

Lab 2: assembler x86 y call conventions

[9557] Organización del Computador
Curso: Moreno-Simó
Segundo cuatrimestre 2019

Alumno: Nicolás Continanza
Padrón: 95576
Email: ncontinanza94@gmail.com Fecha de entrega:

Índice

1. x86-write	2
2. x86-call	3
3. x86-libc	5
4. x86-ret	7
5. x86-ebp	8
6. x86-argv	9
7. x86-frames	9

1. x86-write

- *¿Por qué se le resta 1 al resultado de `sizeof`?:* porque `sizeof` tomará en cuenta el caracter `'\0'` al final del arreglo. Restándole 1 al resultado, se obtendrá la cantidad de caracteres del arreglo sin incluir el caracter final `'\0'`.
- *¿Funcionaría el programa si se declarase `msg` como `const char *msg = "...";`?* ¿Por qué?: no funcionaría como se espera, pues `sizeof` devolverá el tamaño de un puntero y no el del arreglo de caracteres.
- *Compilar ahora `libc_hello.S` y verificar que funciona correctamente. Explicar el propósito de cada instrucción, y cómo se corresponde con el código C original:* las instrucciones `push` se ocupan de apilar los valores pasados por parámetro, para que luego sean desapilados en las funciones `write` y `exit`. El apilado se hace en el orden inverso al que deben ser leídos por la función debido al principio de funcionamiento de una pila (LIFO). Las instrucciones `call` "llamarán."^a las funciones `write` y `exit` en cada caso. Esto es, un `jmp` a la dirección donde se haya puesto el label que acompañe a la instrucción, para luego volver a ese punto luego de un `ret`.
- *Examinar, con `objdump -S libc_hello`, el código máquina generado e indicar el valor de `len` en la primera instrucción `push`. Explicar el efecto del operador `.` en la línea `.set len, . - msg` :* el valor de `len` en la primera instrucción `push` es `0xE`. El `.` representa el valor del *program counter*. En esa línea, restará al *program counter* el valor de `msg`.
- *Mostrar un hex dump de la salida del programa en assembler.*

```
nicolascontinanza@bsmith-ThinkPad-T470:~/Desktop/FIUBA/Orga/LABs/LAB2$ ./libc_hello | od -t x1 -c
00000000 48 65 6c 6c 6f 2c 20 77 6f 72 6c 64 21 0a
          H e l l o ,      w o r l d ! \n
00000016
```

Figura 1: hex dump de la salida del programa en assembler

- *Cambiar la directiva `.ascii` por `.asciz` y mostrar el hex dump resultante con el nuevo código. ¿Qué está ocurriendo? ¿Qué ocurre con el valor de `len`?:* Ahora en el hex dump se ve cómo se incluye

un caracter nulo al final de msg. El valor de `len` pasa a ser `0xF`, esto es, una unidad más que el valor que tenía anteriormente.

2. x86-call

```
(gdb) x/6i $pc
=> 0x804843b <main>:    push    $0xe
    0x8048440 <main+5>: push    $0x804a020
    0x8048445 <main+10>: push    $0x1
    0x8048447 <main+12>: call    0x8048320 <write@plt>
    0x804844c <main+17>: push    $0x7
    0x804844e <main+19>: call    0x8048300 <_exit@plt>
```

```
(gdb) display/1i $pc # Opcional
=> 0x804843b <main>:    push    $0xe
```

```
(gdb) stepi
10                push $msg
=> 0x8048440 <main+5>: push    $0x804a020
```

```
(gdb) si
11                push $1
=> 0x8048445 <main+10>: push    $0x1
```

```
(gdb)
14                call write
=> 0x8048447 <main+12>: call    0x8048320 <write@plt>
```

```
(gdb) x/4i $sp
0xffffcd10: add    %eax, (%eax)
// Guarda en %eax la suma de su contenido y lo apuntado.
// Esto es, si %eax = n, (%eax) = %eax + n.

0xffffcd12: add    %al, (%eax)
// Apunta %eax a la dirección que resulta de sumar la parte
// baja del acumulador con lo apuntado por %eax.
```

```
0xffffcd14: and    %ah,0xe0804(%eax)
// Guarda en %eax + 0xe0804 la conjunción entre el contenido
// de %ah y la dirección a la que apunta %eax
// desplazada 0xe0804 posiciones.
```

```
0xffffcd1a: add    %al,(%eax)
// Apunta %eax a la dirección que resulta de sumar la parte
// baja del acumulador con lo apuntado por %eax.
```

```
(gdb) si
0x8048320 <write@plt>: jmp    *0x804a014
```

```
(gdb) x/1i %sp
0xffffccbc: test    %al,(%si)
```

Finalmente, sustituir la instrucción `call write` por `jmp write`, y añadir el código y preparaciones necesarias para que el programa siga funcionando (ayuda: usar una etiqueta `posicion_retorno`: dentro de `main` para computar la dirección de retorno). Las llamadas a `strlen` y `_exit` pueden quedar. Incluir esta última versión en la entrega.

```
.globl main
// Call libc's wrappers to write(2) and _exit(2):
//
//     void _exit(int status);
//     ssize_t write(int fd, const void *buf, size_t count);
//
main:
    // Call convention: arguments on the stack (reverse order).
    push $msg
    call strlen
    push %eax
    push $msg
    push $1
```

```

    push $posicion_retorno
    // No declaration needed; asm assumes symbols always exist.
    jmp write

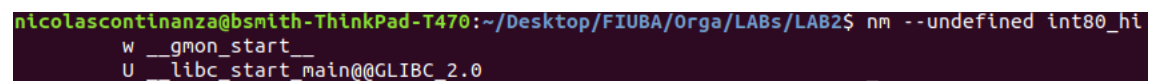
posicion_retorno:
    push $7
    call _exit

.data
msg:
    .asciz "Hello, world!\n"

```

3. x86-libc

1. *Compilar y ejecutar el archivo completo `int80_hi.S`. Mostrar la salida de `nm -undefined` para este nuevo binario.*



```

nicolascontinanza@bsmith-ThinkPad-T470:~/Desktop/FIUBA/Orga/LABs/LAB2$ nm --undefined int80_hi
w __gmon_start__
U __libc_start_main@@GLIBC_2.0

```

Figura 2: salida de `nm -undefined`

2. *Escribir una versión modificada llamada `sys_strlen.S` en la que, eliminando la directiva `.set len`, se calcule la longitud del mensaje (tercer parámetro para `write`) usando directamente `strlen(3)` (el código será muy parecido al de ejercicios anteriores).*

```

#include <sys/syscall.h> // SYS_write, SYS_exit

.globl main
main:
    push $msg
    call strlen

    mov %eax, %edx        // %edx == third argument (count)
    mov $SYS_write, %eax  // %eax == syscall number

```

```

    mov $1, %ebx           // %ebx == first argument (fd)
    mov $msg, %ecx         // %ecx == second argument (buf)

    int $0x80

    mov $SYS_exit, %eax
    mov $7, %ebx
    int $0x80

.data
msg:
    .ascii "Hello, world!\n"

```

3. a) *¿qué significa que un registro sea callee-saved en lugar de caller-saved?*: significa que quien llame a un procedimiento que hace uso de este registro, puede estar seguro de que el registro en cuestión preservará su valor al terminar dicho procedimiento.
- b) *en x86 ¿de qué tipo, caller-saved o callee-saved, es cada registro según la convención de llamadas de GCC?*: EAX, ECX y EDX son *caller-saved*. Los demás, *callee-saved*.

4. Pruebas de compilación:

a) int80_hi:

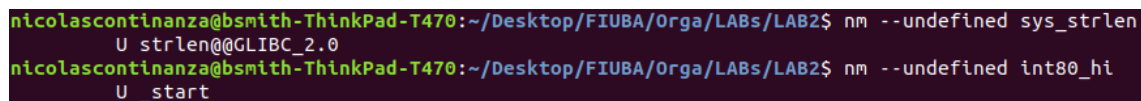
-nodefaultlibs	-nostartfiles
undefined reference to '___libc_csu_fini'	
undefined reference to '___libc_csu_init'	
undefined reference to 'main'	
undefined reference to '___libc_start_main'	
error: ld returned 1 exit status	OK

b) sys_strlen:

-nodefaultlibs	-nostartfiles
undefined reference to strlen	
error: ld returned 1 exit status	OK

Responder: ¿alguno de los dos archivos compila con -nostdlib?:
 int80_hi no compila, con un error que reza `undefined reference to 'strlen'`. sys_strlen compila, pero con el warning `warning: cannot find entry symbol _start; defaulting to 00000000080480b8`

5. *Mostrar la salida de nm -undefined para el binario sys_strlen, y explicar las diferencias respecto a int80_hi.*



```

nicolascontinanza@bsmith-ThinkPad-T470:~/Desktop/FIUBA/Orga/LABs/LAB2$ nm --undefined sys_strlen
U strlen@@GLIBC_2.0
nicolascontinanza@bsmith-ThinkPad-T470:~/Desktop/FIUBA/Orga/LABs/LAB2$ nm --undefined int80_hi
U _start
  
```

Figura 3: Salida de nm -undefined para ambos binarios

4. x86-ret

Se pide ahora modificar int80_hi.S para que, en lugar de invocar a _exit(), la ejecución finalice sencillamente con una instrucción ret. ¿Cómo se pasa en este caso el valor de retorno?: El valor de retorno se pasa a través del registro %eax.

```
#include <sys/syscall.h> // SYS_write, SYS_exit
```

```
.globl main
```

```
main:
```

```

    mov $SYS_write, %eax    // %eax == syscall number
    mov $1, %ebx           // %ebx == first argument (fd)
    mov $msg, %ecx         // %ecx == second argument (buf)
    mov $len, %edx         // %edx == third argument (count)
    int $0x80

    mov $0, %eax
    ret
  
```

```
.data
```

```
msg:
```

```
    .ascii "Hello, world!\n"
```



```
.set len, . - msg
```

5. x86-ebp

1. ¿Qué valor sobrescribió GCC cuando usó `mov $7, (%esp)` en lugar de `push $7` para la llamada a `_exit`? sobrescribió el valor del tope de la pila (en este caso, `0x1`).
2. La versión C no restaura el valor original de los registros `%esp` y `%ebp`. Cambiar la llamada a `_exit(7)` por `return 7`, y mostrar en qué cambia el código generado. ¿Se restaura ahora el valor original de `%ebp`? al hacer las modificaciones enunciadas, obtenemos:

```
7          return 7;
0x08048445 <+31>: mov     $0x7,%eax
0x0804844a <+36>: mov     -0x4(%ebp),%ecx
0x0804844d <+39>: leave
0x0804844e <+40>: lea     -0x4(%ecx),%esp
0x08048451 <+43>: ret
```

Podemos ver que, en este caso, se tiene en cuenta el valor de `%ebp` y `%esp`, y después de mover `0x7` al registro `%eax`, se restaura el estado del stack pointer con la instrucción `leave`, y luego el de `%ebp`.

3. Crear un archivo llamado `lib/exit.c` y usar en `hello.c` `my_exit(7)`. ¿Qué ocurre con `%ebp`? obtenemos la siguiente salida:

```
10          my_exit(7);
0x08048475 <+31>: movl    $0x7, (%esp)
0x0804847c <+38>: call   0x804848e <my_exit>

11
0x08048481 <+43>: mov     $0x0,%eax
0x08048486 <+48>: mov     -0x4(%ebp),%ecx
0x08048489 <+51>: leave
0x0804848a <+52>: lea     -0x4(%ecx),%esp
0x0804848d <+55>: ret
```

Podemos ver que se hace un llamado a `my_exit` sin *pushear* el valor de retorno a la pila (se usa la instrucción `movl` en su lugar), pero se maneja a `%ebp` igual que en el punto anterior.

4. En *hello.c*, cambiar la declaración de *my_exit* a:

```
extern void __attribute__((noreturn)) my_exit(int status);
```

y verificar qué ocurre con `%ebp`, relacionándolo con el significado del atributo *noreturn*.: obtenemos la salida:

```
10      my_exit(7);  
0x08048475 <+31>: movl    $0x7, (%esp)  
0x0804847c <+38>: call    0x8048481 <my_exit>
```

Dado que la ejecución no retornará debido al atributo **noreturn**, `%ebp` mantendrá su valor original sin sufrir alteraciones posteriores.

6. x86-argv

7. x86-frames