UNIVERSIDAD NACIONAL DE CÓRDOBA

FACULTAD DE CIENCIAS EXACTAS, FÍSICAS Y NATURALES

CÁTEDRA DE SISTEMAS DE COMPUTACIÓN - 2025

TRABAJO PRÁCTICO N°2

“Stack Frame & API Rest”

GRUPO

Fork Around & Checkout

PROFESORES

Jorge, Javier & Solinas, Miguel

ALUMNOS

Nombre	DNI
Ávalos, Sofia	44195495
Ramirez, Valentin José	43700362
Stefanovic, Fernando	44090692

1. Introducción

1.1. Contexto del Problema

Los sistemas informáticos modernos, desde aplicaciones embebidas hasta complejos servicios en la nube, suelen construirse utilizando arquitecturas de software por capas. Esta estratificación permite separar responsabilidades, facilitando el desarrollo, el mantenimiento y la evolución de los sistemas. En las capas superiores, encontramos lenguajes de alto nivel como Python o C, que ofrecen abstracciones poderosas y son más "amigables" con el programador, permitiendo un desarrollo rápido y eficiente de la lógica de negocio y las interfaces de usuario.

Sin embargo, en la base de toda computación se encuentra el hardware. Para interactuar directamente con él, optimizar tareas críticas en términos de rendimiento o acceder a funcionalidades específicas de la arquitectura, a menudo es necesario descender a capas más bajas. Inmediatamente por encima del hardware reside el lenguaje ensamblador (Assembler), un lenguaje de bajo nivel que proporciona una representación simbólica de las instrucciones máquina nativas del procesador. Aunque más complejo de manejar que los lenguajes de alto nivel, ofrece un control granular sobre los recursos del sistema.

La necesidad de que las capas superiores interactúen con las inferiores es frecuente. Los lenguajes de alto nivel deben poder invocar rutinas escritas en ensamblador para realizar operaciones específicas. Esta comunicación no es trivial y se basa en **convenciones de llamada** (calling conventions) bien definidas. Estas convenciones son conjuntos de reglas que dictan cómo se pasan los parámetros entre funciones (por ejemplo, a través de registros o de la pila de memoria - stack), cómo se devuelven los valores y quién es responsable de gestionar el entorno de la llamada (como preservar ciertos registros o limpiar la pila). Comprender estas convenciones es fundamental no solo para la interoperabilidad entre lenguajes, sino también para áreas avanzadas como el desarrollo de sistemas operativos, la ingeniería inversa, el análisis de seguridad y la optimización de rendimiento.

1.2. ObjetIvos del Trabajo Práctico

El objetivo principal de este Trabajo Práctico (TP#2) es diseñar, implementar y analizar un sistema simple pero representativo que demuestre la interacción entre diferentes capas de software, involucrando lenguajes de alto nivel (Python y C) y un lenguaje de bajo nivel (Ensamblador x86).

1.3. Estructura del Informe

Este documento se organiza de la siguiente manera: La **Sección 2** detalla el entorno de trabajo configurado y las herramientas utilizadas. La **Sección 3** presenta el diseño de la solución, describiendo la arquitectura general y el enfoque específico para cada capa (Python, C, Ensamblador) y la interfaz entre ellas. La **Sección 4** describe la implementación

concreta mediante el uso de herramientas visuales como diagramas. La **Sección 5** se dedica a las pruebas y, de manera crucial, al análisis detallado de la ejecución y el estado de la pila utilizando GDB. Por último, la **Sección 6** manifiesta conclusiones y observaciones al realizar la experiencia.

2. Entorno de Trabajo y Herramientas

El desarrollo y las pruebas de este trabajo práctico se realizaron en un entorno Linux, específicamente distribuciones basadas en Debian/Ubuntu (64 bits). La selección de herramientas se centró en cumplir los requisitos del proyecto, permitiendo la compilación cruzada y la depuración multinivel.

- **Sistema Operativo:** Linux (basado en Debian/Ubuntu 64 bits).
- **Arquitectura:**
    - Host (Desarrollo/Ejecución Python): x86-64 (64 bits).
    - Objetivo (C/Ensamblador): x86 (32 bits).
- **Lenguajes de Programación:**
    - **Python 3:** Utilizado para la capa de interfaz gráfica (GUI), la lógica de negocio principal (incluyendo llamadas a API REST) y la comunicación entre procesos 64/32 bits mediante msl-loadlib.
    - **C (Estándar C99/C11):** Empleado para crear la función puente (gini_processor.c) que actúa como intermediaria entre el servidor Python 32-bit y la rutina en ensamblador.
    - **Ensamblador (NASM):** Utilizado para implementar la rutina de bajo nivel (float_rounder.asm) que realiza el cálculo específico (redondeo float a int) siguiendo la sintaxis Intel.
- **Compiladores y Ensambladores:**
    - **GCC (GNU Compiler Collection):** Usado para compilar el código C. Se empleó la opción -m32 para generar código objeto y la biblioteca compartida final para la arquitectura de 32 bits. Flags adicionales relevantes: -shared, -fPIC (para la biblioteca compartida), -g (para información de depuración), -Wall (para habilitar advertencias).
    - **NASM (Netwide Assembler):** Utilizado para ensamblar el código fuente .asm. Se empleó la opción -f elf para generar un archivo objeto en formato ELF de 32 bits, compatible con el enlazador de GCC en Linux. Flags adicionales: -g -F dwarf (para información de depuración compatible con GDB).
- **Bibliotecas y Frameworks Clave:**
    - **Python requests:** Para realizar solicitudes HTTP a la API REST del Banco Mundial de forma sencilla.
    - **Python tkinter:** Para la construcción de la interfaz gráfica de usuario (GUI) de la aplicación.
    - **Python msl-loadlib:** Biblioteca fundamental para este proyecto, utilizada para gestionar la IPC entre el cliente Python 64-bit y el servidor Python 32-bit. Permite que el proceso de 64 bits invoque funciones dentro de una biblioteca compartida de 32 bits cargada por el servidor de 32 bits.

    - **Biblioteca C Estándar:** Incluida implícitamente por GCC (stdio.h utilizada para printf en depuración del puente C).
- **Herramientas de Construcción y Ejecución:**
    - **Scripts Bash:** Se crearon scripts (setup.sh, build.sh, run.sh) para automatizar la configuración del entorno, la compilación de los componentes C/ASM y la ejecución de la aplicación principal.
- **Control de Versiones:**
    - **Git:** Utilizado para el control de versiones local.
    - **GitHub:** Empleado como repositorio remoto centralizado para la colaboración y gestión del código fuente del grupo.
- **Herramientas de Depuración:**
    - **GDB (GNU Debugger):** Herramienta esencial utilizada para depurar la interacción entre el código C y el código Ensamblador, con un enfoque específico en el análisis del estado de la pila de llamadas (stack) durante la ejecución.

3. Diseño de la Solución

3.1. Arquitectura General

La solución se diseñó siguiendo una arquitectura por capas bien definida para separar las responsabilidades y facilitar la interacción entre los diferentes niveles de abstracción y arquitecturas (64 bits vs 32 bits), como se aprecia en la Tabla 1.

Python: GUI
Python: Core Logic Module (64 bits)
Python: 32b C Bridge Module (32Bits)
C: ASM Bridge Module (32 bits)
ASM: ASM Round Function

Tabla 1. Representación por capas del proyecto.

3.2. Capa Superior (Python)

3.2.1. Interfaz Gráfica

Se implementó una GUI simple y funcional utilizando tkinter y los widgets temáticos ttk. Proporciona un campo de entrada para el código de país, un botón para iniciar la obtención de datos, áreas para mostrar el último valor GINI encontrado y un historial de datos, un botón para disparar el procesamiento C/ASM, y una barra de estado. La GUI interactúa exclusivamente con el módulo core_logic.py, delegando toda la lógica de negocio y comunicación. Esta se aprecia en la Figura 3.1.

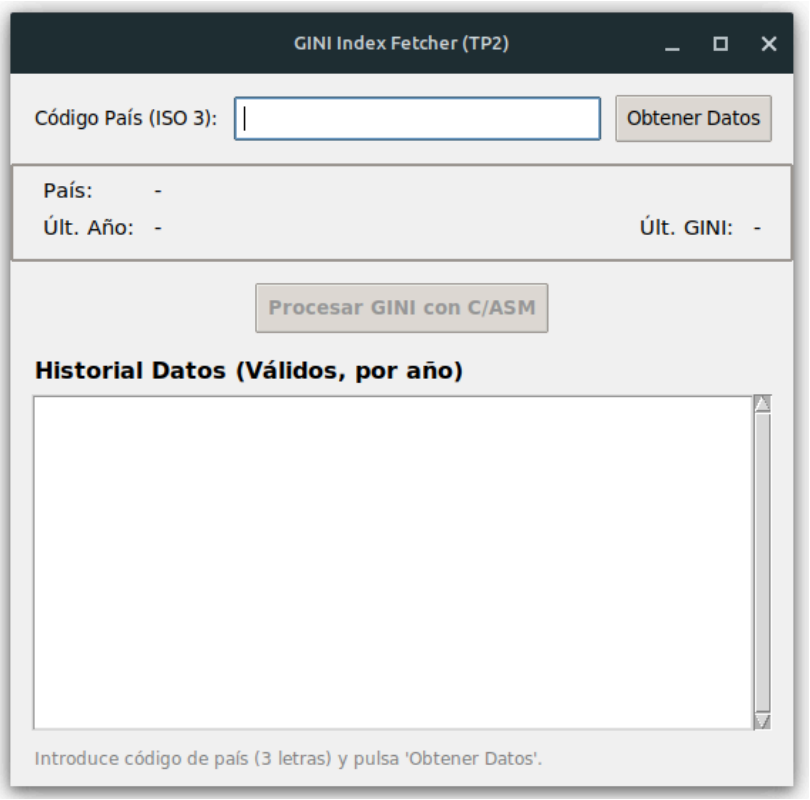

Figura 3.1. GUI implementada para el proyecto.

3.2.2. Lógica Central

Este módulo contiene el núcleo de la aplicación Python.

- **Obtención de Datos:** La función get_gini_data encapsula la lógica para construir la URL de la API del Banco Mundial, realizar la solicitud HTTP GET con requests, manejar diferentes tipos de respuestas (éxito, errores HTTP, errores de API, JSON inválido, sin datos) y devolver los registros o un mensaje de error.
- **Procesamiento de Datos:** La función find_latest_valid_gini itera sobre los registros devueltos por la API para encontrar el dato correspondiente al año más reciente con un valor GINI numéricamente válido.
- **Cliente 64-bit (GiniClient64):** Subclase de msl.loadlib.Client64. Se encarga de iniciar y comunicarse con el Server32. Expone un método (process_gini_float_on_server) que oculta los detalles de request32 y llama al método correspondiente en el servidor 32-bit.
- **Orquestación:** La función process_gini_with_c_asm coordina la llamada al servidor 32-bit a través de la instancia GiniClient64 y devuelve el resultado final a la GUI.

3.3. Capa Intermedia (Puentes Python y C)

3.3.1. Servidor 32-bit (server32_bridge.py)

Script Python diseñado para ejecutarse en un intérprete de 32 bits gestionado por msl-loadlib.

- **Clase GiniProcessorServer:** Subclase de msl.loadlib.Server32. En su inicialización (`__init__`), localiza y carga la biblioteca compartida libginiprocessor.so (que contiene el código C y ASM compilado) utilizando ctypes (manejado internamente por Server32). Define la firma (argtypes, restype) de la función C process_gini_float que se va a llamar.
- **Exposición de Funciones:** Expone un método (con el mismo nombre, `process_gini_float`) que es llamado por el Client64 vía request32. Este método recibe el float GINI, llama a la función C cargada desde la biblioteca .so (`self.lib.process_gini_float(...)`) y devuelve el resultado entero recibido de C al Client64.

3.3.1. Puente C (gini_processor.c)

Componente clave compilado en libginiprocessor.so.

- **Función process_gini_float:** Es la función exportada y llamada por el Server32. Recibe un único argumento float. Declara una variable local `int result_from_asm`. Llama a la función externa `asm_float_round`, pasándole el float recibido y la dirección de la variable local result_from_asm. **Esta llamada sigue la convención cdecl**. Finalmente, retorna el valor contenido en `result_from_asm` (que fue modificado por la rutina ASM).

3.4. Capa Inferior (Ensamblador)

3.4.1. Rutina Ensamblador (float_rounder.asm)

Contiene la lógica de bajo nivel, implementada en NASM para x86 32-bit.

- **Función asm_float_round:** Se declara como global para ser visible al enlazador C. Implementa el prólogo y epílogo estándar de cdecl (push ebp, mov ebp, esp, ..., mov esp, ebp, pop ebp, ret).
- **Acceso a Parámetros:** Accede a los parámetros pasados por C desde la pila utilizando el puntero base ebp: el valor float en [ebp+8] y el puntero int* en [ebp+12].
- **Cálculo:** Utiliza instrucciones de la FPU (Unidad de Punto Flotante): fld para cargar el float de la pila ([ebp+8]) en el registro FPU st0, y fistp para redondear st0 al entero más cercano (según el modo de redondeo de la FPU, típicamente "round half to even") y almacenar el resultado entero en una ubicación temporal en la pila local ([ebp-8]).
- **Escritura del Resultado:** Carga el puntero int* ([ebp+12]) en un registro (e.g., edx), carga el resultado entero redondeado ([ebp-8]) en otro registro (e.g., eax), y

finalmente escribe el valor entero en la dirección de memoria apuntada por el puntero (mov [edx], eax).
- **Retorno:** Ejecuta ret para devolver el control a la función C llamante

3.5. Mecanismo de Interfaz C - Ensamblador

La comunicación entre la capa C y la capa Ensamblador se diseñó adhiriendo estrictamente a la utilización de la pila de llamadas (stack) y su stack frame para el intercambio de datos.

Para ello, se implementó una convención de llamadas correspondiente a cdecl, la cuál es estándar para sistemas de 32 bits en Linux y GCC. Es decir:

- Los argumentos se pasan de C a ASM *empujándolos* en la pila de derecha a izquierda. Al llamar `asm_float_round(float_val, &int_ptr)`, C primero empuja `&int_ptr` y luego `float_val`.
- Dentro de ASM, después del prólogo, `float_val` está en `[ebp+8]` y `&int_ptr` en `[ebp+12]`.
- La rutina ASM (callee) no es responsable de limpiar los parámetros de la pila; esto lo hará la función C (caller) después de que asm_float_round retorne.
- La rutina ASM preserva los registros callee-save requeridos (ebp se guarda y restaura; ebx, esi, edi no se usan por lo que no es necesario reestablecerlos). Los registros eax y edx se usan como scratch (caller-save).

Cabe destacar que el resultado entero no se devuelve a través de un registro (como EAX) desde ASM a C. En su lugar, C pasa un puntero a una ubicación de memoria, y ASM escribe directamente el resultado en esa ubicación de memoria. La función C luego lee esa variable local para obtener el resultado.

4. Descripción del Flujo del Programa

Esta sección se centra en describir uno de los flujos de ejecución más relevantes del sistema: el procesamiento de un valor GINI utilizando las capas de C y Ensamblador, iniciado desde la interfaz de usuario. En lugar de repetir fragmentos de código que pueden consultarse directamente en el repositorio fuente, se utilizará el siguiente diagrama de secuencia para visualizar la interacción entre los distintos componentes involucrados en este caso de uso específico. En la Figura 4.1, se muestra el diagrama de secuencia correspondiente para un caso de uso típico del programa.

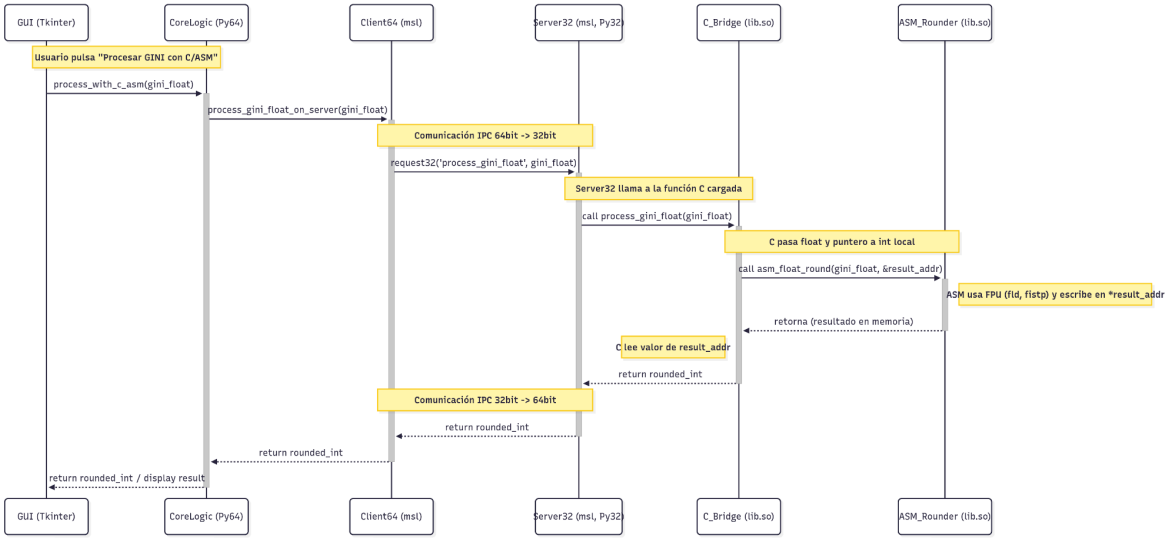

Figura 4.1. Diagrama de Secuencia de la Solución.

El diagrama ilustra la secuencia de llamadas y el paso de datos que ocurren cuando el usuario activa el procesamiento C/ASM desde la GUI (Tkinter):

- **Inicio (GUI**): La GUI (G) invoca la función process_with_c_asm en el módulo CoreLogic (CL) de 64 bits, pasando el valor GINI float.
- **Cliente 64-bit**: El CoreLogic utiliza la instancia Client64 (C64) proporcionada por msl-loadlib para solicitar la ejecución remota de process_gini_float_on_server.
- **Comunicación IPC (64->32)**: El Client64 se comunica con el Server32 (S32), que corre en un proceso Python de 32 bits, enviando el nombre de la función y el argumento float.
- **Servidor 32-bit**: El Server32 recibe la petición y llama a la función C process_gini_float dentro de la biblioteca compartida (C_Bridge, CB) que tiene cargada.
- **Puente C**: La función C (process_gini_float) prepara la llamada a la rutina ensamblador. Llama a asm_float_round (dentro de ASM_Rounder, ASM), pasando el valor float original y la dirección de memoria (&result_addr) de una variable local int donde se debe escribir el resultado.
- **Rutina Ensamblador**: La rutina asm_float_round recibe los parámetros desde la pila (siguiendo cdecl), utiliza instrucciones de la FPU (fld, fistp) para cargar el float, redondearlo al entero más cercano y almacenar el resultado directamente en la dirección de memoria apuntada por &result_addr. Finalizada la operación, retorna el control a C.
- **Retorno C**: La función C ahora tiene el resultado redondeado en su variable local (result_addr). Retorna este valor int al Server32.
- **Comunicación IPC (32->64)**: El Server32 envía el resultado int de vuelta al Client64.
- **Retorno 64-bit**: El Client64 devuelve el int al CoreLogic.
- **Resultado Final (GUI)**: El CoreLogic pasa el resultado final a la GUI, que lo muestra al usuario.

5. Pruebas y Análisis con GDB

5.1. Propósito de la Depuración con GDB

Esta sección cumple con el requisito fundamental de cualquier proyecto: demostrar y verificar el mecanismo de interacción entre partes críticas del código que de manera nativa son puntos de fallo por excelencia: la interacción C (gini_processor.c) y la rutina en Ensamblador (float_rounder.asm), todo utilizando el depurador GDB. El objetivo es visualizar el estado de la pila de llamadas (stack) en puntos clave del código para confirmar que la convención de llamada `cdecl` se aplica correctamente para el paso de parámetros y la gestión del marco de pila (stack frame) en la arquitectura de 32 bits. Se utilizó un programa C mínimo (gdb_test.c) enlazado directamente con los componentes C y ASM compilados para depuración (-m32 -g) para facilitar este análisis.

5.2. Programa de Prueba (gdb_test.c) y Compilación

Se utilizó el programa `gdb_test.c` (disponible en el repositorio) que realiza dos llamadas consecutivas a la función puente C `process_gini_float`, primero con el valor `42.75f` y luego con `35.2f`. La compilación se realizó mediante el script debug.sh (Figura 5.1), que ejecuta los siguientes pasos clave:

1. Compila gini_processor.c a objeto (.o) con `gcc -m32 -g`.
2. Ensambla float_rounder.asm a objeto (.o) con `nasm -f elf -g -F dwarf`.
3. Compila gdb_test.c a objeto (.o) con `gcc -m32 -g.`
4. Enlaza los tres objetos .o en el ejecutable build/gdb_test con `gcc -m32 -g -no-pie`.

```
(venv) fernando@Main-Fluor-Linux:~/Documents/tp2-SdC$ ./debug.sh
--- Iniciando Compilación para Depuración y Lanzamiento de GDB ---
Paso 1: Verificando archivos fuente...
Archivos fuente encontrados.

Paso 2: Asegurando directorio de build...
Directorio 'build' listo.

Paso 3: Compilando componentes C/ASM para debug (32-bit)...
Compilando C: src/c_bridge/gini_processor.c -> build/gini_processor_dbg.o
  -> Compilación C OK.
Ensamblando ASM: src/c_bridge/float_rounder.asm -> build/float_rounder_dbg.o
  -> Ensamblado ASM OK.
Compilando Test GDB: src/gdb_test.c -> build/gdb_test_dbg.o
  -> Compilación Test GDB OK.

Paso 4: Enlazando objetos para crear ejecutable 'build/gdb_test'...
Enlazando: build/gdb_test_dbg.o, build/gini_processor_dbg.o, build/float_rounder_dbg.o -> build/gdb_test
Enlazado completado. Ejecutable creado en 'build/gdb_test'.

Paso 5: Lanzando GDB sobre 'build/gdb_test'...
```

FIgura 5.1. Ejecución del Batch Script.

5.3. Ejecución y Análisis con GDB

Se inició GDB con `gdb ./build/gdb_test` y se establecieron los siguientes breakpoints estratégicos, tal como se aprecia en la Figura 5.2:

- b main: Al inicio de gdb_test.c.
- b process_gini_float: Al inicio de la función puente C.
- b asm_float_round: Al inicio de la rutina ASM.
- b gdb_test.c:25: Después del retorno de la primera llamada en main.
- b gdb_test.c:32: Después del retorno de la segunda llamada en main.

```
Reading symbols from build/gdb_test...
(gdb) b main
Breakpoint 1 at 0x80491b3: file src/gdb_test.c, line 14.
(gdb) b process_gini_float
Breakpoint 2 at 0x80492c4: file src/c_bridge/gini_processor.c, line 25.
(gdb) b asm_float_round
Breakpoint 3 at 0x8049370: file src/c_bridge/float_rounder.asm, line 25.
(gdb) b gdb_test.c:25
Breakpoint 4 at 0x80491fb: file src/gdb_test.c, line 25.
(gdb) b gdb_test.c:32
Breakpoint 5 at 0x8049259: file src/gdb_test.c, line 32.
(gdb) run
```

Figura 5.2. Breakpoints Estratégicos.

A continuación, se detalla el análisis en los puntos más críticos de la ejecución:

5.3.1. Antes de llamar a process_gini_float (1ra llamada, en main)

La ejecución se detiene en main antes de la primera llamada (Figura 5.3). Se inspeccionan las variables locales y la dirección donde se espera el resultado.

```
Breakpoint 1, main () at src/gdb_test.c:14
14          float test_gini_value = 42.75f; // Ejemplo con parte fraccionaria > 0.5
(gdb) print "Estado en main antes de la llamada 1"
$1 = "Estado en main antes de la llamada 1"
(gdb) info locals
test_gini_value = -1.02190267e+34
result = 1
(gdb) p test_gini_value
$2 = -1.02190267e+34
(gdb) p &result
$3 = (int *) 0xffffcafc
(gdb) x/16wx $esp
0xffffcaf0:     0xffffcb30      0xf7fbe66c      0xf7fbeb20      0x00000001
0xffffcb00:     0xffffcb20      0xf7f9d000      0xf7ffd020      0xf7d94519
0xffffcb10:     0xffffcdd9      0x00000070      0xf7ffd000      0xf7d94519
0xffffcb20:     0x00000001      0xffffcbd4      0xffffcbdc      0xffffcb40
(gdb) info reg ebp esp
ebp            0xffffcb08          0xffffcb08
esp            0xffffcaf0          0xffffcaf0
```

Figura 5.3. Estado de la ejecución para el Breakpoint 1.

Como se puede observar, es posible identificar la dirección `0xffffcafc` para la variable result local de main.

5.3.2. Al entrar en process_gini_float (1ra llamada)

La ejecución se detiene al inicio de la función C (Figura 5.4). Se verifica el argumento recibido y la dirección de la variable local donde ASM escribirá.

```
(gdb) c
Continuing.
[gdb_test] Iniciando prueba con GINI = 42.750000
[gdb_test] Llamando a process_gini_float...

Breakpoint 2, process_gini_float (gini_value=42.75) at src/c_bridge/gini_processor.c:25
25      int process_gini_float(float gini_value) {
(gdb) print "Estado al Entrar a process_gini_float en la llamada 1"
$4 = "Estado al Entrar a process_gini_float en la llamada 1"
(gdb) info args
gini_value = 42.75
(gdb) info locals
result_from_asm = 0
(gdb) p gini_value
$5 = 42.75
(gdb) p &result_from_asm
$6 = (int *) 0xffffcac8
(gdb) x/16wx $esp
0xffffcac0:     0xffffcb08      0xf7fd9004      0x00000000      0x0804c000
0xffffcad0:     0xffffcbd4      0x0804c000      0xffffcb08      0x080491f5
0xffffcae0:     0x422b0000      0x00000000      0x40456000      0x080491ad
0xffffcaf0:     0xffffcb30      0xf7fbe66c      0x422b0000      0x00000001
(gdb) info reg ebp esp
ebp            0xffffcad8          0xffffcad8
esp            0xffffcac0          0xffffcac0
```

Figura 5.4. Ejecución del Breakpoint 2.

Aquí se confirma que `gini_value` es `42.75f` y se identifica la dirección `0xffffcac8` para `result_from_asm`. Esta dirección es la que se pasará a la rutina ensamblador.

5.3.3. Al entrar en asm_float_round (1ra llamada)

Este es el punto crucial para verificar la pila según `cdecl`. Mediante la Figura 5.5, se confirma que el valor float (`42.75`, representado como `0x422b0000`) se encuentra en la pila en la posición `[ebp+8]`, y la dirección para el resultado (`0xffffcac8`) se encuentra en `[ebp+12]`. Esto valida el correcto paso de parámetros de C a ASM según la convención `cdecl`.

```
(gdb) c
Continuing.
INFO [C Bridge] Recibido de Python/Server32: float = 42.750000
INFO [C Bridge] Dirección para resultado ASM (&result_from_asm): 0xffffcac8
INFO [C Bridge] Llamando a asm_float_round...

Breakpoint 3, asm_float_round () at src/c_bridge/float_rounder.asm:25
25          push    ebp                 ; 1. Guarda EBP del llamante (Preserva EBP)
(gdb) print "Estado al entrar en asm_float en la primera llamada"
$7 = "Estado al entrar en asm_float en la primera llamada"
(gdb) disas
Dump of assembler code for function asm_float_round:
=> 0x08049370 <+0>:     push   %ebp
   0x08049371 <+1>:     mov    %esp,%ebp
   0x08049373 <+3>:     sub    $0x8,%esp
   0x08049376 <+6>:     flds   0x8(%ebp)
   0x08049379 <+9>:     mov    0xc(%ebp),%edx
   0x0804937c <+12>:    fistpl -0x8(%ebp)
   0x0804937f <+15>:    mov    -0x8(%ebp),%eax
   0x08049382 <+18>:    mov    %eax,(%edx)
   0x08049384 <+20>:    mov    %ebp,%esp
   0x08049386 <+22>:    pop    %ebp
   0x08049387 <+23>:    ret
   0x08049388 <+24>:    xchg   %ax,%ax
   0x0804938a <+26>:    xchg   %ax,%ax
   0x0804938c <+28>:    xchg   %ax,%ax
   0x0804938e <+30>:    xchg   %ax,%ax
End of assembler dump.
(gdb) info reg ebp esp
ebp            0xffffcad8          0xffffcad8
esp            0xffffcaac          0xffffcaac
(gdb) x/wx $ebp+4
0xffffcadc:     0x080491f5
(gdb) x/f $ebp+8
0xffffcae0:     42.75
(gdb) x/a $ebp+12
0xffffcae4:     0x0
(gdb) x/20wx
0xffffcae8:     0x40456000      0x080491ad      0xffffcb30      0xf7fbe66c
0xffffcaf8:     0x422b0000      0x00000001      0xffffcb20      0xf7f9d000
0xffffcb08:     0xf7ffd020      0xf7d94519      0xffffcdd9      0x00000070
0xffffcb18:     0xf7ffd000      0xf7d94519      0x00000001      0xffffcbd4
0xffffcb28:     0xffffcbdc      0xffffcb40      0xf7f9d000      0x08049196
(gdb) x/20wx $esp
0xffffcaac:     0x08049329      0x422b0000      0xffffcac8      0x40456000
0xffffcabc:     0x080492be      0xffffcb08      0xf7fd9004      0xffffffc19
0xffffcacc:     0x88635d00      0xffffcbd4      0x0804c000      0xffffcb08
0xffffcadc:     0x080491f5      0x422b0000      0x00000000      0x40456000
0xffffcaec:     0x080491ad      0xffffcb30      0xf7fbe66c      0x422b0000
```

Figura 5.5. Resultado del Breakpoint 3.

5.3.4. Después de retornar de process_gini_float (1ra llamada, en main)

La ejecución se detiene en main (Figura 5.6) después de que la función C (y ASM) completaran su trabajo.

```
(gdb) c
Continuing.
INFO [C Bridge] Retorno de asm_float_round.
INFO [C Bridge] Valor escrito por ASM en &result_from_asm: 43

Breakpoint 4, main () at src/gdb_test.c:25
25          printf("[gdb_test] Retorno de process_gini_float.\n");
(gdb) print "Estado de main despues de la 1ra llamada"
$8 = "Estado de main despues de la 1ra llamada"
(gdb) info locals
test_gini_value = 42.75
result = 43
(gdb) p result
$9 = 43
(gdb) x/16wx $esp
0xffffcaf0:     0xffffcb30      0xf7fbe66c      0x422b0000      0x0000002b
0xffffcb00:     0xffffcb20      0xf7f9d000      0xf7ffd020      0xf7d94519
0xffffcb10:     0xffffcdd9      0x00000070      0xf7ffd000      0xf7d94519
0xffffcb20:     0x00000001      0xffffcbd4      0xffffcbdc      0xffffcb40
(gdb) info reg ebp esp
ebp            0xffffcb08          0xffffcb08
esp            0xffffcaf0          0xffffcaf0
```

Figura 5.6. Finalización de la primera llamada a ASM.

La variable result en main ahora contiene el valor `43`, el entero redondeado esperado de `42.75f`. Esto demuestra que ASM escribió correctamente en la dirección proporcionada y C devolvió el valor adecuadamente.

5.3.5. Análisis de la Segunda Llamada (Valor 35.2f)

Se repitió el proceso para la segunda llamada con el valor `35.2f`. El análisis en los breakpoints correspondientes mostró resultados análogos, ilustrado en la Figura 5.7:

- En `process_gini_float`: Se recibió `gini_value = 35.2000008f.` La dirección para result_from_asm fue nuevamente `0xffffcac8`.
- En `asm_float_round`: La pila contenía el float `35.2000008` en `[ebp+8]` y la dirección `0xffffcac8` en `[ebp+12]`.
- En main (después de la 2da llamada): La variable result contenía `35`, el redondeo correcto de `35.2f.`

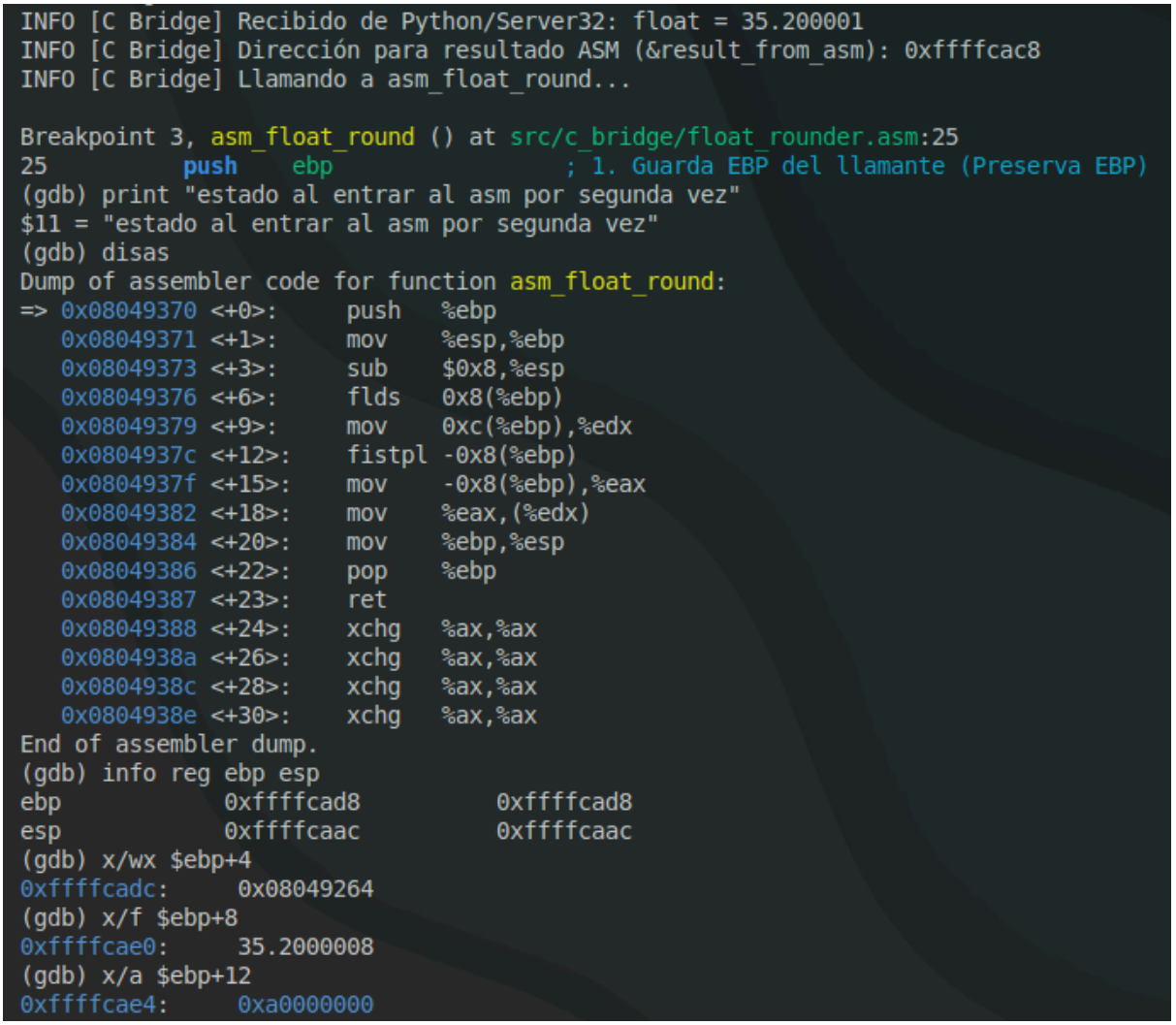

Figura 5.7. Finalización de la segunda llamada.

5.4. Interpretación y Conclusiones de la Depuración

El análisis detallado de la ejecución paso a paso con GDB permitió verificar de forma concluyente los siguientes aspectos de la interfaz C-Ensamblador implementada:

- **Paso de Parámetros (cdecl):** Se demostró visualmente que C “empujó” los argumentos (valor float, puntero int*) en la pila en el orden correcto (derecha a izquierda) y que ASM pudo accederlos usando los desplazamientos estándar relativos a EBP (+8, +12).
- **Retorno Indirecto vía Puntero:** Se confirmó que ASM escribió el resultado entero redondeado directamente en la dirección de memoria que C especificó a través del puntero pasado por la pila.
- **Integridad Funcional:** Se validó que el resultado final obtenido en main después de la cadena de llamadas C->ASM->C era el esperado matemáticamente para ambos casos de prueba (42.75 -> 43, 35.2 -> 35).
- **Gestión del Marco de Pila:** La correcta recepción de parámetros y retorno de valores evidencia la adecuada gestión del marco de pila según cdecl.

6. Conclusiones

Este Trabajo Práctico ha culminado exitosamente en el diseño e implementación de un sistema funcional que integra múltiples capas de software y arquitecturas, demostrando la viabilidad y los mecanismos necesarios para la comunicación entre lenguajes de alto nivel (Python, C) y bajo nivel (Ensamblador x86).

6.1. Objetivos Cumplidos

Se cumplieron los objetivos principales planteados:

1. **Integración Multi-Capa y Multi-Arquitectura:** Se desarrolló una aplicación Python 64-bit con interfaz gráfica (Tkinter) capaz de consumir una API REST externa (Banco Mundial). Fundamentalmente, se implementó con éxito un puente, utilizando `msl-loadlib`, para que esta aplicación pudiera invocar funcionalidad residente en una biblioteca compartida de 32 bits.
2. **Interfaz C-Ensamblador:** Se creó una biblioteca (`libginiprocessor.so`) conteniendo un módulo C que actúa como intermediario y una rutina en Ensamblador (`float_rounder.asm`) que realiza una operación específica (redondeo float a entero mediante FPU).
3. **Uso de Convenciones de Llamada (cdecl):** La comunicación entre C y Ensamblador se basó estrictamente en la convención de llamada cdecl, utilizando la pila para el paso de parámetros (un valor `float` y un puntero `int*` para retorno indirecto) y para la gestión del marco de pila.
4. **Validación con GDB:** Mediante el uso de un programa de prueba dedicado (gdb_test.c) y el depurador GDB, se **verificó empíricamente** el correcto funcionamiento de la interfaz C-Ensamblador. El análisis detallado de la pila ($esp, $ebp) y el acceso a parámetros ([ebp+8], [ebp+12]) en puntos clave de la ejecución demostró que los datos se transmitían y recibían según lo estipulado por la convención cdecl, validando así el núcleo de bajo nivel de la solución.

6.2. Consideraciones Finales

Si bien el objetivo principal no era la optimización de rendimiento (la sobrecarga de la comunicación inter-proceso probablemente supera cualquier ganancia del cálculo en ASM para esta tarea simple), el proyecto sirvió como un excelente ejercicio práctico para aplicar conceptos de organización del computador, lenguajes de bajo nivel y arquitecturas de software. La capacidad de integrar componentes de diferentes naturalezas es una habilidad valiosa en el desarrollo de software moderno.