SISTEMAS OPERATIVOS – 95.03

DOCENTES:
MENDEZ, MARIANO
SIMÓ PIQUERES, ADEODATO

TRABAJO PRÁCTICO N° 1 : “LAB x86”

FECHA: 05/04/19

ALUMNO: SCHISCHLO, FRANCO DANIEL

PADRÓN: 100615

Ej: x86-write

Sobre el código anterior, responder:

•¿Por qué se le resta 1 al resultado de *sizeof?*

Respuesta: Se le resta 1 para no tomar en cuenta el '\0' que indica el fin del string.

•¿Funcionaría el programa si se declarase *msg* como `const char *msg = "...";`? ¿Por qué?

Respuesta: Funciona, pero solo imprime los primeros 3 caracteres, ya que al hacer sizeof(puntero), el puntero ocupa 4 bytes, y al restarle 1, resulta en 3 la cantidad de bytes a escribir.

•Explicar el efecto del operador `.` en la línea `.set len, . - msg`.

Respuesta: El punto indica que es la dirección de memoria actual. La cuenta es restarle el valor del label "msg" y finalmente asignarlo a len.

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. Después:

Respuesta:

```
push $len
push $msg
push $1
```

1) Se apilan los 3 parámetros de write en el stack (orden inverso).

```
call write
```

2) Se llama a la función.

```
Push $7
```

3) Se apila el parámetro de exit.

```
call _exit
```

4) Se llama a la función.

•Mostrar un hex dump de la salida del programa en assembler. Se puede obtener con el comando `od`: `./libc_hello | od -t x1 -c`

Respuesta:

```
0000000  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
0000016
```

•Cambiar la directiva `.ascii` por `.asciz` y mostrar el hex dump resultante con el nuevo código. ¿Qué está ocurriendo?

Respuesta: Se le agrega un \0 al final del string, porque así es el formato asciz.

```
0000000  48  65  6c  6c  6f  2c  20  77  6f  72  6c  64  21  0a 00
          H   e   l   l   o   ,       w   o   r   l   d   !  \n \0
0000017
```

Ej: x86-call

Mostrar en una sesión de GDB cómo imprimir las mismas instrucciones usando la directiva x $pc y el modificador *i*. Después, usar el comando stepi *(step instruction)* para avanzar la ejecución hasta la llamada a *write*. En ese momento, mostrar los primeros cuatro valores de la pila justo antes e inmediatamente después de ejecutar la instrucción call, y explicar cada uno de ellos

Respuesta:

```
Reading symbols from ./libc_hello...done.
(gdb) b main
Breakpoint 1 at 0x804846b: file libc_hello.S, line 4.
(gdb) r
Starting program: /home/franco/Documents/Sistemas
Operativos/Sistemas-Operativos/Trabajos/labx86/libc_hello

Breakpoint 1, main () at libc_hello.S:4
4           push $msg
(gdb) x/8i $pc
=> 0x804846b <main>:    push   $0x804a024
   0x8048470 <main+5>:  call   0x8048330 <strlen@plt>
   0x8048475 <main+10>: push   %eax
   0x8048476 <main+11>: push   $0x804a024
   0x804847b <main+16>: push   $0x1
   0x804847d <main+18>: call   0x8048350 <write@plt>
   0x8048482 <main+23>: push   $0x7
   0x8048484 <main+25>: call   0x8048320 <_exit@plt>
(gdb) display/i $pc
1: x/i $pc
=> 0x804846b <main>:    push   $0x804a024
(gdb) stepi
5           call strlen
1: x/i $pc
=> 0x8048470 <main+5>:  call   0x8048330 <strlen@plt>
(gdb) stepi
0x08048330 in strlen@plt ()
1: x/i $pc
=> 0x8048330 <strlen@plt>:   jmp    *0x804a010
(gdb) finish
Run till exit from #0  0x08048330 in strlen@plt ()
main () at libc_hello.S:6
6           push %eax
1: x/i $pc
=> 0x8048475 <main+10>: push   %eax
(gdb) stepi
7           push $msg
1: x/i $pc
=> 0x8048476 <main+11>: push   $0x804a024
```

```
(gdb) stepi
8           push $1
1: x/i $pc
=> 0x804847b <main+16>:  push   $0x1
(gdb) stepi
9           call write
1: x/i $pc
=> 0x804847d <main+18>:  call   0x8048350 <write@plt>
(gdb) x/4xw $sp
0xffffcb7c:    0x00000001     0x0804a024     0x0000000e
     0x0804a024
```

Explicación:
0x00000001: Último valor apilado, indica el FD.
0x0804a024: Dirección donde se encuentra almacenado msg
0x0000000e: 15 (longitud de msg)
0x0804a024: Dirección donde se encuentra almacenado msg

```
(gdb) si
0x08048350 in write@plt ()
1: x/i $pc
=> 0x8048350 <write@plt>:    jmp    *0x804a018
(gdb) x/4xw $sp
0xffffcb78:    0x08048482     0x00000001     0x0804a024
     0x0000000e
```

Explicación:
Los valores previamente apilados, agregandole "0x08048482" que es la dirección de la próxima instrucción.

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.

Respuesta: libc_hello.S

```
.globl main

main:
        push $msg
        call strlen
        push %eax
        push $msg
        push $1
        push $ret_pos
        jmp write
ret_pos:
        push $7
        call _exit

.data
```

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

Ej: x86-libc

Se pide:

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

Respuesta:

w __gmon_start__
w _ITM_deregisterTMCloneTable
w _ITM_registerTMCloneTable
w _Jv_RegisterClasses
U __libc_start_main@@GLIBC_2.0

2. Escribir una versión modificada llamada *int80_strlen.S* en la que, de nuevo 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). **Mostrar la salida de nm --undefinedpara este nuevo binario.**

Respuesta:

w __gmon_start__
w _ITM_deregisterTMCloneTable
w _ITM_registerTMCloneTable
w _Jv_RegisterClasses
U __libc_start_main@@GLIBC_2.0
U strlen@@GLIBC_2.0

3. En la convención de llamadas de GCC, ciertos registros son *caller-saved* (por ejemplo *%ecx*) y ciertos otros *callee-saved* (por ejemplo *%ebx*). Responder:

- ¿qué significa que un registro sea *callee-saved* en lugar de *caller-saved*?

Respuesta:
Callee saved: El valor del registro debe ser restaurado a su valor original, si fue alterado durante el llamado a una función, por la misma función, antes de retornar.
Caller saved: El valor del registro debe ser guardado en el stack antes de hacer un llamado, y posteriormente restaurado, una vez que la función retorno a él.

- en x86 ¿de qué tipo, *caller-saved* o *callee-saved*, es cada registro según la convención de llamadas de GCC?

Respuesta:
Callee-saved registers: %ebp, %ebx, %esi, %edi
Caller-saved registers: %eax, %ecx, %edx

4. Copiar *int80_strlen.S* a un nuevo archivo *sys_strlen.S*, renombrando main a _start en el proceso. **Mostrar la salida de nm –undefined para este nuevo binario**, y describir brevemente las diferencias con los casos anteriores.

Respuesta: U strlen@@GLIBC_2.0

-nostartupfiles le indica al linker que no utilice las funciones estándar de inicio del sistema, ni tampoco que enlace el codigo de aquellas funciones.

Ej: x86-ret

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

Respuesta: Cuando se utiliza sencillamente ret (si esta linkeado con la biblioteca de C runtime), retorna a C runtime, que enventualmente termina el proceso. Libc se encarga de propagar el valor de retorno de main a la llamada exit. Se hace mov $7,%eax al final de la función.

Se pide también escribir un nuevo programa, *libc_puts.S*, que use una instrucción `ret` en lugar de una llamada a *_exit*. Al contrario que *int80_hi.S*, este programa sí modifica la pila. Para simplificar la tarea, *libc_puts.S* puede usar `puts(3)` en lugar de `write(2)`

Respuesta: (El programa en este caso termina con 7, sin mover a eax 7).

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

.globl main

main:

          push $msg

          call puts

          pop %eax

          ret

.data

msg:

        .ascii "Hello, world!"
```

Se pide mostrar, usando un *catchpoint*, una sesión de GDB el momento en que el binario *libc_puts* realiza la llamada a *exit* con `int $0x80` o `sysenter`, y dónde reside dicha instrucción. En este caso, el interés residirá en detener la ejecución en el momento de la llamada al syscall *exit*.

En el momento en que se llegue a la condición de corte y se detenga la ejecución, se debe mostrar el código colindante con `disas` y los marcos de ejecución mediante el comando `backtrace` de GDB.

•Finalmente se indicará, para cada función en el *backtrace*, en qué archivo o biblioteca se aloja, esto es, la correspondencia entre las funciones del backtrace, las posiciones de memoria donde reside el código de cada función y los archivos donde se aloja el código.

```
Reading symbols from ./libc_puts...done.
(gdb) catch syscall exit_group
Catchpoint 1 (syscall 'exit_group' [252])
(gdb) r
Starting program: /home/franco/Documents/Sistemas
Operativos/Sistemas-Operativos/Trabajos/labx86/libc_puts
Hello, world!

Catchpoint 1 (call to syscall exit_group), 0xf7fd7fe9 in
__kernel_vsyscall ()
(gdb) disas
Dump of assembler code for function __kernel_vsyscall:
   0xf7fd7fe0 <+0>: push   %ecx
   0xf7fd7fe1 <+1>: push   %edx
   0xf7fd7fe2 <+2>: push   %ebp
   0xf7fd7fe3 <+3>: mov    %esp,%ebp
   0xf7fd7fe5 <+5>: sysenter
   0xf7fd7fe7 <+7>: int    $0x80
=> 0xf7fd7fe9 <+9>: pop    %ebp
   0xf7fd7fea <+10>:    pop    %edx
   0xf7fd7feb <+11>:    pop    %ecx
   0xf7fd7fec <+12>:    ret
End of assembler dump.
(gdb) bt
#0  0xf7fd7fe9 in __kernel_vsyscall ()
#1  0xf7eb4588 in _exit () from /lib32/libc.so.6
#2  0xf7e3372a in ?? () from /lib32/libc.so.6
#3  0xf7e337cf in exit () from /lib32/libc.so.6
#4  0xf7e1d643 in __libc_start_main () from /lib32/libc.so.6
#5  0x08048331 in _start ()
(gdb) info shared
From        To          Syms Read   Shared Object Library
0xf7fd9860  0xf7ff28dd  Yes (*)     /lib/ld-linux.so.2
0xf7e1c750  0xf7f4588d  Yes (*)     /lib32/libc.so.6
(*): Shared library is missing debugging information.
```

Funciones	Dirección	Biblioteca/Archivo*
__kernel_vsyscall()	0xf7fd7fe0	/lib/ld-linux.so.2
_exit()	0xf7eb4578	/lib32/libc.so.6
exit()	0xf7e337b0	/lib32/libc.so.6
__libc_start_main	0xf7e1d540	/lib32/libc.so.6
_start()	0x8048310	libc_puts.S

Ej: x86-ebp

1. ¿Qué valor sobreescribió GCC cuando usó mov $7, (%esp) en lugar de push $7 para la llamada a *_exit*? ¿Tiene esto alguna consecuencia?

Respuesta: Sobreescribió el último parámetro pusheado al stack antes de hacer el call, es el valor hexadecimal 1. No.

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*?

Respuesta: Si, se restaura con en “0x08048449 <+62>: pop %ebp”.

```
_exit(7);
   0x0804846f <+52>:  movl   $0x7,(%esp)
   0x08048476 <+59>:  call   0x8048300 <_exit@plt>


return 7;
   0x0804843f <+52>:  mov    $0x7,%eax
   0x08048444 <+57>:  lea    -0x8(%ebp),%esp
   0x08048447 <+60>:  pop    %ecx
   0x08048448 <+61>:  pop    %edi
   0x08048449 <+62>:  pop    %ebp
   0x0804844a <+63>:  lea    -0x4(%ecx),%esp
   0x0804844d <+66>:  ret
```

3. Crear un archivo llamado *lib/exit.c* con la siguiente función:

```
#include <unistd.h>

void my_exit(int status) {
    _exit(status);
}
```

y usar en *hello.c* my_exit(7):

```
extern void my_exit(int status);

int main(void) {
    // ...
    my_exit(7);
}
```

¿Qué ocurre con *%ebp*?

Nota: el binario se puede compilar con make hello tras añadir la siguiente línea al *Makefile*:

hello: hello.c lib/exit.c

Respuesta: Antes de volver a main, luego de my_exit(), se restaura el valor de %ebp.

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.

Respuesta: Al añadirle el atributo (noreturn), esto le indica a GCC que una vez que se llame a esa función, no va a retornar. Por lo tanto, no se ve restaurado el valor de %ebp.

Ej: x86-frames

Responder, en términos del frame pointer %ebp de una función f:

•¿dónde se encuentra (de haberlo) el primer argumento de f?

Respuesta: El primer argumento se encuentra en 8(%ebp).

•¿dónde se encuentra la dirección a la que retorna f cuando ejecute ret?

Respuesta: La dirección se encuentra en 4(%ebp).

•¿dónde se encuentra el valor de %ebp de la función anterior, que invocó a f?
Respuesta: El valor ebp anterior se encuentra en 0(%ebp).
•¿dónde se encuentra la dirección a la que retornará la función que invocó a f?
Respuesta: Se encuentra en 4(0(%ebp)).
Se pide ahora escribir una función: void backtrace();
que obtenga, usando __builtin_frame_address(0), el frame pointer actual, e imprima la secuencia de marcos anidados en el formato que se indica a continuación:

#numfrm [FP] ADDR (ARG1 ARG2 ARG3)

donde para cada frame FP es el frame pointer (registro %ebp), ADDR es el punto de retorno a la función, y ARGS sus tres primeros argumentos.

Incluir en la entrega:

1. el código de la función backtrace.
Respuesta:

```
// Formato #numfrm [FP] ADDR ( ARG1 ARG2 ARG3 )
void backtrace() {
     uintptr_t* ebp = __builtin_frame_address(0);
     unsigned int numfrm = 1;

    ebp = *ebp;
```

```
    while (ebp != 0) {
        printf("#%u [%p] %p ( %p %p %p )\n",
                numfrm,ebp,*(ebp+1),
                *(ebp+2),*(ebp+3),*(ebp+4));

        ebp = *ebp;
        numfrm++;
    }
}
```

2. una sesión de GDB en la que se muestre la equivalencia entre el comando `bt` de GDB y el código implementado; en particular, se debe incluir:

- la salida del comando `bt` al entrar en la función *backtrace*
- la salida del programa al ejecutarse la función *backtrace* (el número de frames y sus direcciones de retorno deberían coincidir con la salida de `bt`)
- usando los comandos de selección de frames, y antes de salir de la función *backtrace*, el valor de *%ebp* en cada marco de ejecución detectado por GDB (valores que también deberían coincidir).

Respuesta:

```
Reading symbols from backtrace...done.
(gdb) b backtrace
Breakpoint 1 at 0x80484eb: file backtrace.c, line 5.
(gdb) r
Starting program: /home/franco/Documents/Sistemas
Operativos/Sistemas-Operativos/Trabajos/labx86/backtrace

Breakpoint 1, backtrace () at backtrace.c:5
warning: Source file is more recent than executable.
5    void backtrace() {
(gdb) bt
#0  backtrace () at backtrace.c:5
#1  0x08048540 in my_write (fd=2, msg=0x8048691, count=15) at
backtrace.c:22
#2  0x0804859d in recurse (level=0) at backtrace.c:31
#3  0x08048587 in recurse (level=1) at backtrace.c:29
```

```
#4  0x08048587 in recurse (level=2) at backtrace.c:29
#5  0x08048587 in recurse (level=3) at backtrace.c:29
#6  0x08048587 in recurse (level=4) at backtrace.c:29
#7  0x08048587 in recurse (level=5) at backtrace.c:29
#8  0x080485af in start_call_tree () at backtrace.c:35
#9  0x080485ca in main () at backtrace.c:39
(gdb) list
1    #include <stdint.h>
2    #include <stdio.h>
3    #include <unistd.h>
4     // Formato #numfrm [FP] ADDR ( ARG1 ARG2 ARG3 )
5    void backtrace() {
6         uintptr_t* ebp = __builtin_frame_address(0);
7         unsigned int numfrm = 1;
8
9        ebp = *ebp;
10
(gdb) until 19
#1 [0xffffca88] 0x804859d ( 0x2 0x8048691 0xf )
#2 [0xffffcaa8] 0x8048587 ( (nil) (nil) 0xf7ffdad0 )
#3 [0xffffcac8] 0x8048587 ( 0x1 0x1 0xf7fd3490 )
#4 [0xffffcae8] 0x8048587 ( 0x2 0x1 0xc2 )
#5 [0xffffcb08] 0x8048587 ( 0x3 0xf7ffd918 0xffffcb30 )
#6 [0xffffcb28] 0x8048587 ( 0x4 0x2f 0xf7e11dc8 )
#7 [0xffffcb48] 0x80485af ( 0x5 0x7 0xf7e33830 )
#8 [0xffffcb68] 0x80485ca ( 0xf7fb53dc 0xffffcb90 (nil) )
#9 [0xffffcb78] 0xf7e1d637 ( 0xf7fb5000 0xf7fb5000 (nil) )
backtrace () at backtrace.c:19
19   }
(gdb) up
```

```
#1  0x08048540 in my_write (fd=2, msg=0x8048691, count=15) at
backtrace.c:22
22       backtrace();
(gdb) p/x $ebp
$1 = 0xffffca88
(gdb) up
#2  0x0804859d in recurse (level=0) at backtrace.c:31
31           my_write(2, "Hello, world!\n", 15);
(gdb) p/x $ebp
$2 = 0xffffcaa8
(gdb) up
#3  0x08048587 in recurse (level=1) at backtrace.c:29
29           recurse(level - 1);
(gdb) p/x $ebp
$3 = 0xffffcac8
(gdb) up
#4  0x08048587 in recurse (level=2) at backtrace.c:29
29           recurse(level - 1);
(gdb) p/x $ebp
$4 = 0xffffcae8
(gdb) up
#5  0x08048587 in recurse (level=3) at backtrace.c:29
29           recurse(level - 1);
(gdb) p/x $ebp
$5 = 0xffffcb08
(gdb) up
#6  0x08048587 in recurse (level=4) at backtrace.c:29
29           recurse(level - 1);
(gdb) p/x $ebp
$6 = 0xffffcb28
(gdb) up
```

```
#7  0x08048587 in recurse (level=5) at backtrace.c:29
29          recurse(level - 1);
(gdb) p/x $ebp
$7 = 0xffffcb48
(gdb) up
#8  0x080485af in start_call_tree () at backtrace.c:35
35      recurse(5);
(gdb) p/x $ebp
$8 = 0xffffcb68
(gdb) up
#9  0x080485ca in main () at backtrace.c:39
39      start_call_tree();
(gdb) p/x $ebp
$9 = 0xffffcb78
(gdb) up
Initial frame selected; you cannot go up.
(gdb) frame 0
#0  backtrace () at backtrace.c:19
19   }
(gdb) c
Continuing.
=> write(2, 0x8048691, 15)
Hello, world!
[Inferior 1 (process 9968) exited normally]
```