Ejemplo sobre el uso de pila en Assembler

Arquitectura del Computador - LCC - FCEIA-UNR

Agosto 2021

En este ejemplo se pretende mostrar algunas cuestiones sobre el uso de la pila. También es útil para familiarizarse con GDB.

Dado el siguiente código en Assembler X86-64 en el archivo `stack.s`:

```
 1 .data
 2 a: .quad 45
 3 b: .quad 56
 4
 5 .text
 6 .global main
 7
 8 main:
 9         movq a, %rdi
10         movq b, %rsi
11         call suma
12         xorq %rax, %rax
13         retq
14
15 suma:
16         pushq %rbp
17         movq %rsp, %rbp
18         pushq $99
19         addq %rdi, %rsi
20         addq -8(%rbp), %rsi
21         movq %rsi, %rax
22         movq %rbp, %rsp
23         popq %rbp
24         retq
```

Compilar utilizando la opción `-g` para poder *debuggear* utilizando GDB:

```
$ gcc -g stack.s
```

Luego ejecutar utilizando GDB:

```
gdb ./a.out
```

Una vez que estamos dentro de la sesión de *debugging*, ponemos un *breakpoint* en `main` y ejecutamos el comando `run`:

```
(gdb) br main
Breakpoint 1 at 0x4004b6: file stack.s, line 9.
(gdb) r
Starting program: /home/dferoldi/2020/a.out
```

```
Breakpoint 1, main () at stack.s:9
9 movq a, %rdi
```

Luego vamos ejecutando línea a línea utilizando el comando `next` hasta llegar a la línea 11 donde se hace el llamado a la función `suma`:

```
9 movq a, %rdi
(gdb) n
10 movq b, %rsi
(gdb) n
11 call suma
```

Veamos el valor del registro `rsp` (*stack pointer*):

```
(gdb) i r rsp
rsp            0x7fffffffe1e8 0x7fffffffe1e8
```

Vemos que el registro `rsp` está "apuntando" a la dirección `0x7fffffffe1e8`. Ahora sigamos con la ejecución del código con el comando `step` para poder entrar a la función `suma`:

```
(gdb) s
suma () at stack.s:16
16 pushq %rbp
```

Veamos ahora el nuevo valor del registro `rsp`:

```
(gdb) i r rsp
rsp            0x7fffffffe1e0 0x7fffffffe1e0
```

Vemos que el valor del registro disminuyó en 8 bytes. Veamos que hay almacenado en la dirección a la que apunta:

```
(gdb) x $rsp
0x7fffffffe1e0: 0x004004cb
```

De acuerdo a lo que hemos visto en teoría, el valor `0x004004cb` es la dirección de retorno de la función `suma`, es decir la dirección a la que hay retornar cuando finalice la función. ¿Cómo podemos verificarlo? Podemos ver si en la dirección `0x004004cb` efectivamente está la instrucción siguiente al llamado a la función, es decir la instrucción de la línea 12. Esto lo podemos lograr utilizando el comando `x` con la opción `i`:

```
(gdb) x/i 0x4004cb
   0x4004cb <main+21>: xor    %rax,%rax
```

Vemos que efectivamente en la dirección que se "pusheó" se encuentra almacenada la instrucción de la línea 12 a la cual tiene que retornar el flujo del proceso una vez que finalice la función `suma`.

Ahora ejecutemos una línea más, verifiquemos el nuevo valor del registro `rsp` y lo que hay almacenado en la dirección a la que apunta:

```
(gdb) n
17 movq %rsp, %rbp
(gdb) i r rsp
rsp            0x7fffffffe1d8 0x7fffffffe1d8
(gdb) x/1xg $rsp
0x7fffffffe1d8: 0x0000000000000000
```

Vemos que el valor de `rsp` disminuyó otros 8 bytes y que en la dirección a la que apunta está almacenado el valor del registro `rbp` anterior al llamado a función, por lo cual ahora podremos trabajar con el registro `rbp` y modificarlo

sin inconvenientes porque posteriormente podremos recuperarlo. Recordemos que el registro `rbp` es *calle saved*, por lo cual es responsabilidad de la función llamada de salvar y luego restaurar el valor que tenía antes de llamar a la función.

Avancemos una línea más y veamos el contenido de los registros `rsp` y `rbp`:

```
(gdb) i r rsp rbp
rsp            0x7fffffffe1d8 0x7fffffffe1d8
rbp            0x7fffffffe1d8 0x7fffffffe1d8
```

Vemos que ahora ambos registros apuntan a la misma dirección, la cual es el comienzo del marco de activación de la función `suma`.

Si ejecutamos una línea más, "pusheamos" el valor 99 a la pila. Este valor actúa como variable local dentro de la función `suma`. Esta variable local la podemos referenciar de manera relativa utilizando el registro `rbp`, dado que este registro queda "anclado" señalando el comienzo del marco de activación de la función:

```
(gdb) x/d $rbp-8
0x7fffffffe1d0: 99
```

La siguiente línea suma los argumentos de la función que por convención de llamada vienen en los registros `rdi` y `rsi`:

```
19 addq %rdi, %rsi
```

Luego, al resultado en `rsi` se le suma el valor de la variable local referenciando de manera relativa al registro `rbp`:

```
20 addq -8(%rbp), %rsi
```

Posteriormente, en la línea 21 se carga el resultado en el registro `rax`, debido a la convención de llamada.

Veamos ahora que hacen las líneas 22 y 23. Este el denominado "epílogo". En la línea 22 se "mueve" el puntero `rsp` hacia el comienzo del marco de activación, el cual está apuntado por `rbp`:

```
22 movq %rbp, %rsp
(gdb) i r rbp rsp
rbp            0x7fffffffe1d8 0x7fffffffe1d8
rsp            0x7fffffffe1d8 0x7fffffffe1d8
```

Vemos que ahora ambos registros apuntan nuevamente al comienzo del marco de activación. Luego, en la línea 23 se restaura en el registro `rbp` el valor que tenía antes de llamar a la función y que habíamos "salvado" en la pila:

```
23 popq %rbp
(gdb) i r rbp
rbp            0x0 0x0
```

En este punto el registro `rsp` está apuntando a la dirección donde se "pusheo" la direción de retorno de la función `suma`. Por lo tanto, al ejecutar la instrucción en la línea 24 se retornará a la línea 12. Si chequeamos el contenido del registro `rax`, vemos que efectivamente tenemos el resultado de las operaciones que se realizaron en la función `suma`:

```
(gdb) i r rax
rax            0xc8 200
```

Es importante notar que si no hubiéramos ejecutado la línea 22, el registro `rsp` no hubiera estado apuntando a la dirección de retorno al pretender retornar con la instrucción `ret` en la línea 24 y por lo tanto se hubiera producido una violación de segmento.

Finalmente, luego de ejecutar la línea 12 se obtuvo un valor 0 en el registro `rax`, debido a las propiedades de la operación *exclusive or*, el cual será el valor de retorno de `main` una vez ejecutada la línea 13. Esto se puede verificar con el comando `echo` inmediatamente después de haber sido ejecutado el proceso:

```
$ ./a.out
$ echo $?
0
```

En la siguiente figura podemos ver el esquema de la pila con las direcciones de memoria a la izquierda y a la derecha cómo van apuntando los registros `rbp` y `rsp` a lo largo de la ejecución del proceso: