FACULTAD
DE INGENIERIA
Universidad de Buenos Aires

Lab x86: Assembler y Call Conventions

Sistemas Operativos (75.08 / 95.03)

Apellidos: Escobar Benítez
Nombres: María Soledad
Padrón: 97877

Resolución

Llamadas a biblioteca y llamadas al sistema

- **Ej: x86-write**

  - ¿Por qué se le resta 1 al resultado de sizeof?
    - Se le resta 1 porque los strings en C tienen un caracter adicional '\0', el mismo se utiliza para marcar el final del string, por lo que si se cuenta el espacio total que ocupa el string con un sizeof, se cuenta también el caracter adicional, por lo tanto se le resta 1 para obtener la cantidad de caracteres del mensaje sin tomar en cuenta la marca de final del string, y así imprimir correctamente por pantalla el mensaje que se desea mostrar.

  - ¿Funcionaría el programa si se declarase msg como const char *msg = "...";? ¿Por qué?
    - La asignación const char msg[] = "mensaje" lo que hace es asignar espacio del stack para los 8 caracteres que construyen el string "mensaje\0", ya que añade automaticamente el cracter '\0', y lo copiará a ese espacio, por lo que se tiene es la dirección real del string
      Y la asignación const char *msg = "mensaje" representa a msj como un puntero que almacena la dirección del primer elemento del string. La diferencia con la asignación anterior es que el tamaño no necesita ser definido en este caso, dado que lo que tenemos es un puntero apuntando a la dirección del string.
      Si modifico esto en el programa no funciona de la misma manera, ya que con la definición original pide el tamaño del string para imprimirlo haciendo sizeof msg, en el caso de usar la segunda definición, al hacer el sizeof lo que obtendríamos sería el tamaño del puntero, que es 4, por lo que no se obtendría el mensaje correctamente.

  - Explicar el efecto del operador . en la línea .set len, . - msg.
    - El operador . Indica que es una variable que sólo tienen alcance local y no se incluyen en la tabla de símbolos del archivo de objeto.

  - 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.
    - *Definición de la función main():*

.globl main
main:
Las próximas 3 lineas corresponde a hacer push de los argumentos de la función write() en el stack.
push $len → *corresponde al sizeof msg –1*
push $msg → *corresponde al mensaje*
push $1 → *es el 1 que se le envía a la función write que indica que debe imprimirlo por pantalla*
En la próxima linea lo que se hace es el llamado a la función write para que la misma se ejecute.
call write
Las 4 lineas anteriores equivalen al llamado ***write(1, msg, sizeof msg – 1);*** *en el código .c original.*

push $7 → *hace push del argumento, de la función exit(), en el stack.*
call _exit → *Llamado a la función exit()*
Las dos líneas anteriores son las que equivalen al llamado de la función ***_exit(7)*** *en el código .c.*

La siguiente linea define el área de definición de datos
.data
Y luego se define el string msg en código ascii
msg:
.ascii "Hello, world!\n"
Esta ultima linea lo que hace es setear el valor de len en función del string msg
.set len, . – msg

- Mostrar un hex dump de la salida del programa en assembler. Se puede obtener con el comando od:
  - **$ ./libc_hello | od –t x1 –c**
    0000000 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
    0000016

- Cambiar la directiva .ascii por .asciz y mostrar el hex dump resultante con el nuevo código. ¿Qué está ocurriendo?
  - **6$ ./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
    Lo que sucede es que se muestra además el '\0' que se encuentra al final del string. Al declararlo como .asciz implica que el string termina en '\0'.

- **Ej: x86–call**

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

```
(gdb) x/6i $pc
=> 0x8048456 <main>:    push   $0xf
   0x804845b <main+5>:  push   $0x804a020
   0x8048460 <main+10>: push   $0x1
   0x8048462 <main+12>: call   0x8048320 <write@plt>
   0x8048467 <main+17>: push   $0x7
   0x8048469 <main+19>: call   0x8048300 <_exit@plt>
```

- 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.

```
(gdb) stepi
10              push $msg
(gdb) stepi
11              push $1
(gdb) stepi
14              call write
(gdb) x/4xw $esp
0xffffce40: 0x00000001  0x0804a020  0x0000000f  0xf7df5e81
```

Los primeros 3 elementos corresponden a los argumentos de la función write, 1 → 0x00000001, msg → 0x0804a020 (dirección de memoria en la que se encuentra el mensaje) y strlen → 0x0000000f (15).
Y el último valor corresponde al valor de retorno de la función.

```
(gdb) si
0x08048320 in write@plt ()
(gdb) x/4xw $sp
0xffffce3c: 0x08048467  0x00000001  0x0804a020  0x0000000f
```

El nuevo valor → 0x08048467 corresponde a la dirección en la que se debe continuar luego de la ejecución de write.
Los otros valores son los 3 definidos anteriormente.

- Sustituir la instrucción call write por jmp write, y añadir el código y preparaciones necesarias para que el programa siga funcionando. Código modificado:

```
.globl main

main:

        push $strlen
        push $msg
        push $1

        call posicion_retorno
```

```
push $7
call _exit

.data
posicion_retorno:
        jmp write
        ret
msg:
        .asciz "Hello, world!\n"

.set strlen, . - msg
```

- **Ej: x86-libc**
  - Compilar y ejecutar el archivo completo int80_hi.S. Mostrar la salida de nm --undefined para este nuevo binario.
    - **$ nm --undefined int80_hi**
      w __gmon_start__
      U __libc_start_main@@GLIBC_2.0
  - Escribir una versión modificada llamada int80_strlen.S en la que, de nuevo eliminando la directiva .set len, se calcule la longitud del mensaje usando directamente strlen(3). Mostrar la salida de nm --undefined para este nuevo binario.
    - **$ nm --undefined int80_strlen**
    - **w __gmon_start__**
    - **U __libc_start_main@@GLIBC_2.0**
    - **U strlen@@GLIBC_2.0 =>** símbolo correspondiente a la llamada a la biblioteca de C para utilizar la función strlen.
  - ¿Qué significa que un registro sea callee-saved en lugar de caller-saved?
    - Un registros caller-saved es un registro de propósito general orientado a almacenar información temporal, volátil, que puede ser sobrescrita por cualquier subrutina por lo que se lo debe insertar manualmente a la pila si se desea poder restaurar sus valores luego de una subrutina.
      En cambio los registros callee-saved se utilizan para mantener valores de larga duración, no volátiles, que deben conservarse en todas las llamadas por lo tanto cuando una persona realiza una llamada de procedimiento, puede esperar que esos registros conserven el mismo valor una vez que el destinatario vuelva.
  - En x86 ¿de qué tipo, caller-saved o callee-saved, es cada registro según la convención de llamadas de GCC?
    - Los registros caller-saved son EAX, ECX y EDX.
    - Los registros ESP y EBP son callee-saved.

- Copiar int80_strlen.S a un nuevo archivo sys_strlen.S, renombrando main a _start en el proceso. Mostrar la salida de nm --undefined para este nuevo binario, y describir brevemente las diferencias con los casos anteriores.
  - **$ nm --undefined sys_strlen**
  - **U strlen@@GLIBC_2.0**

    Ejecutar el comando sólo muestra el símbolo correspondiente a strlen la diferencia es que en los casos anteriores se mostraban los símbolos referentes al inicio del programa, tanto de .global main como main que está referenciando a la biblioteca estándar de C. Al reemplazarlo por _start se está indicando el inicio de la ejecución del programa directamente sin utilizar ninguna biblioteca.
-

• Ej: x86-ret

- Un main estándar devuelve int, y comúnmente se usa return, no exit, para devolver un código de error. Los "start files" de libc y su definición de _start se encargan de propagar ese valor de retorno al syscall exit.
  Se pide ahora modificar int80_hi.S para que, en lugar de invocar a a _exit(), la ejecución finalice sencillamente con una instrucción ret. ¿Cómo se pasa en este caso el valor de retorno?
  - Modifico int80_hi.S utilizando un rotulo llamado fin y hago call fin, lo que sucede internamente es que call guarda en el stack la dirección de la siguiente instrucción y ejecuta un salto al rotulo fin. En el mismo muevo el valor de retorno 7 al %ebx y luego utilizo ret que retorna a la dirección que fue almacenada en el stack por la instrucción call.

- Se pide mostrar, usando un catchpoint, una sesión de GDB el momento en que el binario libc_puts realiza la llamada a exit con int $0x80 o sysenter, y dónde reside dicha instrucción.
  - **$ gdb -q ./libc_puts**
  - Leyendo símbolos desde ./libc_puts...hecho.
  - (gdb) catch syscall exit_group
  - Punto de captura 1 (syscall 'exit_group' [252])
  - (gdb) r
  - Starting program: /home/sol/Sistemas_Operativos/kernx86/libc_puts
  - Hello, world!
  -
  -
  - Catchpoint 1 (call to syscall exit_group), 0xf7fd5059 in __kernel_vsyscall ()
  - (gdb) disas
  - Dump of assembler code for function __kernel_vsyscall:
  - 0xf7fd5050 <+0>: push %ecx
  - 0xf7fd5051 <+1>: push %edx
  - 0xf7fd5052 <+2>: push %ebp
  - 0xf7fd5053 <+3>: mov %esp,%ebp

```
   0xf7fd5055 <+5>:     sysenter
   0xf7fd5057 <+7>:     int    $0x80
=> 0xf7fd5059 <+9>:     pop    %ebp
   0xf7fd505a <+10>:    pop    %edx
   0xf7fd505b <+11>:    pop    %ecx
   0xf7fd505c <+12>:    ret
End of assembler dump.
(gdb) backtrace
#0  0xf7fd5059 in __kernel_vsyscall ()
#1  0xf7e9b345 in _exit () from /lib32/libc.so.6
(gdb) info shared
From        To          Syms Read   Shared Object Library
0xf7fd6ab0  0xf7ff177b  Yes (*)     /lib/ld-linux.so.2
0xf7df5610  0xf7f3fbb6  Yes (*)     /lib32/libc.so.6
(*): A la biblioteca compartida le falta la información de
depuración.
(gdb)
```

Stack frames y calling conventions

- **Ej: x86-ebp**

```
$ gdb -batch -ex 'disas/s main' ./hello
Dump of assembler code for function main:
hello.c:
6     int main(void) {
   0x08048456 <+0>:     lea    0x4(%esp),%ecx
   0x0804845a <+4>:     and    $0xfffffff0,%esp
   0x0804845d <+7>:     pushl  -0x4(%ecx)
   0x08048460 <+10>:    push   %ebp
   0x08048461 <+11>:    mov    %esp,%ebp
   0x08048463 <+13>:    push   %edi
   0x08048464 <+14>:    push   %ebx
   0x08048465 <+15>:    push   %ecx
   0x08048466 <+16>:    sub    $0x10,%esp

7         write(1, msg, strlen(msg));
   0x08048469 <+19>:    mov    0x804a020,%edx
   0x0804846f <+25>:    mov    $0xffffffff,%ecx
   0x08048474 <+30>:    mov    $0x0,%eax
   0x08048479 <+35>:    mov    %edx,%edi
   0x0804847b <+37>:    repnz scas %es:(%edi),%al
   0x0804847d <+39>:    mov    %ecx,%eax
   0x0804847f <+41>:    not    %eax
   0x08048481 <+43>:    sub    $0x1,%eax
```

```
0x08048484 <+46>:    push   %eax
0x08048485 <+47>:    push   %edx
0x08048486 <+48>:    push   $0x1
0x08048488 <+50>:    call   0x8048320 <write@plt>

8         _exit(7);
0x0804848d <+55>:    movl   $0x7,(%esp)
0x08048494 <+62>:    call   0x8048300 <_exit@plt>
End of assembler dump.

$ gdb -batch -ex 'disas main' libc_hello
Dump of assembler code for function main:
0x08048496 <+0>:     call   0x8048340 <strlen@plt>
0x0804849b <+5>:     push   $0x804a02a
0x080484a0 <+10>:    push   $0x1
0x080484a2 <+12>:    call   0x804a024
0x080484a7 <+17>:    push   $0x7
0x080484a9 <+19>:    call   0x8048330 <_exit@plt>
0x080484ae <+24>:    xchg   %ax,%ax
End of assembler dump.
```

- ¿Qué valor sobreescribió GCC cuando usó mov $7, (%esp) en lugar de push $7 para la llamada a _exit? ¿Tiene esto alguna consecuencia?
  - Lo que sobrescribe es el último elemento de la pila, no tiene consecuencias pues lo que se sobrescribe es uno de los parámetros de la función write que ya fue utilizado, más específicamente el valor 1 correspondiente al primer argumento de la función.

- 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?

```
$ gdb -batch -ex 'disas/s main' ./hello
Dump of assembler code for function main:
hello.c:
6     int main(void) {
0x08048456 <+0>:     lea    0x4(%esp),%ecx
0x0804845a <+4>:     and    $0xfffffff0,%esp
0x0804845d <+7>:     pushl  -0x4(%ecx)
0x08048460 <+10>:    push   %ebp
0x08048461 <+11>:    mov    %esp,%ebp
0x08048463 <+13>:    push   %edi
0x08048464 <+14>:    push   %ebx
0x08048465 <+15>:    push   %ecx
0x08048466 <+16>:    sub    $0x10,%esp

7         write(1, msg, strlen(msg));
0x08048469 <+19>:    mov    0x804a020,%edx
0x0804846f <+25>:    mov    $0xffffffff,%ecx
0x08048474 <+30>:    mov    $0x0,%eax
```

```
   0x08048479 <+35>:    mov    %edx,%edi
   0x0804847b <+37>:    repnz scas %es:(%edi),%al
   0x0804847d <+39>:    mov    %ecx,%eax
   0x0804847f <+41>:    not    %eax
   0x08048481 <+43>:    sub    $0x1,%eax
   0x08048484 <+46>:    push   %eax
   0x08048485 <+47>:    push   %edx
   0x08048486 <+48>:    push   $0x1
   0x08048488 <+50>:    call   0x8048320 <write@plt>
   
8          return 7;
9      }
   0x0804848d <+55>:    mov    $0x7,%eax
   0x08048492 <+60>:    lea    -0xc(%ebp),%esp
   0x08048495 <+63>:    pop    %ecx
   0x08048496 <+64>:    pop    %ebx
   0x08048497 <+65>:    pop    %edi
   0x08048498 <+66>:    pop    %ebp
   0x08048499 <+67>:    lea    -0x4(%ecx),%esp
   0x0804849c <+70>:    ret
End of assembler dump.
```

Lo que cambia es la sección que antes hacía la llamada a _exit, podemos ver como se hace pop de los argumentos que se pushearon anteriormente y se restaura el valor de %ebp pues el mismo se encuentra antes de los argumentos, tengo tres pops a %ecx, %ebx y %edi que corresponden a los argumentos de write, y luego el último pop a %ebp restaura el valor anterior al push de los argumentos.

- Crear un archivo llamado lib/exit.c. ¿Qué ocurre con %ebp?

```
$ gdb -batch -ex 'disas/s main' ./hello
Dump of assembler code for function main:
hello.c:
6      int main(void) {
   0x08048456 <+0>:     lea    0x4(%esp),%ecx
   0x0804845a <+4>:     and    $0xfffffff0,%esp
   0x0804845d <+7>:     pushl  -0x4(%ecx)
   0x08048460 <+10>:    push   %ebp
   0x08048461 <+11>:    mov    %esp,%ebp
   0x08048463 <+13>:    push   %edi
   0x08048464 <+14>:    push   %ebx
   0x08048465 <+15>:    push   %ecx
   0x08048466 <+16>:    sub    $0x10,%esp
   
7          write(1, msg, strlen(msg));
   0x08048469 <+19>:    mov    0x804a020,%edx
   0x0804846f <+25>:    mov    $0xffffffff,%ecx
   0x08048474 <+30>:    mov    $0x0,%eax
   0x08048479 <+35>:    mov    %edx,%edi
   0x0804847b <+37>:    repnz scas %es:(%edi),%al
   0x0804847d <+39>:    mov    %ecx,%eax
```

```
   0x0804847f <+41>:    not    %eax
   0x08048481 <+43>:    sub    $0x1,%eax
   0x08048484 <+46>:    push   %eax
   0x08048485 <+47>:    push   %edx
   0x08048486 <+48>:    push   $0x1
   0x08048488 <+50>:    call   0x8048320 <write@plt>

8         my_exit(7);
   0x0804848d <+55>:    movl   $0x7,(%esp)
   0x08048494 <+62>:    call   0x80484a9 <my_exit>

9     }
   0x08048499 <+67>:    mov    $0x0,%eax
   0x0804849e <+72>:    lea    -0xc(%ebp),%esp
   0x080484a1 <+75>:    pop    %ecx
   0x080484a2 <+76>:    pop    %ebx
   0x080484a3 <+77>:    pop    %edi
   0x080484a4 <+78>:    pop    %ebp
   0x080484a5 <+79>:    lea    -0x4(%ecx),%esp
   0x080484a8 <+82>:    ret
End of assembler dump.
```

Vemos que aunque se llame a _exit desde my_exit, el comportamiento luego de esa llamada es igual al de usar return, el valor de ebp se restaura.

- En hello.c, cambiar la declaración de my_exit a:
  `extern void __attribute__((noreturn)) my_exit(int status);`
  y verificar qué ocurre con %ebp, relacionándolo con el significado del atributo noreturn.

```
$ gdb -batch -ex 'disas/s main' ./hello
Dump of assembler code for function main:
hello.c:
6     int main(void) {
   0x08048456 <+0>:     lea    0x4(%esp),%ecx
   0x0804845a <+4>:     and    $0xfffffff0,%esp
   0x0804845d <+7>:     pushl  -0x4(%ecx)
   0x08048460 <+10>:    push   %ebp
   0x08048461 <+11>:    mov    %esp,%ebp
   0x08048463 <+13>:    push   %edi
   0x08048464 <+14>:    push   %ebx
   0x08048465 <+15>:    push   %ecx
   0x08048466 <+16>:    sub    $0x10,%esp

7         write(1, msg, strlen(msg));
   0x08048469 <+19>:    mov    0x804a020,%edx
   0x0804846f <+25>:    mov    $0xffffffff,%ecx
   0x08048474 <+30>:    mov    $0x0,%eax
   0x08048479 <+35>:    mov    %edx,%edi
   0x0804847b <+37>:    repnz scas %es:(%edi),%al
   0x0804847d <+39>:    mov    %ecx,%eax
```

```
0x0804847f <+41>:    not    %eax
0x08048481 <+43>:    sub    $0x1,%eax
0x08048484 <+46>:    push   %eax
0x08048485 <+47>:    push   %edx
0x08048486 <+48>:    push   $0x1
0x08048488 <+50>:    call   0x8048320 <write@plt>

8         my_exit(7);
0x0804848d <+55>:    movl   $0x7,(%esp)
0x08048494 <+62>:    call   0x8048499 <my_exit>
End of assembler dump
```

Se puede ver ahora como ya no se llama a return, y tenemos algo muy parecido al código inicial, donde no se hacía pop de los elementos de la pila ni se restaura el valor de %ebp. Esto tiene que ver con la definición de la función my_exit() usando el atributo **noreturn** se le indica al compilador que la función no tiene retorno entonces el mismo puede aplicar optimizaciones sin preocuparse por si la función retornará algo alguna vez.

- **Ej: x86-frames**
  - Responder, en términos del frame pointer %ebp de una función f:
    - ¿dónde se encuentra (de haberlo) el primer argumento de f?
      El primer argumento de f se encuentra en la dirección del frame pointer + 8 bytes.
    - ¿dónde se encuentra la dirección a la que retorna f cuando ejecute ret?
      La dirección de retorno de la función f se encuentra 4 bytes arriba de la dirección del frame pointer.
    - ¿dónde se encuentra el valor de %ebp de la función anterior, que invocó a f?
      El valor de %ebp de la función anterior se encuentra en el puntero almacenado en el %ebp de la función actual.
    - ¿dónde se encuentra la dirección a la que retornará la función que invocó a f?
      La dirección de retorno de la función que invocó a f se encuentra 4 posiciones sobre la dirección a la que apunta el puntero almacenado en %ebp.
  - Se pide ahora escribir una función **void backtrace();** que obtenga, usando __builtin_frame_address(0), el frame pointer actual, e imprima la secuencia de marcos anidados en el formato que se indica a continuación:
    - **#numfrm [FP] ADDR (ARG1 ARG2 ARG3)**

    donde para cada frame FP es el frame pointer (registro %ebp), ADDR es el punto de retorno a la función, y ARGS sus tres primeros argumentos.
    - **La siguiente resolución de backtrace no funciona correctamente:**
      - void backtrace(void) {

```
        uint8_t numfrm = 1;
        int *FP = __builtin_frame_address(0);

        while(*FP) {
                unsigned int *ADDR = &(*FP) + 1;

              printf("#%u [%p] %p (%p %p %p)\n",
                    numfrm++,
                    (void *) *FP,
                    (void *) *ADDR,
                    (void *) *(ADDR + 1),
                    (void *) *(ADDR + 2),
                    (void *) *(ADDR + 3)
              );
              FP = (void*) * FP;
        }
  }
```

- sesión de GDB en la que se muestre la equivalencia entre el comando bt de GDB y el código implementado; en particular, se debe incluir:
  - la salida del comando bt al entrar en la función backtrace
  - la salida del programa al ejecutarse la función backtrace (el número de frames y sus direcciones de retorno deberían coincidir con la salida de bt)
  - usando los comandos de selección de frames, y antes de salir de la función backtrace, el valor de %ebp en cada marco de ejecución detectado por GDB (valores que también deberían coincidir).

```
$ gdb -q ./backtrace
Leyendo símbolos desde ./backtrace...hecho.
(gdb) b backtrace
Punto de interrupción 1 at 0x8048506: file backtrace.c, line 5.
(gdb) r
Starting program: /home/sol/Sistemas_Operativos/kernx86/backtrace

Breakpoint 1, backtrace () at backtrace.c:5
5     void backtrace(void) {
(gdb) list
1     #include <stdint.h>
2     #include <stdio.h>
3     #include <unistd.h>
4
5     void backtrace(void) {
6           uint8_t numfrm = 1;
7           int *FP = __builtin_frame_address(0);
8
9           while(*FP) {
10                unsigned int *ADDR = &(*FP) + 1;
(gdb) list
11
12                printf("#%u [%p] %p (%p %p %p)\n",
13                      numfrm++,
```

```
14                      (void *) *FP,
15                      (void *) *ADDR,
16                      (void *) *(ADDR + 1),
17                      (void *) *(ADDR + 2),
18                      (void *) *(ADDR + 3)
19              );
20              FP = (void*) * FP;
(gdb) list
21          }
22      }
23
24      void my_write(int fd, const void *msg, size_t count) {
25          backtrace();
26          fprintf(stderr, "=> write(%d, %p, %zu)\n", fd, msg,
count);
27          write(fd, msg, count);
28      }
29
30      void recurse(int level) {
(gdb) until 22
#1 [0xffffcd38] 0x8048564 (0xffffcd74 0xffffcd70 0x3)
#2 [0xffffcd58] 0x80485b0 (0x2 0x80486af 0xf)
#3 [0xffffcd78] 0x80485c1 ((nil) (nil) (nil))
#4 [0xffffcd98] 0x80485c1 (0x1 0xf63d4e2e 0xf7ffdaf8)
#5 [0xffffcdb8] 0x80485c1 (0x2 0x1 0xf7fcf410)
#6 [0xffffcdd8] 0x80485c1 (0x3 0xc30000 (nil))
#7 [0xffffcdf8] 0x80485c1 (0x4 0xffffd0bc 0xf7e0d049)
#8 [0xffffce18] 0x80485d3 (0x5 (nil) 0xffffcedc)
#9 [0xffffce28] 0x80485ee (0xf7fe59b0 0xffffce40 (nil))
backtrace () at backtrace.c:22
22      }
(gdb) up
#1  0x08048564 in my_write (fd=2, msg=0x80486af, count=15) at
backtrace.c:25
25          backtrace();
(gdb) p/x $ebp
$1 = 0xffffcd38
(gdb) up
#2  0x080485b0 in recurse (level=0) at backtrace.c:34
34              my_write(2, "Hello, world!\n", 15);
(gdb) p/x $ebp
$2 = 0xffffcd58
(gdb) up
#3  0x080485c1 in recurse (level=1) at backtrace.c:32
32              recurse(level - 1);
(gdb) p/x $ebp
$3 = 0xffffcd78
(gdb) up
#4  0x080485c1 in recurse (level=2) at backtrace.c:32
32              recurse(level - 1);
```

```
(gdb) p/x $ebp
$4 = 0xffffcd98
(gdb) up
#5  0x080485c1 in recurse (level=3) at backtrace.c:32
32            recurse(level - 1);
(gdb) p/x $ebp
$5 = 0xffffcdb8
(gdb) up
#6  0x080485c1 in recurse (level=4) at backtrace.c:32
32            recurse(level - 1);
(gdb) p/x $ebp
$6 = 0xffffcdd8
(gdb) up
#7  0x080485c1 in recurse (level=5) at backtrace.c:32
32            recurse(level - 1);
(gdb) p/x $ebp
$7 = 0xffffcdf8
(gdb) up
#8  0x080485d3 in start_call_tree () at backtrace.c:38
38        recurse(5);
(gdb) p/x $ebp
$8 = 0xffffce18
(gdb) up
#9  0x080485ee in main () at backtrace.c:42
42        start_call_tree();
(gdb) p/x $ebp
$9 = 0xffffce28
(gdb) up
Initial frame selected; you cannot go up.
(gdb) frame 0
#0  backtrace () at backtrace.c:22
22    }
(gdb) c
Continuando.
=> write(2, 0x80486af, 15)
Hello, world!
[Inferior 1 (process 11177) exited normally]
(gdb)
```