
```

Luego, para compilar el código, se utiliza el compilador **gcc**, el cual viene por defecto en la instalación de la mayoría de distribuciones LINUX/UNIX. Para ello, se requiere abrir un terminal en la ubicación del código del programa (se le denominó **m_trace.c**) y ejecutar la siguiente instrucción en el terminal:

```
$ gcc -o calc_trace m_trace.c -lm
```

En este caso, se ha hecho uso de la bandera de compilación -lm la cual invoca funciones matemáticas y cuyo ejecutable se llama **calc_trace**. Finalmente, para efectuar el programa, se debe invocar al ejecutable en el terminal de la siguiente forma:

```
$ ./calc_trace
```

El programa **calc_trace** presenta un tiempo de ejecución de T_b =337 us. Adicionalmente, de las mejoras expuestas, las técnicas de expansión y fusión de bucle son aplicables al ejemplo. Las dos mejoras aplicadas se observan en la Listas 10 y la Lista 11.

Lista 10: Primera mejora mediante el uso de *loop fusion*

```
42 float trace_m1(float *dsec, float *arr, int numel){
43     int i, m = 0;
44         for(i = 0; i < numel; ++i){
45                 m += arr[i * numel + i];
46                 dsec[i] = arr[(i + 1) * numel - i];
47         }
48         return m;
49 }
```

Lista 11: Segunda mejora mediante el uso *loop unrolling* y *loop fusion*

```
56 float trace_m2(float *dsec, float *arr, int numel){
57         int i, m = 0;
58         for (i = 0; i < numel; i += 2){
59                 m += arr[i * numel + i];
60                 dsec[i] = arr[(i + 1) * numel - i];
61                 m += arr[(i + 1) * numel + (i + 1)];
62                 dsec[i + 1] = arr[(i + 2) * numel - i - 1];
63         }
64         return m;
65 }
```

El tiempo de ejecución obtenido para la primera mejora es $T_{m1} = 241\ us$ y para la segunda mejora es $T_{m2} = 235\ us$. Ello indica que la primera técnica ha permitido una mejora sustancial; sin embargo, la segunda no ha incrementado el tiempo de ejecución significativamente. Finalmente, los *speedups* que corroboran la observación hecha con los tiempos de ejecución obtenidos son los siguientes:

$$S_{p1} = \frac{T_b}{T_{m1}} = \frac{337}{241} = 1{,}39$$

$$S_{p2} = \frac{T_b}{T_{m2}} = \frac{337}{235} = 1{,}42$$

Referencias

[1] Chris Lomont. Introduction to x64 assembly. `https://software.intel.com/en-us/articles/introduction-to-x64-assembly`, March 2012. Accessed on 2019-09-14.

[2] Ed Jorgensen. *x86-64 Assembly Language Programming with Ubuntu*. 2019.

[3] Linda Null and Julia Lobur. *The essentials of computer organization and architecture*. Jones & Bartlett Publishers, 2014.

[4] John L Hennessy and David A Patterson. *Computer architecture: a quantitative approach*. Elsevier, 2011.