Función main con argumentos por línea de comandos

Arquitectura del Computador - LCC - FCEIA-UNR

26 de octubre de 2022

En este tutorial se pretende mostrar algunas cuestiones sobre la función `main` en C con argumentos por línea de comando. También es útil para familiarizarse con GDB.

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

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

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:

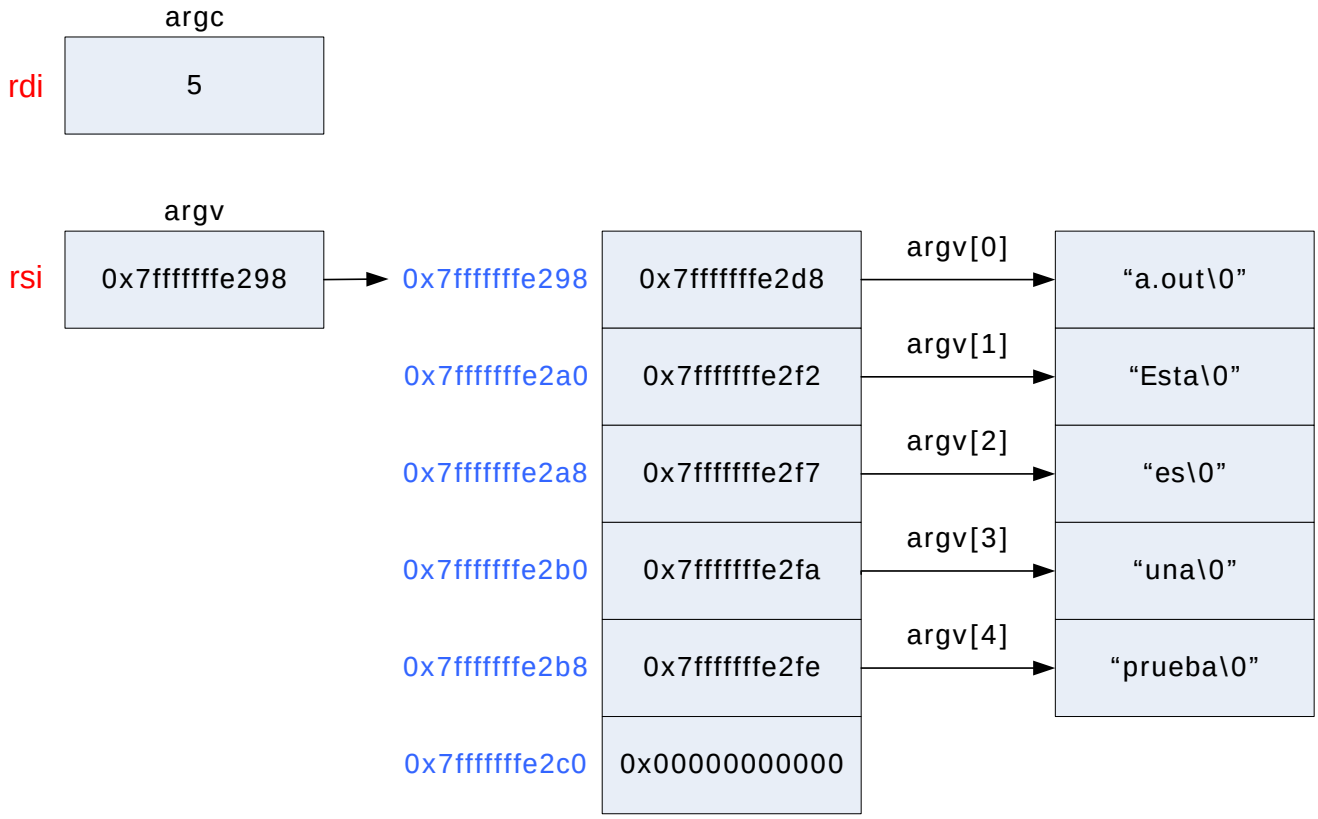
```
gdb ./a.out
```

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

Finalmente, mostramos como ejemplo una implementación en Assembler x86-64 para mostrar por pantalla la cantidad de argumentos ingresados y el primer argumento ingresado:

```
.data
str_1: .asciz "Cantidad de argumentos: %d\n"
str_2: .asciz "Primer argumento: %s\n"

.text
.global main
main:
prologo:
        pushq %rbp              # Salvamos el registro rbp
        movq %rsp, %rbp         # Usamos el registro rbp como puntero a la base del pila
        subq $16, %rsp          # Reservamos espacio en la pila
        movq %rdi, -8(%rbp)     # Guardamos el registro rdi en la pila
        movq %rsi, -16(%rbp)    # Guardamos el registro rsi en la pila
imprime_numero:
        movq -8(%rbp), %rsi     # rsi es el segundo argumento para llamar a printf
        movq $str_1, %rdi       # rdi es el primer argumento para llamar a printf
        xorq %rax, %rax         # La cantidad de argumentos de tipo vectorial es cero
        call printf             # Llamamos a printf para imprimir el primer mensaje
imprime_primero:
        movq -16(%rbp), %rsi    # Ahora en rsi tenemos la dirección del arreglo de punteros
        leaq 8(%rsi), %rax      # Apuntamos al siguiente puntero
        movq (%rax), %rsi       # dereferenciamos para tener la dirección de argv[1]
        movq $str_2, %rdi       # rdi es el primer argumento para llamar a printf
        xorq %rax, %rax         # La cantidad de argumentos de tipo vectorial es cero
        call printf             # Volvemos a llamar a printf
epilogo:
        movq %rbp, %rsp         # Restauramos el valor de rsp
        pop %rbp                # Restauramos el valor de rbp
        ret                     # Finalmente retornamos
```

Por lo tanto, si se compila como:

```
$ gcc -o arg_var -no-pie arg_var.s
```

y luego se ejecuta como:

```
$ ./arg_var Hola mundo
```

imprime lo siguiente:

```
Cantidad argumentos: 3
Primer argumento: Hola
```