Arquitecturas de las Computadoras
TP3 - Assembler y C

Introducción

Todo programa que corra en una computadora, tiene que estar en un lenguaje que ésta entienda; en código de máquina. El código de máquina hace uso de los registros del procesador, del stack, de las interrupciones, etc. Así que un programa escrito en C, en Assembler, COBOL, Pascal, Ada, etc; a fin de cuentas termina en código de máquina, lo que implica es que siempre se puede hacer una interface para conectar los lenguajes entre sí. En particular, cómo entre código de máquina y Assembler hay una relación de biyectiva, es fácil conectar este lenguaje con cualquier otro, ya que con Assembler hay que respetar el funcionamiento interno del otro lenguaje para poder interactuar.

Es interesante poder hacer una interface con Assembler ya que éste tiene control directo sobre las operaciones del procesador. No como los lenguajes de propósito general que se abstraen de la arquitectura. Con este lenguaje se pueden deshabilitar las interrupciones, habilitar modo protegido, cambiar la dirección del directorio de páginas, modificar el stack pointer, etc.

El lenguaje C es por excelencia el lenguaje de elección para el desarrollo básico de un sistema operativo. Por su bajo nivel, versatilidad, manejo de memoria y su independencia de servicios que necesitan otros lenguajes. Otros lenguajes de programación de más alto nivel, por ejemplo C++, necesitan configuraciones especiales para funcionar.

La convención de cómo realizar las llamadas entre distintos lenguajes, cómo se reserva espacio para las variables, cómo se alinean los datos, etc; es parte de la ***Application Binary Interface (ABI)***

Llamadas de Assembler a C

Este camino es el más fácil de entender. Si recordamos que al final de cuentas, una función en C es una dirección de memoria, no debería ser difícil hacer una llamada desde ASM a C. Vamos a utilizar el programa ***nasm*** para compilar los archivos en asm, ***gcc*** para compilar los archivos en C y también para linkeditarlos entre sí y contra la biblioteca estándar de C.

Ejemplo 1

Escribir una función en C que imprima por pantalla “Hola Mundo”. Luego, escribir una función en ASM que llame a dicha función

```
; main.asm
GLOBAL main
EXTERN hello_world

section .text

main:
        call hello_world

        mov eax, 1        ; systemcall exit()
        mov ebx, 0        ; parametro para exit()
        int 80h
```

```
//hello.c
#include <stdio.h>

void hello_world();

void hello_world() {
         printf (“Hello World!\n”);
}
```

Para compilar y correr los archivos anteriores correr las siguientes líneas:

```
$> nasm -f elf32 main.asm
$> gcc -c -m32 hello.c
$> gcc -m32 main.o hello.o -o hello
$> ./hello
```

La directiva **-f elf32** le indica a *nasm* que genere un archivo objeto con formato *elf (executable linux format)* para una arquitectura de 32 bits. Algo equivalente hace la directiva ***-m32.*** Si no se especifica esto, el formato default es el de la máquina actual.

Note que ahora, en lugar de utilizar la etiqueta *_start* ahora se utiliza la etiqueta *main.* Esto es porque la biblioteca de C ya tiene una etiqueta de entrada, que configura el sistema para luego llamar a nuestro main.

Pregunta: Ya que ahora estamos en un entorno de C, ¿De qué otra forma se podría haber terminado el programa?

Ejemplo 2

En este ejemplo, vamos a pasar un valor a su representación en ASCII numérica.
Recordar cómo es el pasaje de pasaje de parámetros en C, por Stack en 32 bits:
Si una función tiene la siguiente forma:

```
int fnc(arg1, arg2, arg3);
```

Entonces, los argumentos se pasan de derecha a izquierda, de tal forma, que el primer argumento, sea el primero en el stack. Es decir, primero **arg3**, luego **arg2** y por último **arg1**. Luego, el resultado de dicha función se devuelve por el registro eax.

La función de C *sprintf* escribe un string con formato en un stream, tiene el siguiente prototipo:

```
int sprintf(char *str, const char *fmt, …);
```

Esto quiere decir que para llamar a esta función, primero será necesario pushear al stack los argumentos variables, luego la cadena de formato (que le indicará a la función cómo levantar la información) y por último el destino. Luego, se llama a la función *puts* que imprime por pantalla una cadena.

```
int puts(const char *str);
```

Código:

```
;main.asm
GLOBAL main
EXTERN puts, sprintf

section .rodata
fmt db “%d”, 0
number dd 123456790

section .text
main:
      push dword [number]
      push fmt
      push buffer

      call sprintf

      add esp, 3*4

      push buffer
      call puts
```

```
      add esp, 4
      ret

section .bss
buffer resb 40
```

Para compilar este código, simplemente hay que correr las siguientes líneas:

```
$> nasm -f elf32 main.asm
$> gcc -m32 main.o -o main
$> ./main
```

Preguntas:

- ¿Porqué se le suma al stack, luego de llamar a la función sprintf, 12 bytes? ¿Porque se hace lo mismo luego de llamar a puts?
- ¿Cual es el valor de retorno de la función *main?*

Registros a preservar entre llamadas

Entre llamadas de C, hay ciertos registros que las funciones deben preservar para no afectar el funcionamiento de otras funciones. Las funciones que llaman deben tener en cuenta que registros pueden cambiar, de tal forma de hacer un backup antes de llamarlas.

Los registros a preservar entre las llamadas de las funciones son:

- ebx
- esi
- edi
- ebp
- esp

Es decir, cuando una función termina, debe dejar exactamente los registros anteriores como los recibió.

Stack Frame

Es parte de la ABI de C, cuando se entra a una función, armar esta estructura. Su función es preservar los registros de stack y proveer un sistema estándar de acceso a los parámetros de dicha función.

Con este sistema, la el stack queda de la siguiente forma:

EBP	EBP del contexto anterior
EBP + 4	Dirección de Retorno
EBP + 8	Primer argumento
EBP + 12	Segundo argumento
EBP + (n+1)*4	n-esimo argumento

En C *main* es una función que tiene el siguiente prototipo

```
int main(int argc, char *argv[]);
```

El *program loader*, además de configurar algunas cosas, pasa como parámetro de dicha función los argumentos del programa. Entonces se pueden acceder de la siguiente forma:

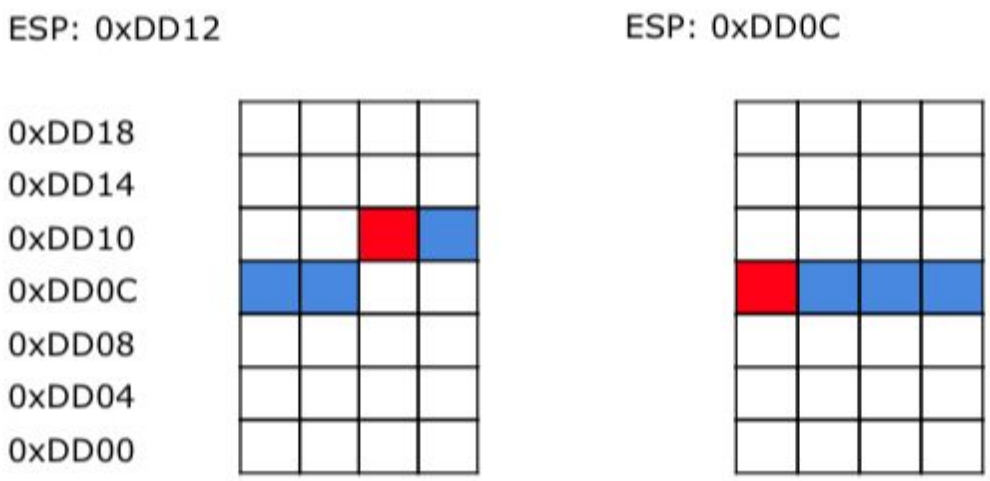
```
;main.asm
GLOBAL main
EXTERN printf

section .rodata
fmt db “Cantidad de argumentos: %d\n”, 0

section .text
main:
        push ebp      ; Armado de stack frame
        mov ebp, esp  ;

        push dword [ebp+8]
        push fmt
        call printf
        add esp, 2*4

        mov eax, 0

        mov esp, ebp  ; Desarmado de stack frame
        pop ebp
        ret
```

Ejercicio 1

Modificar el ejemplo anterior, para que además de imprimir la cantidad de argumentos, imprima cada uno de los argumentos.

Alineamiamiento a palabra

El procesador cada vez que accede a memoria, lo hace leyendo y escribiendo siempre 4 bytes (el tamaño de la palabra en una arquitectura x86).



Cuando el procesador accede a un dato des-alineado, debe hacer dos lecturas, una para pedir la primer porción de los datos, y otra para la segunda porción; es decir, dos accesos. Para la escritura requiere más operaciones, ya que debe leer de memoria para salvar el dato que no va a sobreescribir, luego combinarlo con el dato que si va a escribir para la primera porción, y luego recuperar el dato de la segunda porción para escribir la segunda porción.

Una forma muy fácil de arreglar este problema, es negar los últimos 4 bits. Cómo el stack crece hacia menores valores de esp, seguro esa posición va a estar por detrás del puntero actual. De esta forma se puede hacer un acceso a memoria más eficiente.

```
GLOBAL main

ALIGN 4

main:
        push ebp
        mov ebp, esp

        ; declaración de variables

        and esp, -16

        ; … programa

        mov esp, ebp
        pop ebp
        ret
```

Responder: ¿Cual es la representación de -16 en hexadecimal?

Ejercicio 2

Escriba un programa en C, que realice lo mismo que el ejercicio anterior (imprimir por salida estándar la cantidad de argumentos del programa),

```
// arguments.c
#include <stdio.h>

int main(int argc, char *argv[]) {
     printf(“Cantidad de argumentos %d\n”, argc);
     return 0;
}
```

Compilarlo con el siguiente comando (que elimina un montón de elementos de control de gcc):

```
$> gcc -c arguments.c -m32 -fno-dwarf2-cfi-asm -fno-exceptions -S
-fno-asynchronous-unwind-tables -masm=intel
```

Compare la salida del archivo *arguments.s* con el ejemplo provisto en ASM. Más allá de los elementos de control, ¿Que diferencia encuentra? ¿Qué similitudes?

Para pensar: ¿Se imagina porque no se deben comparar las estructuras por valor, sinó por miembro por miembro? ¿Porqué habría *basura* entre sus componentes?

Llamadas de C a Assembler

Este es el camino que tiene una aplicación más práctica. ¿Qué punto tiene programar en Assembler cuando se tiene un lenguaje eficiente, práctico y liviano? Salvo que exista una razón muy puntual; cómo que no exista un compilador; evitaremos programar en Assembler.

Nuevamente, las funciones a fin de cuentas son direcciones, así que es posible llamar funciones desde C hacia Assembler.

Ejemplo

Si no contamos con una biblioteca estándar que acceda a la API del SO para enviar o recibir información, debería ser implementado por nosotros. Ya vimos que hacer esto completamente en Assembler, ahora lo tendremos que hacer nosotros.

El PID (Process ID) es un número que utiliza Linux para identificar los procesos que están corriendo. Si se corre el comando **ps x**, se puede ver una lista de los procesos que están corriendo actualmente que son del usuario. Obviamente, como **ps** es un proceso, también aparece en la lista.

No hay ninguna función estándar que informe cual es el **pid** de un programa. Cómo es información que maneja el OS, habrá que pedírsela a él.

```
//main.c
#include<stdio.h>

unsigned int pid();

int main(int argc, char *argv[]) {
      int mpid = pid();
      printf("Process Id: %d\n", mpid);
      return 0;
}
```

```
; libasm.asm
GLOBAL pid

pid:
       push ebp
       mov ebp, esp

       mov eax, 0x14 ; syscall getpid
       int 0x80
       ; el resultado ya está en eax

       mov esp, ebp
       pop ebp
       ret
```

Compilar y correr el código de ejemplo con las siguientes líneas:

```
$> nasm -f elf32 libasm.asm
$> gcc -c -m32 main.c
$> gcc -m32 main.o libasm.o -o pid
$> ./pid
```

Ejecútelo varias veces para comprobar que el **pid** va en aumento.

Compile el archivo *main.c* de tal forma de obtener su salida en Assembler, compruebe que efectivamente se hace una llamada a la función **pid()** mediante la línea.

```
     call pid
```

Para que C sepa de qué forma, con que argumentos y con qué tipo de retorno se va a llamar a una función, es necesario especificar su prototipo, por eso se escribió antes de main la línea:

```
unsigned int pid();
```

Esto le indica al compilador que no tiene que chequear, ni mandar argumentos a pid para invocarla, y que luego, el tipo de datos que tendrá en *eax* será de tipo *int* sin signo. Recordar que en ASM no hay tipo de datos!

Recepción de Parámetros

También es necesario poder acceder a los argumentos de las funciones, por ejemplo si se implementara la función puts en una biblioteca de funciones, sería algo equivalente a:

```
//libc.c
#define STDOUT 1

int sys_write(int fd, void *buffer, int size);

int puts(const char* str) {
     int len = strlen(str);
     return sys_write(STDOUT, (void *) str, len);
}
```

Y la implementación de la función sys_write podría ser algo cómo:

```
; libasm.asm
GLOBAL sys_write

ALIGN 4
sys_write:
       push ebp
       mov ebp, esp

       push ebx ;preservar ebx

       mov eax, 0x4
       mov ebx, [ebp+8]   ; fd
       mov ecx, [ebp+12]  ; buffer
       mov edx, [ebp+16] ; length
       int 0x80

       pop ebx

       mov esp, ebp
       pop ebp
       ret
```

Ejercicio 3

Realice una implementación para cumplir con el ejemplo. Utilice la directiva ***-fno-builtin*** para indicarle a C que no debe linkeditar contra la biblioteca estándar.

Variables

Hasta ahora, siempre que hizo falta reservamos espacio en la zona de datos de un programa. Por ejemplo, para guardar una variable. Pero, ¿Qué ocurriría en el caso de que una función sea recursiva?

¿Cómo hace C a que las variables automáticas sólo vivan en el scope de la función? La respuesta es guardar esas variables en el stack, así, cuando un stack frame se destruye, al mismo tiempo se libera la memoria utilizada para sus variables.

Ejemplo

```
//suma.c

int suma(int a, int b) {
     int resultado;
     resultado = a + b;
     return resultado;
}
```

Una posible salida en Assembler podría ser:

```
suma:
       push ebp
       mov  ebp, esp

       sub  esp, 16 ; reservar espacio para las variables
       mov  eax, DWORD PTR [ebp+12]
       mov  edx, DWORD PTR [ebp+8]
       add  eax, edx
       mov  [ebp-4], eax ; resultado
       mov  eax, [ebp-4]

       leave  ; otra forma de desarmar el stack frame
       ret
```

De esta forma, la variable resultado, existe únicamente para la función, no tiene visibilidad para una función externa.

Reponder:

- ¿Porqué no se debe devolver un arreglo local a una función cómo valor de retorno?
- ¿Porque restan 16 bytes de ESP y no simplemente 4?
- ¿Porqué luego, de haber obtenido el resultado de la suma, que queda en el registro EAX, se lo guarda en la zona de memoria asignada a la variable resultado y luego se la vuelve a leer de ese lugar?

Ejemplo 4

Genere la salida en Assembler con GCC de la siguiente función:

```
//factorial.c
int factorial(int n) {
	if (n == 0)
		return 1;
	int factorial_n_1 = factorial(n-1);
	return n*factorial_n_1;
}
```

Identifique a que dirección de memoria relativa corresponde la variable *factorial_n_1*.

Ahora es obvio que se puede llamar de manera recursiva, ya que cada invocación de la función tendrá su propia versión de dicha variable.

Ejercicios

Ejercicio 5

Escriba una función en Assembler que retorne siempre el valor 7. Llamela con C y muestre por salida estándar dicho valor.

Ejercicio 6

Enumere las diferencias de declarar las variables y utilizarlas de las siguientes formas.

a.

```
//variables1.c
int foo() {
    int numero;
}
```

b.

```
//variables2.c
int foo() {
    int numero = 21;
}
```

c.

```
//variables3.c
int numero;
int foo() {
    numero = 21;
}
```

d.

```
int foo() {
    static int numero = 21;
}
```

e.

```
extern int numero;
int foo() {
     numero = 10;
}
```

```
int numero = 21;
int bar() {
   numero = 30;
}
```

f.

```
static int numero = 10;
int foo() {
   numero = 20;
}
```

Ejercicio 7

Declare en una función de C, un arreglo de las siguientes formas:

- Sin inicializar
- Con inicialización (int numeros[20] = {0}, ó char msg[] = "mensaje")
- Sin inicializar y luego realizando una escritura en el índice 10
- Con inicialización y luego realizando una escritura en el índice 10
- De manera global, sin inicializar
- De manera global inicializando.

Primero suponga cómo debería hacer C para su declaración, y luego compare sus suposiciones con la salida en assembler.

Ejercicio 8

Suponga que ud. está implementando la biblioteca estándar de C. Basándose en la guía anterior, prepare el contexto para llamar a la función inicial (main), y luego termine el programa con el valor de retorno que haya indicado la función main. Su *loader* debe utilizar la etiqueta *_start*

Ejercicio 9

Cómo parte de la implementación de su biblioteca estándar de C, implemente las funciones que le permitan acceder al sistema operativo para:

- Cerrar un programa
- Leer de un file descriptor
- Escribir a un file descriptor (ya implementado)
- Abrir un archivo y cerrarlo

Lista de system calls:
http://docs.cs.up.ac.za/programming/asm/derick_tut/syscalls.html,
http://syscalls.kernelgrok.com/

Escriba un programa que haga uso de su nueva biblioteca para abra un archivo, imprima línea por línea con el número de línea a la derecha y luego lo cierre. Recuerde compilar con la directiva **-fno-builtin**

Ejercicio 10

Escriba un programa que imprima por salida estándar el fabricante del procesador. Investigue la función *cpuid* de assembler.

Ejercicio 11

a. Implemente la función en C fibonacci(n), que reciba un número, y devuelva el número de fibonacci que corresponda.
b. Compile a mano, una posible salida en assembler de dicho programa.
c. Comprube su resultado con el de haber generado el código con **gcc**
d. Haga un seguimiento, paso a paso, de la pila por cada instrucción de assembler ejecutada para un fibonacci de 3.

Ejercicio 12

Dado el siguiente programa:

```
;ejemplo.s
main:
       push   ebp
       mov    ebp, esp
       and    esp, -16
       sub    esp, 32
       mov    DWORD PTR [esp+19], 1819043176
       mov    DWORD PTR [esp+23], 1870078063
       mov    DWORD PTR [esp+27], 174353522
       mov    BYTE PTR [esp+31], 0
       lea    eax, [esp+19]
       mov    DWORD PTR [esp], eax
       call   magia
       lea    eax, [esp+19]
       mov    DWORD PTR [esp], eax
       call   printf
       mov    eax, 0
       leave

magia:
       push   ebp
       mov    ebp, esp

       sub    esp, 16
       jmp    .L4
.L6:
       mov    eax, DWORD PTR [ebp+8]
       movzx  eax, BYTE PTR [eax]
       cmp    al, 96
       jle    .L5
       mov    eax, DWORD PTR [ebp+8]
       movzx  eax, BYTE PTR [eax]
       cmp    al, 122
       jg     .L5
       mov    eax, DWORD PTR [ebp+8]
       movzx  eax, BYTE PTR [eax]
       mov    BYTE PTR [ebp-1], al
       movzx  eax, BYTE PTR [ebp-1]
       sub    eax, 32
       mov    BYTE PTR [ebp-1], al
       mov    eax, DWORD PTR [ebp+8]
       movzx  edx, BYTE PTR [ebp-1]
       mov    BYTE PTR [eax], dl
.L5:
       add    DWORD PTR [ebp+8], 1
.L4:
       mov    eax, DWORD PTR [ebp+8]
       movzx  eax, BYTE PTR [eax]
       test   al, al
       jne    .L6
       leave
       ret
```

Indique lo que está haciendo, haga un seguimiento de la pila.

Ejercicio 13

Deduzca cómo se pasan las estructuras entre funciones, ¿Que diferencia hay entre pasar una estructura por referencia o por copia? ¿Cómo es el retorno?

x86_64 ABI

Hasta ahora trabajamos con la ABI de 32 bits de C y Assembler. Básicamente los conceptos son los mismos, lo único que cambia es el pasaje de parámetros.

El pasaje de parámetros de realiza de la siguiente forma:

- Se cargan los argumentos en los registros
- Se llama a la función
- Los argumentos se copian al stack y se referencian desde ahí.

Clasificación:

Los argumentos se clasifican de la siguiente forma:

- INTEGER: char, short, int, long, long long y punteros
- SSE: floats y doubles
- MEMORY: datos mayores a un quadword (8 bytes) y datos desalineados

Y se pasan de la siguiente forma:

Si el dato es INTEGER,
    se van ocupando los registros `rdi`, `rsi`, `rdx`, `rcx`, `r8` y `r9` en orden
Si el dato es SSE,
    se van ocupando los registros `xmm0` a `xmm7` en orden
Si el dato es MEMORY, se pasan por stack y devuelven por stack, de izquierda a derecha (igual que en 32 bits)

Para devolver los valores,

Si el dato es INTEGER,
    se utiliza `rax` y `rdx`
Si el dato es SSE,
    se retorna por `xmm0` y `xmm1`

Los registros a preservar son:

- rbp
- rsp
- rbx
- r12
- r13
- r15

Esto implica que hay funciones que aunque tengan distinto prototipo, tienen la misma forma de pasar parámetros, por ejmplo las functiones

```
int foo(int arg1, double arg2);
```

tiene el mismo prototipo que la función:

```
int bar(double arg2, int arg1);
```

En síntesis:

```
typedef struct {
  int a, b;
  double d;
} structparm;
structparm s;
int e, f, g, h, i, j, k;
long double ld;
double m, n;
__m256 y;

extern void func (int e, int f,
                  structparm s, int g, int h,
                  long double ld, double m,
                  __m256 y,
                  double n, int i, int j, int k);

func (e, f, s, g, h, ld, m, y, n, i, j, k);
```

Los registros van a quedar de la siguiente forma:

General Purpose Registers	Floating Point Registers	Stack Frame Offset
%rdi: e	%xmm0: s.d	0: ld
%rsi: f	%xmm1: m	16: j
%rdx: s.a,s.b	%ymm2: y	24: k
%rcx: g	%xmm3: n	
%r8: h		
%r9: i		

Ejercicios

Realice los ejercicios de pasaje de parámetros, pero esta vez en 64 bits.

Para compilar, reemplazar los argumentos **elf32** por **elf64** de **nasm** y **-m32** por **-m64** en gcc.