Ejemplo sobre convención de llamada para C en X86-64

Arquitectura del Computador - LCC - FCEIA-UNR

Septiembre 2020

En este ejemplo se pretende mostrar algunas cuestiones sobre la convención de llamada para C en X86-64. También es útil para familiarizarse con GDB.

Dado el siguiente código en lenguaje C en el archivo `funcion_larga.c`:

```
 1 struct punto{
 2         int x;
 3         int y;
 4 } p;
 5
 6 int funcion(int, int, char, float, int, long, long, long, struct punto);
 7
 8 int main(){
 9 int a=3, b=5, c=6;
10 char d='a';
11 float f=3.14;
12 long g=78, h=99, l=100;
13
14 p.x=60;
15 p.y=90;
16
17 int r = funcion(a,b,d,f,c,g,h,l,p);
18
19 return 0;
20 }
21
22
23 int funcion(int a,int b,char d,float f,int c,long g,long h,long l, struct punto p){
24 return a+b;
25 }
```

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

```
$ gcc -g funcion_larga.c
```

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 0x4004be: file funcion_larga.c, line 9.
(gdb) r
```

```
Starting program: /home/dferoldi/2020/a.out

Breakpoint 1, main () at funcion_larga.c:9
```

Luego vamos ejecutando línea a línea utilizando el comando `next` hasta llegar a la línea 17 donde se hace el llamado a la función `funcion` y entramos a la misma con el comando `step`:

```
(gdb) s
funcion (a=3, b=5, d=97 'a', f=3.1400001, c=6, g=78, h=99, l=100, p=...) at funcion_larga.c:24
```

En este punto es interesante ver el contenido de los registros `rdi, rsi, rdx, rcx, r8, r9` para verificar lo visto en teoría sobre la convención de llamada para C en X86-64:

```
(gdb) i r rdi rsi rdx rcx r8 r9
rdi            0x3      3
rsi            0x5      5
rdx            0x61     97
rcx            0x6      6
r8             0x4e     78
r9             0x63     99
```

Vemos que efectivamente los primeros seis argumentos enteros fueron pasados en los registros mencionados en ese mismo orden. Notemos que el tercer argumento es de tipo `char`, por lo que se pasó el valor 97 que corresponde al carácter `a` en código ASCII.

El cuarto argumento es de tipo `float`. Entonces veamos el contenido del registro `xmm0`:

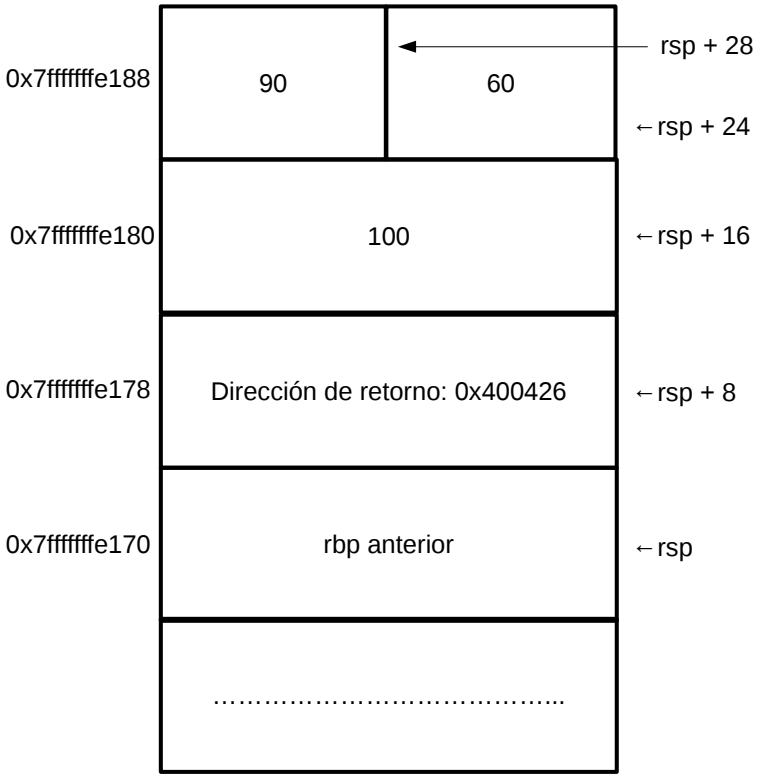
```
(gdb) i r xmm0
xmm0           {v4_float = {0x3, 0x0, 0x0, 0x0}, v2_double = {0x0, 0x0},
v16_int8 = {0xc3, 0xf5, 0x48, 0x40, 0x0 <repeats 12 times>},
v8_int16 = {0xf5c3, 0x4048, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0}, v4_int32 = {0x4048f5c3, 0x0, 0x0, 0x0},
  v2_int64 = {0x4048f5c3, 0x0}, uint128 = 0x00000000000000000000000004048f5c3}
```

Aquí vemos el contenido del registro en varios formatos. Si nos focalizamos en el formato entero de 64 bits vemos el contenido `0x4048f5c3`. Esta secuencia de 32 bits corresponde al número real 3,1400001049 de acuerdo a la norma IEEE 754 para números en punto flotante simple precisión. Notemos que no corresponde al valor 3,14 sino a un valor cercano debido al error de representación.

El octavo argumento es de tipo `long`. Este argumento es el séptimo argumento de tipo entero, por lo tanto según la convención de llamada será pasado utilizando la pila. Podemos verificarlo viendo el contenido de memoria:

```
(gdb) x/d $rsp+16
0x7fffffffe180: 100
```

Vemos que está el valor 100 de la variable `l`. Notemos que en `(rsp)` tendremos el valor del `rbp` antes de la llamada y que en `8(rsp)` la dirección de retorno. Esta distribución se puede observar en la siguiente figura:



Finalmente, el último argumento es una estructura. La convención de llamada establece que los argumentos complejos son pasados por pila. Esta estructura está compuesta por dos enteros (4 bytes cada uno). Por lo tanto podemos verificar cómo fueron pasados los miembros de la estructura de la siguiente manera:

```
(gdb) x/d $rsp+24
0x7fffffffe188: 60
(gdb) x/d $rsp+28
0x7fffffffe18c: 90
```

Vemos que `p.x` está alojado en `rsp+24` mientras que `p.y` está alojado 4 bytes más allá.