

## Llamadas a biblioteca y llamadas al sistema

### x86-write

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

Es necesario restar 1 al resultado de `sizeof` porque definir de esa forma a `msg` le agrega un byte `'\0'` al final del string.

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

El programa no funciona correctamente ya que el operador `sizeof` devuelve 4, el tamaño del puntero `msg`, y no el tamaño de memoria alojada al contenido al que apunta.

#### ¿Qué tipo de entidad es `sizeof`: una función, un operador, una palabra reservada?

`sizeof` es un operador.

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

`.set` hace que el símbolo `len` tenga el valor de la expresión después de la coma, y `.` guarda la dirección de la instrucción actual, entonces `. - msg` guarda la diferencia entre la dirección actual y el tag `msg`, lo que resulta ser el largo del string.

#### 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.

Las tres instrucciones `push` ponen los argumentos con los que llamar a `write` en el stack mientras que la instrucción `call` pone en el stack la dirección de memoria actual (a la cual volver de la llamada a función) y salta la ejecución hacia `write`.

#### Mostrar un hex dump de la salida del programa en assembler.

```
$ ./libc_hello | od -t x1 -c
00000000 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
00000016
```

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

El string ahora es terminado en `'\0'`, por lo que se imprime este caracter extra.

```
$ ./libc_hello | od -t x1 -c
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
```

Mostrar cómo habría que reescribir la línea `push $len` para que el código siga escribiendo el número correcto de bytes. (Nota: no cambiar la definición de `len`.)

```
push $len -1
```

## x86-call

Mostrar en una sesión de GDB cómo imprimir las mismas instrucciones usando la directiva `x $pc` y el modificador adecuado. 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.

```
Reading symbols from ./libc_hello...(no debugging symbols found)...done.
(gdb) b main
Breakpoint 1 at 0x804846b
(gdb) r
Starting program: /home/vasectomio/sisop/lab1/libc_hello

Breakpoint 1, 0x0804846b in main ()
(gdb) x/12i $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>
    0x8048489 <main+30>: xchg    %ax,%ax
    0x804848b <main+32>: xchg    %ax,%ax
    0x804848d <main+34>: xchg    %ax,%ax
    0x804848f <main+36>: nop
(gdb) stepi
```

```

0x08048470 in main ()
(gdb)
0x08048330 in strlen@plt ()
(gdb) finis
Run till exit from #0  0x08048330 in strlen@plt ()
0x08048475 in main ()
(gdb) stepi
0x08048476 in main ()
(gdb)
0x0804847b in main ()
(gdb)
0x0804847d in main ()
(gdb) x/5i $pc
=> 0x0804847d <main+18>: call    0x08048350 <write@plt>
      0x08048482 <main+23>: push   $0x7
      0x08048484 <main+25>: call    0x08048320 <_exit@plt>
      0x08048489 <main+30>: xchg    %ax,%ax
      0x0804848b <main+32>: xchg    %ax,%ax
(gdb) x/4w $esp
0xffffcd8c:      0x00000001      0x0804a024      0x0000000e      0x0804a024
(gdb) stepi
0x08048350 in write@plt ()
(gdb) x/4w $esp
0xffffcd88:      0x08048482      0x00000001      0x0804a024      0x0000000e
(gdb)

```

Valores de la pila antes de la llamada a `write`

0x00000001: Primer argumento de `write`, el *file descriptor*.

0x0804a024: Segundo argumento de `write`, la dirección de memoria en la que comienza el mensaje.

0x0000000e: Tercer argumento de `write`, la cantidad de bytes a escribir.

0x0804a024: La dirección del mensaje, que quedó en el stack por la llamada anterior a `strlen`

Valores de la pila después de la llamada a `write`

0x08048482: Dirección a la cual retornar.

0x00000001: Primer argumento de `write`, el *file descriptor*.

0x0804a024: Segundo argumento de `write`, la dirección de memoria en la que comienza el mensaje.

0x0000000e: Tercer argumento de `write`, la cantidad de bytes a escribir. #  
x86-call

## x86-libc

Compilar y ejecutar el archivo completo `int80_hi.S`. Mostrar la salida de `nm --undefined` para este nuevo binario.

```
~/sisop/lab1 $
$ gcc int80_hi.S -m32 -I/usr/include/x86_64-linux-gnu -o int80_hi
~/sisop/lab1 $
$ ./int80_hi
Hello, world!
X $ ~/sisop/lab1 $
$ nm -u int80_hi
                 w __gmon_start__
                 w _ITM_deregisterTMCloneTable
                 w _ITM_registerTMCloneTable
                 w _Jv_RegisterClasses
                 U __libc_start_main@@GLIBC_2.0
```

Escribir una versión modificada, llamada `sys_strlen.S`, en la que se calcule la longitud del mensaje usando `strlen` (el código será muy parecido al de ejercicios anteriores).

`sys_strlen.S`

```
#include <sys/syscall.h> // SYS_write, SYS_exit
// See: <https://en.wikibooks.org/wiki/X86\_Assembly/Interfacing\_with\_Linux>.

.globl main
main:
    pushl $msg
    call strlen
    mov %eax,%edx
    mov $SYS_write, %eax // %eax == syscall number
    mov $1, %ebx         // %ebx == first argument (fd)
    mov $msg, %ecx       // %ecx == second argument (buf)
    int $0x80

    mov $SYS_exit, %eax
    mov $7, %ebx
    int $0x80

.data
msg:
    .asciz "Hello, world!\n"
```

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*?

Que un registro sea *callee-saved* significa que es responsabilidad de la función que está siendo llamada guardar su estado si necesita sobrescribirlos, y restaurar sus contenidos antes de retornar el control a la función que la llamó.

en x86 ¿de qué tipo es cada registro según la convención de llamadas de GCC?

`eax`, `ecx` y `edx` son *caller-saved*, y el resto son *callee-saved*.

al realizar un `syscall` de manera directa con `int $0x80` ¿qué registros son *caller-saved*?

`eax` es el unico registro que se sobrescribe al hacer una `syscall`, y contiene el valor de retorno de la misma. El resto son *callee-saved*

**En el archivo `sys_strlen.S` del punto anterior, renombrar la función `main` a `_start`, y realizar las siguientes cuatro pruebas de compilación:**

Archivo	-nodefaultlibs	-nostartfiles
<code>sys_strlen.S</code>	no compila	compila
<code>int80_hi.S</code>	no compila	no encuentra <code>_start</code> (warning)

¿alguno de los dos archivos compila con `-nostdlib`?

no, ninguno de los dos compila con `-nostdlib`

Añadir al archivo `Makefile` una regla que permita compilar `sys_strlen.S` sin errores, así como cualquier otro archivo cuyo nombre empiece por `sys`:

```
sys_%: sys_%.S
    $(CC) $(ASFLAGS) $(CPPFLAGS) -nostartfiles $< -o $@
```

Mostrar la salida de `nm --undefined` para el binario `sys_strlen`, y explicar las diferencias respecto a `int80_hi`.

```
$ nm --undefined int80_hi
         w __gmon_start__
         w _ITM_deregisterTMCloneTable
         w _ITM_registerTMCloneTable
         w _Jv_RegisterClasses
         U __libc_start_main@@GLIBC_2.0
~/sisop/lab1 $
$ nm --undefined sys_strlen
         U strlen@@GLIBC_2.0
```

La diferencia radica en que *int80\_hi* hace uso de los archivos de inicio de *glibc*, mientras que en *sys\_strlen*, la ejecución comienza directamente en la etiqueta *\_start*.

## x86-ret

Se pide ahora modificar *int80\_hi.S* para que, en lugar de invocar a *\_exit()*, la ejecución finalice sencillamente con una instrucción *ret*. ¿Cómo se pasa en este caso el valor de retorno?

En tal caso, el valor de retorno debe ser dejado en *eax*

```
.globl main
main:
    mov $SYS_write, %eax // %eax == syscall number
    mov $1, %ebx        // %ebx == 1st argument (fd)
    mov $msg, %ecx      // %ecx == 2nd argument (buf)
    mov $len, %edx      // %edx == 3rd argument (count)
    int $0x80

    mov $0, %eax
    ret
.data
msg:
.asciz "Hello, world!\n"
.set len, . - msg
```

Se pide también escribir un nuevo programa, *libc\_puts.S*, que use una instrucción *ret* en lugar de una llamada a *\_exit*.

*libc\_puts.S*

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

```
.globl main
main:
    push $msg
    call puts

    pop %eax
    mov $0, %eax
    ret
.data
msg:
.asciz "Hello, world!\n"
.set len, . - msg
```

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.

```
(gdb) r
Starting program: /home/vasectomio/sisop/lab1/libc_puts

Breakpoint 1, 0x0804840b in main ()
(gdb) info b
Num      Type           Disp Enb Address      What
1        breakpoint     keep y   0x0804840b <main>
          breakpoint already hit 1 time
(gdb) stepi
0x08048410 in main ()
(gdb)
0x080482e0 in puts@plt ()
(gdb) finish
Run till exit from #0  0x080482e0 in puts@plt ()
Hello, world!

0x08048415 in main ()
(gdb) catch syscall
Catchpoint 5 (any syscall)
(gdb) c
Continuing.

Catchpoint 5 (call to syscall exit_group), 0xf7fd8be9 in __kernel_vsyscall ()
(gdb) where
0  0xf7fd8be9 in __kernel_vsyscall ()
1  0xf7ea27d8 in _exit () from /lib/i386-linux-gnu/libc.so.6
2  0xf7e2094a in ?? () from /lib/i386-linux-gnu/libc.so.6
3  0xf7e209ef in exit () from /lib/i386-linux-gnu/libc.so.6
4  0xf7e0a643 in __libc_start_main () from /lib/i386-linux-gnu/libc.so.6
5  0x08048331 in _start ()
(gdb) disas
Dump of assembler code for function __kernel_vsyscall:
   0xf7fd8be0 <+0>:    push    %ecx
   0xf7fd8be1 <+1>:    push    %edx
   0xf7fd8be2 <+2>:    push    %ebp
   0xf7fd8be3 <+3>:    mov     %esp,%ebp
   0xf7fd8be5 <+5>:    sysenter
   0xf7fd8be7 <+7>:    int     $0x80
=> 0xf7fd8be9 <+9>:    pop     %ebp
   0xf7fd8bea <+10>:   pop     %edx
   0xf7fd8beb <+11>:   pop     %ecx
   0xf7fd8bec <+12>:   ret
```

End of assembler dump.

(gdb) si

[Inferior 1 (process 15352) exited normally]

## x86-watch

guión de gdb

Reading symbols from ./libc\_puts...(no debugging symbols found)...done.

(gdb) b main

Breakpoint 1 at 0x804840b

(gdb) r

Starting program: /home/vasectomio/sisop/lab1/libc\_puts

Breakpoint 1, 0x0804840b in main ()

(gdb) watch \$ebx==55 && (\$eax==0x01 || \$eax==0xfc)

Watchpoint 2: \$ebx==55 && (\$eax==0x01 || \$eax==0xfc)

(gdb) c

Continuing.

Hello, world!

Watchpoint 2: \$ebx==55 && (\$eax==0x01 || \$eax==0xfc)

Old value = 0

New value = 1

0xf7ea27d1 in \_exit () from /lib/i386-linux-gnu/libc.so.6

(gdb) where

0 0xf7ea27d1 in \_exit () from /lib/i386-linux-gnu/libc.so.6

1 0xf7e2094a in ?? () from /lib/i386-linux-gnu/libc.so.6

2 0xf7e209ef in exit () from /lib/i386-linux-gnu/libc.so.6

3 0xf7e0a643 in \_\_libc\_start\_main () from /lib/i386-linux-gnu/libc.so.6

4 0x08048331 in \_start ()

(gdb) info shared

From	To	Syms Read	Shared Object Library
------	----	-----------	-----------------------

0xf7fd9860	0xf7ff273d	Yes (*)	/lib/ld-linux.so.2
------------	------------	---------	--------------------

0xf7e09750	0xf7f351bd	Yes (*)	/lib/i386-linux-gnu/libc.so.6
------------	------------	---------	-------------------------------

(\*): Shared library is missing debugging information.

(gdb)

salida de cat /proc/pidof libc\_puts/maps

\$ cat /proc/`pidof libc\_puts`/maps

08048000-08049000 r-xp 00000000 08:05 540438

/home/vasectomio

08049000-0804a000 r-xp 00000000 08:05 540438

/home/vasectomio

0804a000-0804b000 rwxp 00001000 08:05 540438

/home/vasectomio

0804b000-0806c000 rwxp 00000000 00:00 0

[heap]

f7df2000-f7fa2000 r-xp 00000000 08:05 918417

/lib/i386-linux

f7fa2000-f7fa4000 r-xp 001af000 08:05 918417

/lib/i386-linux

f7fa4000-f7fa5000 rwxp 001b1000 08:05 918417

/lib/i386-linux



f7fa5000-f7fa8000	rwxp	00000000	00:00	0	
f7fd4000-f7fd6000	rwxp	00000000	00:00	0	
f7fd6000-f7fd8000	r--p	00000000	00:00	0	[vvar]
f7fd8000-f7fd9000	r-xp	00000000	00:00	0	[vdso]
f7fd9000-f7ffb000	r-xp	00000000	08:05	916570	/lib/i386-linux
f7ffb000-f7ffc000	rwxp	00000000	00:00	0	
f7ffc000-f7ffd000	r-xp	00022000	08:05	916570	/lib/i386-linux
f7ffd000-f7ffe000	rwxp	00023000	08:05	916570	/lib/i386-linux
ffffd000-ffffe000	rwxp	00000000	00:00	0	

si bien no coinciden exactamente los nombres de los archivos, parece que en el filesystem hay información de versiones que es abstraída al programa en ejecución, porque los módulos `libc` y `ld` están presentes en ambos listados, como bibliotecas dinámicas `.so`

**¿cómo cambiaría la expresión booleana si —hipotéticamente— `exit_group()` tomara el valor de salida como segundo parámetro?**

La expresión booleana no cambiaría ya que lo que se está mirando con el watchpoint es los valores que tienen los registros inmediatamente antes de cederle el control al kernel, y no la llamada al wrapper de `libc`.

## Stack frames y calling conventions

### x86-ebp

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

Se sobrescribe el 1 que había sido pasado a `write`, como ese valor no vuelve a ser necesitado, esto no tiene ninguna consecuencia.

**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`?**

A partir de la primer instrucción `mov` el código difiere, agregando instrucciones que restauran el estado del stack y los registros para retornar de una llamada a función. Se restaura `ebp`

```
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

```

**¿Qué ocurre con %ebp usando my\_exit()?**

Al usar `my_exit()`, se restaura `%ebp` ya que el compilador no tiene forma de saber que no se volverá de `my_exit()` y agrega un `return` implícito a `main`.

**Verificar que ocurre con %ebp al declarar my\_exit(), como noreturn**

Ahora `%ebp` no se restaura, al igual que varios otros registros, y no se incluye una instrucción `ret` en `main`, siendo la última instrucción de la función la llamada a `my_exit()`.

## x86-errno

**Compilar y ejecutar el siguiente programa; mostrar qué se imprime por pantalla, así como el valor de retorno en el intérprete de comandos `echo $?`. Explicar también qué es la variable `errno`, y el funcionamiento de la función `perror(3)`.**

el programa muestra por pantalla lo mismo que `perror`. la variable `errno` es una variable global que se usa para detallar errores mediante códigos estándar. Por ejemplo, un valor de 9 significa 'bad file descriptor'. la función `perror` imprime el string correspondiente al código de error guardado en `errno`, prependeado por un string que recibe como argumento.

**Correr los siguientes comandos y mostrar su salida**

```

$ make clean perror perror2 write2
rm -f perror2 *.o asmexe cexe
make: 'perror' is up to date.
cc -m32      -DUSE_WRITE2 perror.c write2.S -o perror2
cc -m32      write2.S      -o write2
~/sisop/lab1 $
$ ./perror; echo $?
Falló write: Bad file descriptor
9
~/sisop/lab1 $
$ ./perror2; echo $?
Falló write: Success
0
~/sisop/lab1 $

```

```
$ ./write2; echo $?  
0
```

**Estudiar el código de perror.c y write2.S y explicar cómo funciona la compilación condicional; explicar, en particular:**

**qué hace -DUSE\_WRITE2**

-DUSE\_WRITE2 define la constante de preprocesador DUSE\_WRITE2, lo que evita que se genere el main de write2.S  
Adicionalmente, redefine a write como write2 en perror.c.

**qué efecto tiene la instrucción xor en write2.S**

la instrucción xor de un elemento contra si mismo (en este caso el registro `eax`, lo deja en 0)

**qué imprime el programa perror2, y por qué**

pperror2 imprime Falló write: Success, porque si bien se llega a llamar a perror("Falló write"), `errno` no indica ningún error.

**qué significa el atributo *cdecl* asignado a write2**

*cdecl* es el nombre que tiene el conjunto de convenciones usadas para llamadas entre funciones en C (que registros son caller-saved, cuales callee-saved, pasaje de argumentos por la pila y valor de retorno en `%eax`)

**por qué ocurre un error al recompilar (make clean all) si se elimina la definición de main en write2.S**

Ocorre un error de linkeo:

```
$ make clean perror perror2 write2  
rm -f perror2 *.o asmexe cexe  
cc -m32      perror.c      -o perror  
cc -m32      -DUSE_WRITE2 perror.c write2.S -o perror2  
cc -m32      write2.S      -o write2  
/usr/lib/gcc/x86_64-linux-gnu/5/../../../../lib32/crt1.o: In function ;_start':  
(.text+0x18): undefined reference to 'main'  
collect2: error: ld returned 1 exit status  
<built-in>: recipe for target 'write2' failed  
make: *** [write2] Error 1  
X $ ~/sisop/lab1 $
```

ya que el `_start` definido en los startfiles de la biblioteca estandar asume que hay una función llamada `main` y la llama, pero dicha función no está definida.

### **write2.S**

```
#include <bits/errno.h>
#include <sys/syscall.h>

.globl write2
write2:
    push %ebp
    mov %esp, %ebp
    push %edx
    push %ecx
    push %ebx

    mov 16(%ebp), %edx
    mov 12(%ebp), %ecx
    mov 8(%ebp), %ebx
    mov $SYS_write, %eax
    int $0x80
    cmp $0, %eax
    jg okay
    neg %eax
    mov %eax, %ebx
    call __errno_location
    mov %ebx, (%eax)
    mov $-1, %eax
okay:
    pop %ebx
    pop %ecx
    pop %edx
    mov %ebp, %esp
    pop %ebp
    ret

#ifdef USE_WRITE2
.globl main
main:
    xor %eax, %eax
    ret
#endif
```

¿qué registros debe preservar `write2`, según la call convention `cdecl`?

Debe preservar `ebx`, `ebp`, `esp`, `edi` y `esi`.

¿cómo debería cambiar el código de `write2` si se declarase con atributo `stdcall`?

Habría que sacar de la pila los argumentos que se le pasaron a la función.

## x86-argv

`sys__argv.S`

```
#include <sys/syscall.h>
```

```
.globl _start
```

```
_start:
```

```
    push 8(%esp)
    call strlen
    mov %eax, %edx
    pop %ecx
    movb $('\n'),(%ecx,%edx)
    inc %edx //nota: nuestro el largo considerando el '\0'
    mov $SYS_write, %eax
    mov $0x01, %ebx
    int $0x80
    mov %edx,%ebx
    mov $SYS_exit,%eax
    int $0x80
```

¿qué hay en `4(%esp)`?

Está el equivalente a `argv[0]`, el nombre del ejecutable.

`libc__argv.S`

```
.globl main
```

```
main:
```

```
    push %ebp
    mov %esp,%ebp

    mov 12(%ebp),%eax
    add $4,%eax
    mov (%eax),%eax
    mov %eax,%ebx
    push %eax
    call strlen
```

```

    movb $('\n'),(%ebx,%eax)
    inc %eax //vuelvo a considerar el largo con el '\0'
    push %eax
    push %ebx
    push $1
    call write

    mov %ebp,%esp
    pop %ebp
    ret

```

**¿qué hay en (%esp)?**

%esp, apenas se entra a main, apunta al frame pointer de la función que la llamó, y es 0x00000000, ya que no hay un frame superior a ese.

**libc\_argv2.S**

```

.globl main
main:
    push %ebp
    mov %esp,%ebp

    mov 12(%ebp),%eax
    mov %eax,%edi
    mov $0,%ebx
loop:
    push (%edi,%ebx,4)
    call puts
    inc %ebx
    cmp %ebx,8(%ebp)
    jg loop
    mov %ebx,%eax

    mov %ebp,%esp
    pop %ebp
    ret

```

**¿cuántas llamadas al sistema se producen realmente?**

Se hacen un total de 3 llamadas a write, una por cada puts.

```

(gdb) b write
Breakpoint 3 at 0xf7ec6b60
(gdb) run
Starting program: /home/vasectomio/sisop/lab1/libc_argv2 arte azucar

```

```
Breakpoint 3, 0xf7ec6b60 in write () from /lib/i386-linux-gnu/libc.so.6
(gdb) c
Continuing.
/home/vasectomio/sisop/lab1/libc_argv2
```

```
Breakpoint 3, 0xf7ec6b60 in write () from /lib/i386-linux-gnu/libc.so.6
(gdb) c
Continuing.
arte
```

```
Breakpoint 3, 0xf7ec6b60 in write () from /lib/i386-linux-gnu/libc.so.6
(gdb) c
Continuing.
azucar
[Inferior 1 (process 23256) exited with code 03]
(gdb)
```

Sin embargo, se puede ver que se hacen muchas más *syscalls*

```
(gdb) info breakpoints
Num      Type           Disp Enb Address      What
4        catchpoint     keep y          syscall "<any syscall>"
        catchpoint already hit 57 times
```

```
(gdb) ignore 4 1000
Will ignore next 1000 crossings of breakpoint 4.
```

```
(gdb) info breakpoints
Num      Type           Disp Enb Address      What
4        catchpoint     keep y          syscall "<any syscall>"
        catchpoint already hit 57 times
        ignore next 1000 hits
```

```
(gdb) run
Starting program: /home/vasectomio/sisop/lab1/libc_argv2 arte azucar
/home/vasectomio/sisop/lab1/libc_argv2
arte
azucar
```

```
[Inferior 1 (process 23511) exited with code 03]
(gdb) info breakpoints
Num      Type           Disp Enb Address      What
4        catchpoint     keep y          syscall "<any syscall>"
        catchpoint already hit 57 times
        ignore next 943 hits
```

```
(gdb)
```

x86-frames

x86-dwarf

## Creación de stacks en el kernel

kern1-stack

kern1-cmdline

kern1-meminfo