

Inspección del stack en llamadas a ensamblador (TP GINI)

Agustín

Grupo SdeC 2025

18 de abril de 2025

1. Introducción

Este informe demuestra, mediante **GDB**, cómo inspeccionar el **stack** durante la llamada desde un programa *C* de 32 bits a una rutina en ensamblador (*cdecl*). El caso de prueba convierte un valor flotante (índice GINI) a entero (*floor*) y le suma 1.

2. Entorno de trabajo

- **Arquitectura:** x86 (32 bits).
- **Compiladores:** GCC (-m32), NASM (-f elf32).
- **Depurador:** GDB.
- **Entrada:** un float leído desde `stdin`.
- **Función ASM:** `calculate_gini_int`.

3. Inspección del stack

3.1. Estado previo a la llamada

La fig. 1 muestra el estado del stack y del registro `esp` antes de ejecutar la llamada a ensamblador.

```
(gdb) info registers esp ebp
esp      0xffffcb70      0xffffcb70
ebp      0xffffcb88      0xffffcb88
(gdb) █
```

(a) Registros `esp` y `ebp`.

```
(gdb) x/16wx $esp
0xffffcb70: 0x00000000      0x00000000      0x422f3333      0xfd99f900
0xffffcb80: 0xffffc000      0xf7f9be34      0x00000000      0xf7d97c75
0xffffcb90: 0x00000000      0xffffcc54      0xf7db0b59      0xf7d97c75
0xffffcba0: 0x00000001      0xffffcc54      0xffffcc5c      0xffffcbcb
(gdb) print gini
$1 = 43.7999992
(gdb) █
```

(b) Valor flotante en memoria (`0x422f3333` \approx 43.8).

Figura 1: Estado del programa antes de la llamada a ASM.

3.2. Ejecución en la rutina ASM

La fig. 2 resalta las instrucciones críticas `fld`, `fistp`, y `add eax, 1` dentro de la función.

```

B+ calc.asm n(void) {
1 ; gini_calc.asm - Convierte float a int (floor) y suma 1
> 12 ; Ensamblar: nasm -f elf32 gini_calc.asm -g
3
4 section .text
5 global calculate_gini_int ; firma: int calculate_gini_int(float)
6
7 ; int calculate_gini_int(float x)(gini);
8 calculate_gini_int:
> 9 push ebp
10 mov ebp, esp
1
12 sub esp, 4 ; espacio temporal para el entero
13 fld dword [ebp+8] ; Cuando se cargan argumentos float el input
14 fistp dword [esp] ; convertir a int (trunc) y guardar
15 mov eax, [esp] ; mover a eax (valor de retorno)
16 add eax, 1
17 leave
18 ret

```

multi-thre Thread 0xf7fc2500 ((src) In: main
Breakpoint 1 at 0x1090: file gini_processor.c:8
[Thread debugging using libthread_db enabled] llamada a ASM:
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
gdb
Breakpoint 1, main () at gini_processor.c:8
(gdb) step
(gdb) step
43.8
(gdb) step
calculate_gini_int () at gini_calc.asm:9
(gdb)

Figura 2: Vista paso a paso dentro de `calculate_gini_int`.

3.3. Retorno a C

Tras el `ret`, el resultado entero reside en `EAX` y se copia a `result` (fig. 3).

(a) EAX = 0x2D (45).

```

(gdb) info registers eax esp ebp
eax      0x2c      44
esp      0xffffcb54  0xffffcb54
ebp      0xffffcb58  0xffffcb58
(gdb) x/16xw $esp
0xffffcb54: 0x0000002c  0xffffcb88  0x56556111  0x422f3333
0xffffcb64: 0xffffcb78  0x00000000  0x00000000  0x00000000
0xffffcb74: 0x00000000  0x422f3333  0xf99f9000  0xffffc000
0xffffcb84: 0xf7f9be34  0x00000000  0xf7d97c75  0x00000000

```

(b) Variable `result` con el mismo valor.

```

B+ processor.c n(void) {
8 int main(void) {
9 float gini;
10 if (scanf("%f", &gini) != 1) {
11 fprintf(stderr, "Error: no se pudo leer el float desde stdin\n");
12 return 1;
13 }
14
15 int result = calculate_gini_int(gini);
16 printf("%d\n", result);
17 return 0;
18 }

```

multi-thre Thread 0xf7fc2500 ((src) In: main
Breakpoint 1 at 0x1090: file gini_processor.c:8
(gdb) info registers eax esp ebp
eax 0x2d 45
esp 0xffffcb60 0xffffcb60
ebp 0xffffcb88 0xffffcb88
(gdb) x/16xw \$esp
0xffffcb60: 0x422f3333 0xffffcb78 0x00000000 0x00000000
0xffffcb70: 0x00000000 0x00000000 0x422f3333 0xf99f9000
0xffffcb80: 0xffffc000 0xf7f9be34 0x00000000 0xf7d97c75
0xffffcb90: 0x00000000 0xffffc000 0xf7d97c75 0x00000000
(gdb)

Figura 3: Estado del programa después de la llamada.

Para mayor detalle, fig. 4 muestra la dirección del `float` en el stack y corrobora el valor devuelto.

```
(gdb) x/wx $esp
0xffffcb60: 0x422f3333
(gdb) info registers eax
eax          0x2d          45
(gdb) print result
$2 = 45
(gdb)
```

Figura 4: Dirección del argumento y verificación de EAX.

4. Diagrama de secuencia completo

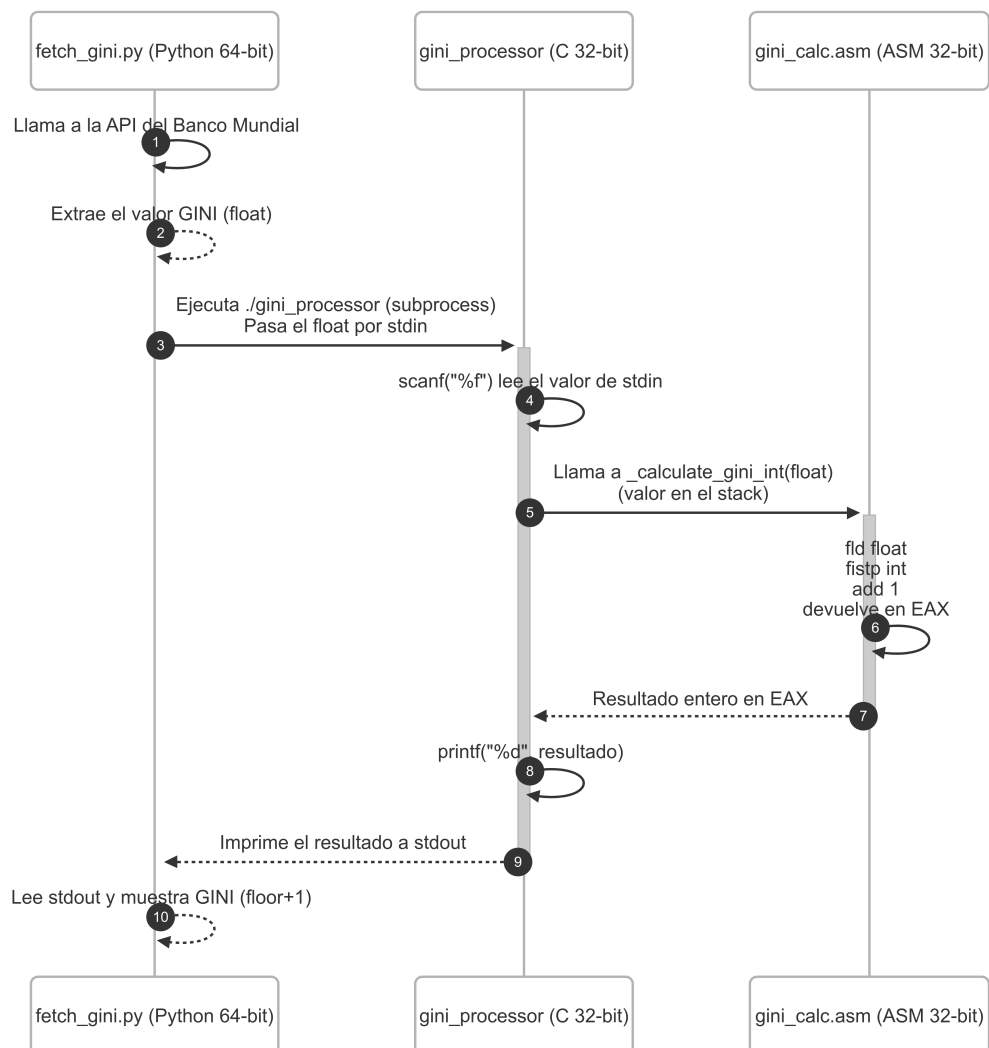


Figura 5: Flujo end-to-end desde Python hasta la rutina ASM.

5. Conclusiones

GDB permitió rastrear el flujo de datos entre C y ensamblador: *argumento en el stack* \rightarrow *FPU* \rightarrow *retorno en EAX*. La rutina en ASM cumple con la convención *cdecl* y la variable `result` refleja el valor correcto (GINI truncado + 1).