Ejemplo sobre complemento a dos

Arquitectura del Computador - LCC - FCEIA-UNR

Septiembre 2020

En este ejemplo se pretende mostrar algunas cuestiones sobre las operaciones en complemento a dos y las bandas *overflow* y carry. También es útil para familiarizarse con GDB.

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

```
.data

a: .byte 56
b: .byte 84

.text

.global main
main:
    movb a, %al
    movb b, %bl
    addb %bl, %al
    retq
```

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

```
$ gcc -g complemento_dos.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 complemento_dos.s, line 11.
(gdb) r
Starting program: /home/dferoldi/2020/a.out

Breakpoint 1, main () at complemento_dos.s:11
```

Luego vamos ejecutando línea a línea utilizando el comando `next`:

```
11 movb a, %al
(gdb) n
12 movb b, %bl
(gdb) n
13 addb %bl, %al
```

Ahora podemos ver el contenido de los subregistros `al` y `bl`:

```
(gdb) i r al bl
al             0x38 56
bl             0x54 84
```

ejecutamos una línea más para realizar la suma y vemos el resultado en `al`:

```
(gdb) n
(gdb) i r al
al             0x8c -116
```

Vemos que el resultado no es el que esperábamos, dado que $56 + 84 = 140$ y no -116 como estamos viendo. Es más, vemos que sumamos dos números positivos y obtuvimos como resultado un número negativo. ¿Qué sucedió? Estamos trabajando con datos de un byte (8 bits). Por lo tanto, el rango de números representables si trabajamos con signo es $-128 \leq rango \leq 127$, con lo cual el número 140 no es representable con 8 bits. Si además chequeamos el contenido del registro `eflags`, vemos que la bandera *overflow* (`OF`) se encendió lo que nos indica que el resultado es incorrecto si trabajamos con números con signo.

```
(gdb) i r eflags
eflags         0xa82 [ SF IF OF ]
```

Ahora bien, ¿por qué el resultado fue -116? Lo que hizo la ALU fue realizar la operación suma bit a bit, en este caso una suma. Por lo tanto, lo que realizó fue la siguiente operación:

$$\begin{array}{r} 0111000 \\ + \\ 01010100 \\ \hline 10001100 \end{array} \tag{1}$$

Efectivamente, la secuencia de bits $(10001100)_2$ representa el valor -116 en decimal utilizando complemento a dos: $-2^7 + 2^3 + 2^2 = -116$. Es ALU sumó las dos secuencias de bits bit a bit y luego GDB no muestra el resultado como número con signo utilizando complemento a dos.

Notar que la misma secuencia de bits del resultado obtenido en (1) representa el valor 140 si lo interpretamos como número sin signo: $2^7 + 2^3 + 2^2 = 140$. Es decir, si estamos trabajando con número sin signos entonces el resultado es correcto. La bandera *carry* (`CF`) apagada efectivamente nos indica que el resultado es correcto si interpretamos a los números como números si signo.

Entonces, la conclusión general es que el resultado será correcto o no dependiendo de cómo interpretemos a los números. La computadora en su nivel más básico meramente hace operaciones entre secuencias de bits. Cómo se interpretan esas secuencias de bits será responsabilidad del programador.

Ejemplo sobre punteros en C

Arquitectura del Computador - LCC - FCEIA-UNR

Septiembre 2020

En este ejemplo se pretende repasar el tema de punteros en C. Si bien es un ejemplo muy elemental de punteros en C, nos va a servir para practicar con GDB.

Dado el siguiente código en lenguaje C `punteros.c`:

```
 1 #include <stdio.h>
 2
 3 int main()
 4 {
 5 int a=145; //Declaración de variable entera de tipo entero
 6 int *puntero; //Declaración de variable puntero de tipo entero
 7 puntero = &a; //Asignación de la dirección memoria de a
 8
 9 printf("El valor de a es: %d\nEl valor de *puntero es: %d\n", a , *puntero);
10 printf("La dirección de memoria de *puntero es: %p\n", puntero);
11
12 return 0;
13 }
```

Si compilamos y ejecutamos el código, la salida es la siguiente:

```
$ ./a.out
El valor de a es: 145
El valor de *puntero es: 145
La dirección de memoria de *puntero es: 0x7ffc33bd4954
```

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

```
$ gcc -g punteros.c
```

y ejecutemos con GDB:

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

Una vez que estamos dentro de la sesión de *debugging*, ponemos un *breakpoint* en `main` y ejecutamos el comando `run`. Luego vamos ejecutando línea a línea hasta ejecutar los dos `printf`:

```
(gdb) br main
Breakpoint 1 at 0x40050e: file punteros.c, line 5.
(gdb) r
Starting program: /home/dferoldi/2020/a.out

Breakpoint 1, main () at punteros.c:5
5 int a=145; //Declaración de variable entera de tipo entero
(gdb) n
```

```
7 puntero = &a; //Asignación de la dirección memoria de a
(gdb) n
9 printf("El valor de a es: %d\nEl valor de *puntero es: %d\n",a,*puntero);
(gdb) n
El valor de a es: 145
El valor de *puntero es: 145
10 printf("La dirección de memoria de *puntero es: %p\n",puntero);
(gdb) n
La dirección de memoria de *puntero es: 0x7fffffffe1d4
12 return 0;
```

Ahora verifiquemos lo anterior utilizando comandos de GDB. En primer lugar, podemos usar el comando `print` para ver el contenido de la variable `a`, luego podemos imprimir a puntero dicha variable y finalmente examinar el contenido de la memoria utilizando el comando `examine`[1]:

```
(gdb) p a
$1 = 145
(gdb) p &a
$2 = (int *) 0x7fffffffe1d4
(gdb) x/d &a
0x7fffffffe1d4: 145
```

Efectivamente, vemos que en la dirección `0x7fffffffe1d4`, que es la dirección almacenada en la variable tipo puntero a entero `puntero`, se encuentra almacenado el valor `145`.

[1] Utilizamos la opción `d` para ver el resultado en formato decimal.

Ejemplo sobre uso de memoria

Arquitectura del Computador - LCC - FCEIA-UNR

Septiembre 2020

En este ejemplo se pretende mostrar algunas cuestiones y precauciones al momento de trabajar con datos en memoria. También es útil para familiarizarse con GDB.

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

```
.data

a: .long 0x11223344
b: .long 0x55667788

.text
.global main

main:
    movq a, %rax    #<----- Línea 1
    retq
```

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

```
$ gcc -g memory_ejemplo.s
```

Luego ejecutar utilizando GDB:

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

En primer lugar, es interesante ver en que parte de la memoria están los datos. Para ello, una vez que hayamos corrido el programa dentro de GDB, podemos ver la dirección de la etiqueta `a` de la siguiente manera:

```
(gdb) info address a
```

El resultado es:

```
Symbol "a" is at 0x600870 in a file compiled without debugging.
```

También podríamos haber imprimido un puntero a la etiqueta `a`:

```
(gdb) print &a
$1 = (<data variable, no debug info> *) 0x600870
```

Ahora, es interesante verificar que efectivamente en esa dirección está el dato en cuestión:

```
(gdb) x/1xb &a
0x600870: 0x44
```

Aquí hemos usado el comando `x`, el cual deriva de "examine", y además como opciones hemos indicado que nos muestre un byte en formato hexadecimal. Entonces vemos que en la dirección `0x600870` hay almacenado un `0x44`. Ahora veamos qué hay almacenado en la siguiente dirección de memoria:

```
(gdb) x/1xb 0x600871
0x600871: 0x33
```

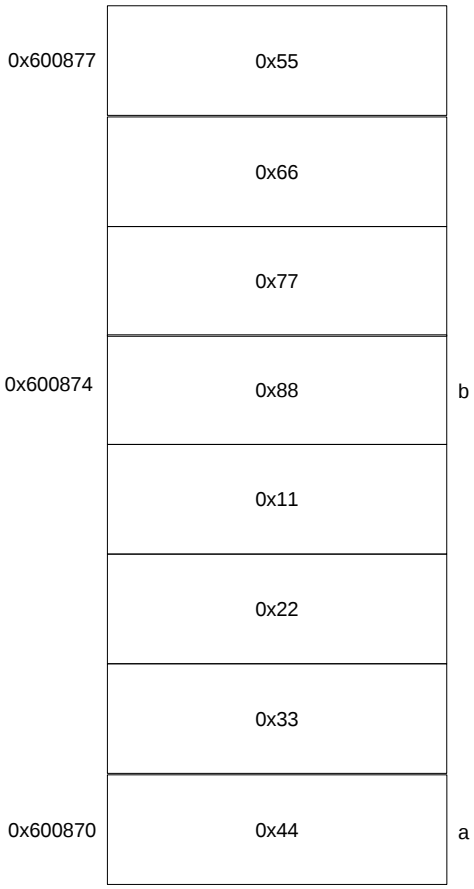
Esto es coherente si recordamos que en las arquitecturas Intel los datos mayores a un byte son almacenados en memoria utilizando el formato *little-endian*. Entonces, en la dirección etiquetada con `a` se encuentra almacenado el byte menos significativo y los siguientes bytes están almacenados hacia direcciones mayores.

Análogamente, podemos ver como está almacenado el dato a partir de la etiqueta `b`:

```
(gdb) x/1xw &b
0x600884: 0x55667788
```

donde la `w` es por *word* (4 bytes).

En la siguiente figura podemos ver una representación del esquema de memoria:



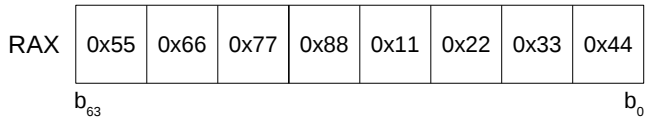
Ahora supongamos que queremos cargar algunos de estos datos en un registro. Por ejemplo, cargar el dato almacenado a partir de la etiqueta `a` al registro `rax` tal como se realiza en la línea 1 del código del ejemplo:

```
movq a, %rax
```

Veamos entonces el contenido del registro `rax`, una vez ejecutada la línea anterior:

```
(gdb) i r rax
rax            0x5566778811223344 6153737367135073092
```

El resultado se puede ver en el siguiente esquema, donde se observan los 8 bytes del registro `rax`:



¿Cómo se interpreta este resultado? La instrucción `movq a, %rax` cargó 8 bytes (debido al sufijo `q`) a partir de la dirección de memoria etiquetada con `a` (`0x600870`). En esta dirección comienza un dato tipo `long` (4 bytes). Entonces, cargó los 4 bytes almacenados a partir de la dirección etiquetada con `b` (`0x11223344`) más los 4 bytes a partir de la dirección de memoria etiquetada como `b` (`0x55667788`).

Como conclusión, este ejemplo muestra varias cuestiones:

- La memoria se va ocupando como bloques consecutivos de acuerdo a la directiva utilizada (por ejemplo, `.long` asigna un bloque de 4 bytes).
- Las etiquetas son formas de referenciar la dirección de memoria dónde comienza el bloque:

```
movq $a, %rax   #----> rax=0x600870
```

- Al momento de acceder a los datos almacenados en memoria hay que tener cuidado con el sufijo utilizado en la instrucción `mov`.

Ejemplo sobre el uso de la instrucción `lea`

Arquitectura del Computador - LCC - FCEIA-UNR

Septiembre 2020

En este ejemplo se pretende mostrar el uso de la instrucción `lea`. La instrucción `lea` (*load effective address*) se usa para poner una dirección de memoria en el destino. Este ejemplo también es útil para familiarizarse con GDB.

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

```
.data

a: .quad 0x1122334455667788

.text
.global main

main:
    leaq a, %rax    # línea 1
    movq $a, %rax   # Línea 2
    retq
```

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

```
$ gcc -g lea_ejemplo.s
```

Luego ejecutar utilizando GDB:

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

Colocar un breakpoint en `main`:

```
(gdb) br main
Breakpoint 1 at 0x4004b6: file ejemplo_lea.s, line 9.
```

Ejecutar con el comando `run`:

```
(gdb) r
```

Ejecutar la primera línea con el comando `next`:

```
(gdb) n
```

Observar el contenido del registro `rax`:

```
(gdb) i r rax
rax            0x600880 6293632
```

Vemos que en el registro `rax` quedó almacenada la dirección de la etiqueta `a`, lo cual podemos verificar de manera adicional mediante el comando:

```
(gdb) info address a
Symbol "a" is at 0x600880 in a file compiled without debugging.
```

Ahora veamos que sucede cuando ejecutamos la siguiente línea:

```
(gdb) n
(gdb) i r rax
rax            0x600880 6293632
```

Vemos que la línea 2 es equivalente a la línea 1.

Como conclusión, hemos ejecutado dos línea de código equivalentes pero que utilizan instrucciones diferentes. en efecto, si bien las instrucciones en las líneas 1 y 2 produjeron el mismo efecto, la instrucción `lea` es diferente a la dirección `mov` y permite realizar operaciones más complejas. Recordemos que la forma general de la instrucción `lea` es

```
leaq displacement(%base, %offset, multiplier), %dest
```

la cual corresponde a `%dest = displacement + %base + %offset * multiplier` donde `displacement` es una constante entera, `multiplier` es 2, 4 o 8 y `%dest`, `%offset` y `%base` son registros. Algunos de los operandos pueden no estar.

Ejemplos:

```
movq $100, %rax
movq $4, %rbx
leaq 16(%rax, %rbx, 2), %rcx        #---> rcx = 124
leaq 16(%rax, %rbx,), %rcx          #---> rcx = 120
leaq (%rax, %rbx,), %rcx           #---> rcx = 104
leaq (%rax), %rcx                   #---> rcx = 100
```

Además de brindar la posibilidad de realizar cálculos relativamente complejos[1], ese no es el propósito principal del la instrucción `lea`. El conjunto de instrucciones x86 fue diseñado para admitir lenguajes de alto nivel como Pascal y C, donde las matrices, especialmente las matrices de enteros, o estructuras pequeñas, son comunes. Consideremos, por ejemplo, una estructura que representa las coordenadas (x, y) de un punto:

```
struct Point
{
     int xcoord;
     int ycoord;
};
```

Ahora supongamos la siguiente declaración:

```
int y = points[i].ycoord;
```

donde `points[]` es una matriz de `Point`. Suponiendo que la base de la matriz ya está en `rbx`, la variable `i` está en `rax`, y `xcoord` e `ycoord` son cada uno de 32 bits (por lo que `ycoord` está en un desplazamiento de 4 bytes en la estructura), esta declaración se puede compilar como:

```
movq 4(%rbx, %rax, 8), %rdx
```

que cargará `y` en `rdx`. El factor de escala de 8 se debe a que cada punto tiene un tamaño de 8 bytes.

Ahora consideremos la siguiente expresión usando el operador "dirección de" &:

```
int *p = &points[i].ycoord;
```

[1]Es importante señalar que la instrucción `lea` no modifica el registro de banderas a diferencia de las instrucciones aritméticas

En este caso, no deseamos el valor de `ycoord`, sino su dirección. Aquí es donde entra la principal utilidad de la instrucción `lea` (*load effective addresss*). En lugar de un `mov`, el compilador puede generar:

```
leaq 4(%rbx, %rax, 8), %rsi
```

que cargará la dirección en `rsi`.

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`:

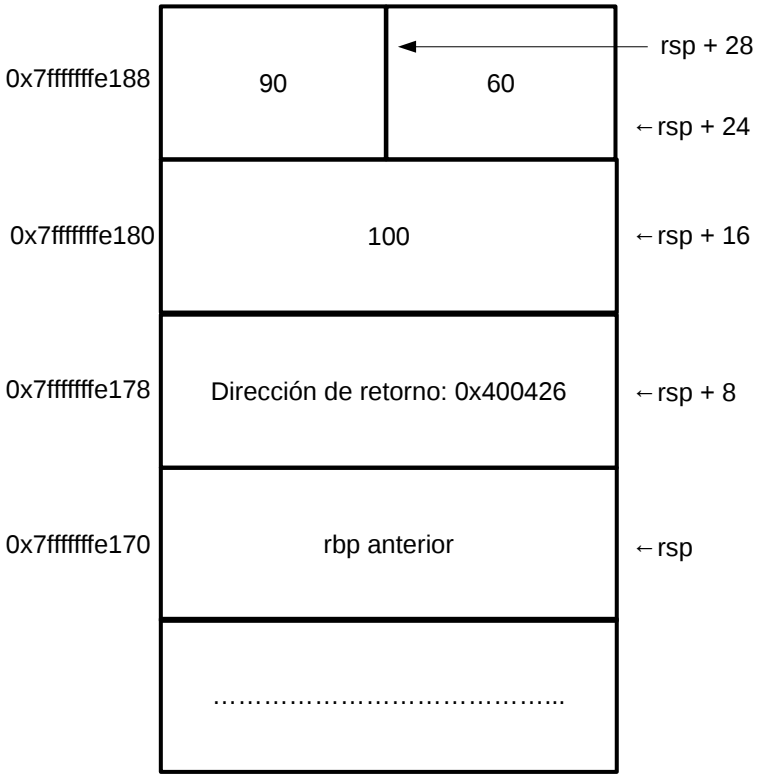
```
(gdb) i r xmm0
xmm0           {v4_float = {0x3, 0x0, 0x0, 0x0}, v2_double = {0x0, 0x0},
v16_int8 = {0xc3, 0xf5, 0x48, 0x40, 0x0 <repeats 12 times>},
v8_int16 = {0xf5c3, 0x4048, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0}, v4_int32 = {0x4048f5c3, 0x0, 0x0, 0x0},
  v2_int64 = {0x4048f5c3, 0x0}, uint128 = 0x00000000000000000000000004048f5c3}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

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.

Ejemplo sobre funciones en C con número variable de argumentos

Arquitectura del Computador - LCC - FCEIA-UNR

Septiembre 2020

En este ejemplo se pretende mostrar algunas cuestiones sobre las funciones en C con número variable de argumentos. También es útil para familiarizarse con GDB.

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

```
 1 #include <stdio.h>
 2 int main(int argc, char *argv[])
 3 {
 4 int i;
 5 for(i = 0; i < argc; i++) {
 6 printf("argv[%d]: %s\n", i, argv[i]);
 7 }
 8 return(0);
 9 }
```

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

```
$ gcc -g main_arg.c
```

Primero ejecutemos pasándole varios argumentos para ver la salida:

```
$ ./a.out Esta es una prueba
argv[0]: ./a.out
argv[1]: Esta
argv[2]: es
argv[3]: una
argv[4]: prueba
```

Ahora ejecutemos utilizando GDB:

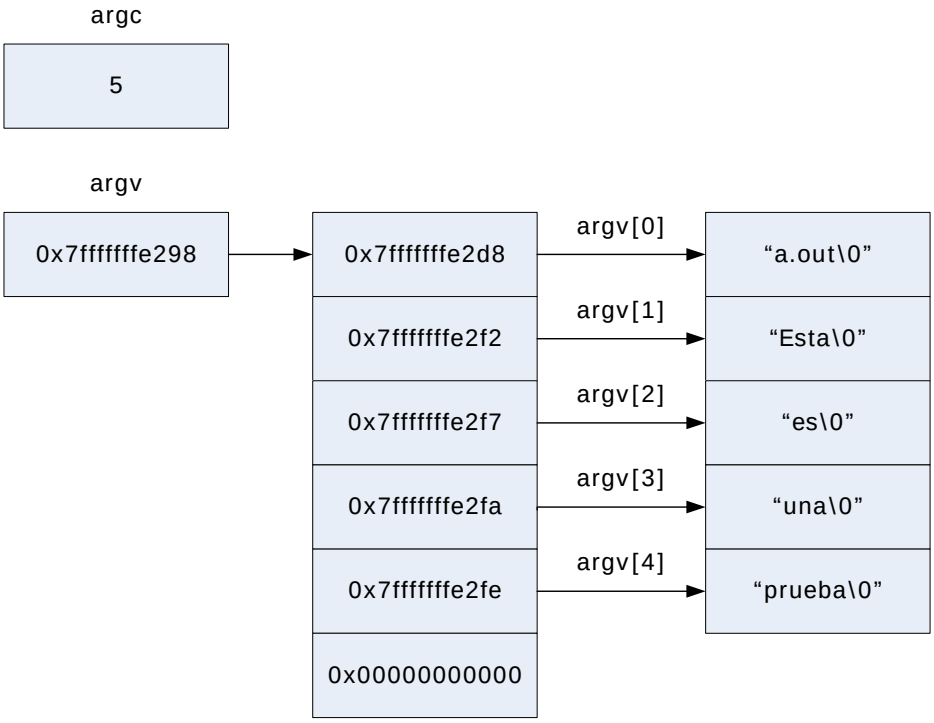
```
gdb ./a.out
```

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

```
(gdb) br main
Breakpoint 1 at 0x400515: file main_arg.c, line 5.
(gdb) r Esta es una prueba
Starting program: /home/dferoldi/2020/a.out Esta es una prueba

Breakpoint 1, main (argc=5, argv=0x7fffffffe298) at main_arg.c:5
```

Vemos que `argc` vale 5 y `argv` vale `0x7fffffffe298`. ¿Qué significa? En primer lugar Vemos que `argc` es el número de argumentos que le pasamos a la función, siendo el primer argumento la cadena `"./a.out"`, el segundo la cadena `"Esta"`, y así sucesivamente. En cuanto a `argv`, es un puntero a un arreglo de punteros, donde cada uno de estos punteros apunta a la dirección de cada una de las cadenas de caracteres que pasamos como argumentos. Esto lo podemos visualizar en la siguiente figura:



Es interesante verificar todo lo anterior utilizando GDB para entender mejor el mecanismo. En primer lugar podemos imprimir el valor de `argc` para verificar que tenemos el 5 correspondiente al número de argumentos pasados. Luego podemos imprimir el valor de `argv`, que es puntero al arreglo de punteros. Entonces luego examinamos el contenido de memoria en esa dirección para encontrar el puntero a la primera cadena pasada y así sucesivamente encontrar el resto de punteros. Luego podemos buscar en memoria el contenido de alguna de las cadenas. Por ejemplo, el primer carácter del segundo argumento. La siguiente lista de comando en GDB sirve para realizar lo anterior:

```
(gdb) p argc
$1 = 5
(gdb) p argv
$2 = (char **) 0x7fffffffe298
(gdb) x/1xg argv
0x7fffffffe298: 0x00007fffffffe4d8
(gdb) x/1xg argv+1
0x7fffffffe2a0: 0x00007fffffffe4f2
(gdb) x/1xg argv+2
0x7fffffffe2a8: 0x00007fffffffe4f7
(gdb) x/1xg argv+3
0x7fffffffe2b0: 0x00007fffffffe4fa
(gdb) x/1xg argv+4
0x7fffffffe2b8: 0x00007fffffffe4fe
(gdb) x/1xg argv+5
0x7fffffffe2c0: 0x0000000000000000
(gdb) x/1cb 0x7fffffffe4f2
0x7fffffffe4f2: 69 'E'
```

Notar que efectivamente la diferencia entre dos punteros consecutivos corresponde a la longitud de la cadena apuntada por el primero de dichos punteros, incluyendo al caracter nulo del final de la cadena. Finalmente, es importante destacar que si alguno de los argumentos pasados es un valor numérico habrá que convertirlo para poder utilizarlo como tal.

Ejemplo sobre el uso de pila en Assembler

Arquitectura del Computador - LCC - FCEIA-UNR

Septiembre 2020

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 "pusheo" 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: