FACULTAD DE CIENCIAS EXACTAS, FÍSICAS y NATURALES

SISTEMAS DE COMPUTACIÓN

TP2: STACK FRAME

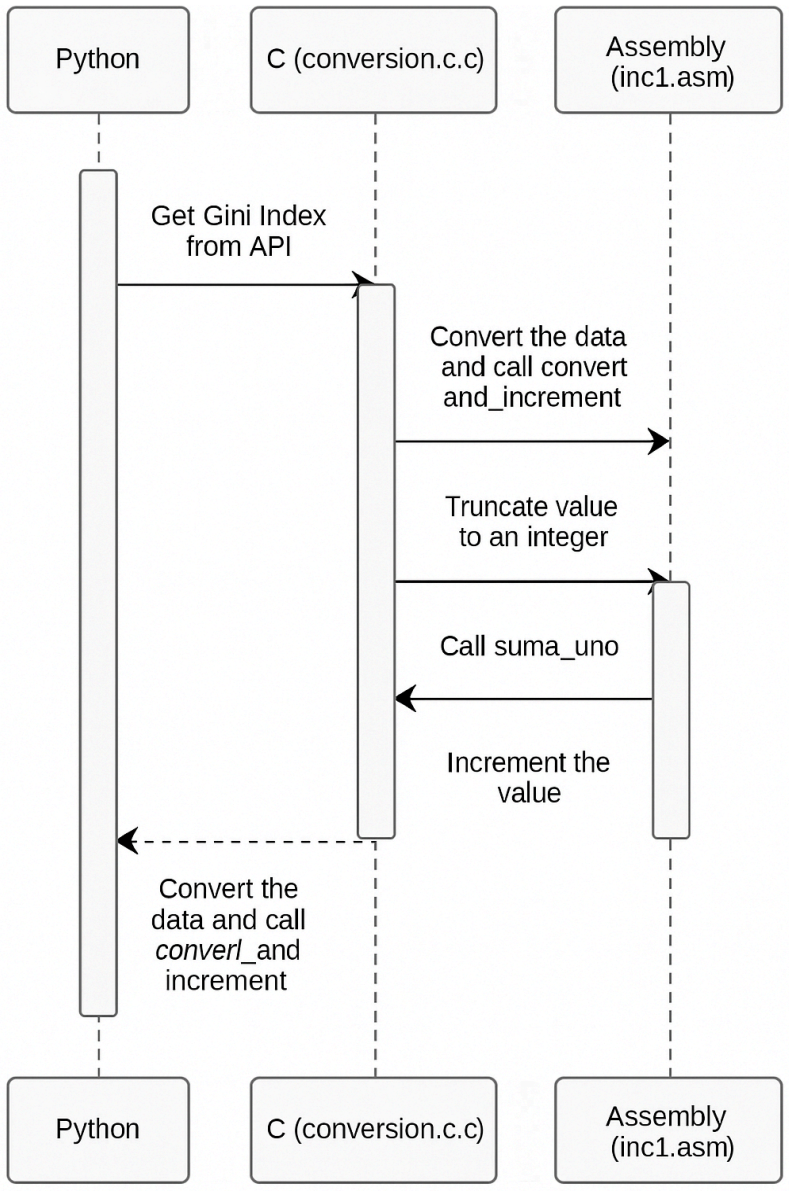
Nombre	DNI	Carrera
Facundo Nahuel Galvagno	40815088	IE
Tomás Ignacio Corbalán	38503847	ICOMP
Hernan Imanol Rodriguez	41808427	ICOMP

Introducción

El objetivo del trabajo es utilizar un stack completo de assembler, C y Python para obtener datos de una API rest, realizar operaciones básicas con los mismos y luego mostrarlos en pantalla. La API consultada fue la del índice Gini, proporcionada por el banco mundial (https://api.worldbank.org/v2/en/country/all/indicator/SI.POV.GINI).

En una primera instancia se utilizó C en conjunto con assembler para mostrar los datos. Luego utilizando el módulo requests y dash de python, se creó una interfaz gráfica para mostrar los datos.

Diagrama de secuencia del programa



Desarrollo

En primer lugar se desarrolló la aplicación utilizando solo C. Como se debía realizar llamado a un lenguaje de más bajo nivel (assembler) desde C, se compiló una librería en assembler que toma el dato y le suma uno.

```
segment .text
        global  suma_uno

suma_uno:
        enter   4,0               ; Asigna espacio en la pila para
variables locales (aunque no las usemos directamente).
                                  ; Esto crea un frame de pila para
fines didácticos.

        push    ebx               ; Guarda EBX, ya que lo usaremos.
IMPORTANTE!


        ; Muestra el frame de pila actual.  Útil para entender cómo se
organiza la pila.
        dump_stack 1, 2, 4        ; Imprime el contenido de la pila
desde ebp-8 hasta ebp+16
        mov     eax, [ebp+8]      ; eax = n
        mov     ebx, [ebp+12]     ; sum (segundo argumento)
        ; Incrementa el valor en EAX.
        inc     eax

        ; Muestra el frame de pila después de la operación de
incremento.
        ;dump_stack 1, 2, 4        ; Imprime el contenido de la pila
desde ebp-8 hasta ebp+16

        ; Almacena el valor incrementado en la dirección apuntada por
EBX.
        mov     [ebx], eax

        pop     ebx               ; Restaura EBX.
        leave                      ; Restaura el frame de pila
(equivalente a 'mov esp, ebp; pop ebp').
        ret
```

Este código se compilo siguiendo las directivas expuestas en los ejemplos de Paul Carter y su libro de assembler x86.

Interfaz C y asmx86

La forma de vincular assembler con C de manera que pueda operar fácilmente como un modulo, haciendo llamados de función, es alinearse a la convención de llamadas de C.

```
#include "cdecl.h"
void PRE_CDECL calc_sum( int, int * ) POST_CDECL; /* prototype for
assembly routine
```

De los ejemplos de paul carter podemos utilizar como pseudolibreria el archivo objeto asm_io.o, que utiliza la convención de llamadas a C y permite usar la Standard Input/Output

```
; file: sub5.asm
; Subprogram to C interfacing example

%include "asm_io.inc"
```

Con el archivo make descargado (el definido para los ejemplo linux) podemos rescatar el proceso de compilación completo de un binario

```
gcc -c -m32  main5.c   //compilo .c con arquitectura 32 bits (-m32) y
solo compile (-c)
nasm -f elf sub5.asm  // ensambla el codigo en un ejecutable elf para
linux
nasm -f elf -d ELF_TYPE asm_io.asm
gcc -m32  -osub5 main5.o sub5.o asm_io.o
```

En base a este modelo, el proceso de compilar el ensamblador es:

```
  nasm -f elf inc1.asm
  gcc -c -m32 suma_uno.c
  nasm -f elf -d ELF_TYPE asm_io.asm // de ser necesario
  gcc -m32  -osumauno suma_uno.o inc1.o asm_io.o
  ./sumauno //ejecuta el binario
```

El codigo en C para operar el assembly es

```
//Para compilar gcc -Iheaders/ -shared -o libconversion.so -fPIC
conversion.c
#include "conversion.h"
#include <stdio.h>
#include "cdecl.h"
```

```
void PRE_CDECL suma_uno(int, int * ) POST_CDECL;

int main(void){
      float num = 3.5;
      convert_and_increment(num);
      return 0;
}

int convert_and_increment(float value) {
      int ivalue = (int)value;      // truncamiento hacia cero
      printf("%d\n", ivalue);
            suma_uno(value, &ivalue); //llamamiento a funcion asm
            printf("%d\n", ivalue);
      return ivalue;
}
```

Como se observa en el código, se llama a la directiva de ensamblador “suma_uno” como si de una función C se tratase.

Interfaz de usuario y get de API rest con Python

Se utilizó Python como lenguaje de alto nivel para obtener los datos de la API, llamar a las funciones en C y posteriormente graficar el resultado obtenido mediante algún sistema de interfaz (en este caso, por la interactividad y flexibilidad decidimos usar dash).

Utilizando el módulo requests, podemos consumir la API y obtener los datos de cualquier país deseado como un dataframe de pandas.

```
def fetch_gini_data(country_code):
    api_url =
f'https://api.worldbank.org/v2/en/country/{country_code}/indicator/SI.PO
V.GINI?format=json&per_page=1000'
    response = requests.get(api_url)
    if response.status_code != 200 or len(response.json()) < 2:
        return pd.DataFrame(columns=['date', 'value'])
    data = response.json()[1]
    df = pd.DataFrame(data)
    df = df[['date', 'value']].dropna()
    df['date'] = df['date'].astype(int)
    df['value'] = df['value'].astype(float)
    return df.sort_values('date')
```

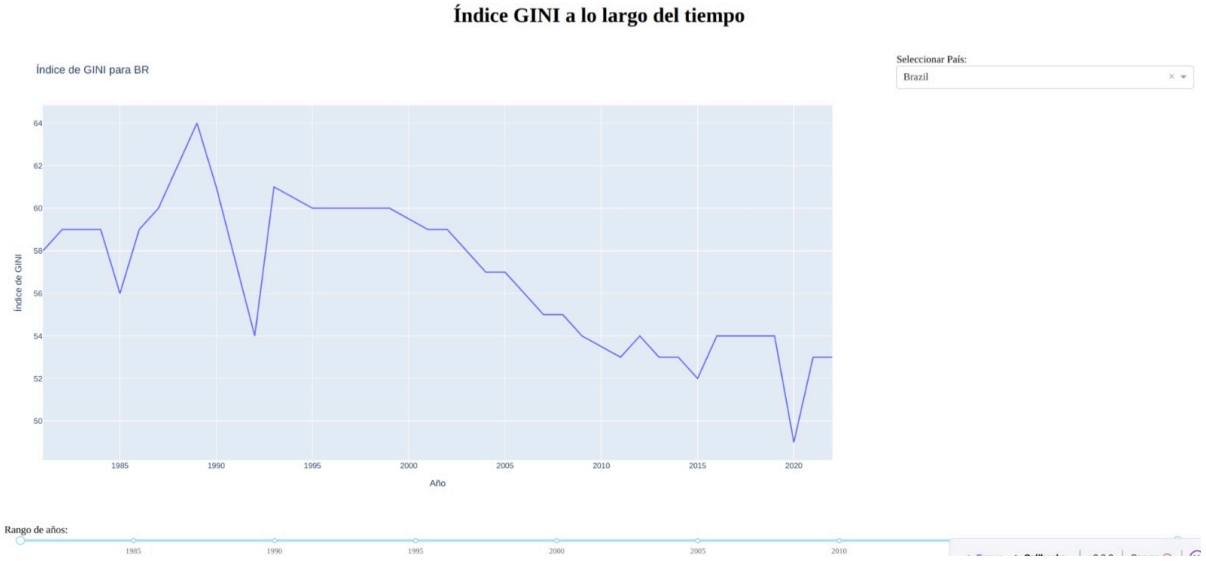
Al utilizar un intérprete de Python de 64 bits, no podemos usar de manera directa nuestras librerías compiladas en C para 32 bits. Para resolver este problema, se utiliza el módulo msl-loadlib. Mediante sockets y un esquema cliente-servidor, el módulo permite que

carguemos una librería compartida de 32 bits desde un intérprete de Python de 64 bits. A partir de la librería compartida libconversion.so se configura el módulo para poder acceder mediante el llamado a un método definido como convert_and_increment() del objeto lib.

El dataframe de pandas permite aplicar iterativamente el método a todas los datos de índice Gini del mismo.

```
filtered_df['value'] = df['value'].apply(lib.convert_and_increment)
```

Los datos se muestran utilizando gráficos de plotly con dash.



Dash es un framework para crear aplicaciones de visualización de datos. El layout de los componentes es realizado con html.

```
app.layout = html.Div([
    html.H1("Índice GINI a lo largo del tiempo", style={'textAlign':
'center'}),

    html.Div([
        dcc.Graph(id='gini-graph', style={'flex': '3', 'height':
'100%'}),

        html.Div([
            html.Label("Seleccionar País:"),
            dcc.Dropdown(
                id='country-selector',
                options=[
                    {'label': 'Argentina', 'value': 'ARG'},
                    {'label': 'Brazil', 'value': 'BR'},
                    {'label': 'India', 'value': 'IN'},
                    {'label': 'Estados Unidos', 'value': 'US'},
```

```
                    {'label': 'Alemania', 'value': 'DE'}
                ],
                value='ARG'
            )
        ], style={'flex': '1', 'padding': '20px'})
    ], style={'display': 'flex', 'height': '80vh'}),

    html.Div([
        html.Label("Rango de años:"),
        dcc.RangeSlider(
            id='year-slider',
            min=df['date'].min() if not df.empty else 2000,
            max=df['date'].max() if not df.empty else 2020,
            value=[df['date'].min() if not df.empty else 2000,
df['date'].max() if not df.empty else 2020],
            marks={str(year): str(year) for year in range(1960, 2023,
5)},
            step=1
        )
    ], style={'padding': '20px'})
])
```

La interactividad de la aplicación es dada por las callback. Las callback son funciones llamadas por dash al realizar alguna acción, como por ejemplo seleccionar un país en el menú desplegable.

Uno de los problemas que posee Python es que, al ser un lenguaje de programación en continuo desarrollo, muchos de sus módulos y librerías se actualizan constantemente. Esto puede causar problemas de compatibilidad entre dos versiones diferentes de la misma librería. Para solucionar este problema, se utiliza pip freeze en un entorno virtual para crear el archivo requirements.txt

El archivo requirements.txt posee el nombre y versión de todos los módulos necesarios para correr el programa en Python.

Debug del programa mediante gdb

A continuación se analiza cómo varía el contenido de la pila durante la ejecución del programa en C y ensamblador, en el contexto de una función llamada desde C a ASM utilizando la convención de llamada estándar `cdecl` en una arquitectura de 32 bits.

Para comenzar compilamos y linkeamos nuestro programa mediante nasm y gcc de la siguiente manera:

1. Se compilan los ASM (`asm_io.asm` e `inc1.asm`)

```
nasm -f elf -g -F dwarf -Iinclude/ -dELF_TYPE src/asm_io.asm -o
build/asm_io.o
nasm -f elf -g -F dwarf -Iinclude/ src/inc1.asm -o build/inc1.o
```

2. Se compila el archivo en C con debug y 32 bits

```
gcc -m32 -g -Iinclude -c src/conversion.c -o build/conversion.o
```

3. Se linkea todo en un ejecutable

```
gcc -m32 -g build/conversion.o build/asm_io.o build/inc1.o -o
build/programa
```

Luego se ejecuta gdb y se colocan los siguientes breakpoints para realizar el debug:

```
(gdb) break main
Punto de interrupción 1 at 0x11e8: file src/conversion.c, line 9.
(gdb) break convert_and_increment
Punto de interrupción 2 at 0x121e: file src/conversion.c, line 14.
(gdb) break conversion.c:17
Punto de interrupción 3 at 0x125f: file src/conversion.c, line 17.
(gdb) break suma_uno
Punto de interrupción 4 at 0x15c0: file src/inc1.asm, line 26.
(gdb) break inc1.asm:44
Punto de interrupción 5 at 0x15d9: file src/inc1.asm, line 45.
(gdb) break conversion.c:19
Punto de interrupción 6 at 0x12a2: file src/conversion.c, line 19.
```

Función analizada

```
void suma_uno(int valor, int *destino);
```

Esta función es invocada desde el siguiente fragmento en C:

```
int ivalue = (int)value;
suma_uno(value, &ivalue);
```

El objetivo de la función es incrementar el valor pasado como parámetro y almacenar el resultado en la dirección apuntada por el segundo parámetro.

1. Estado de la pila antes de la llamada a suma_uno (breakpoint 3)

Justo antes del call a la función en ensamblador, se observa el siguiente estado de la pila:

```
(gdb) x/32xw $esp
0xffffccf0:     0x00000000      0xffffcfbb      0x00000003      0x037f0f7f
```

```
ESP = 0xffffccf0
0xffffccf8 → 0x00000003    ; Primer parámetro (int value = 3)
0xffffccfc → 0x037f0f7f    ; Segundo parámetro (puntero a ivalue)
```

Estos valores se colocan en la pila por el caller según la convención cdecl, de derecha a izquierda. El valor de retorno será manejado posteriormente en el registro EAX.

2. Estado de la pila al ingresar a suma_uno (breakpoint 4)

La primera instrucción ejecutada en ensamblador es:

```
enter 4, 0
```

Esta instrucción realiza las siguientes acciones:
- Empuja el valor actual de EBP en la pila
- Copia ESP en EBP
- Reserva 4 bytes adicionales para variables locales (aunque no se utilicen)

El siguiente push ebx guarda el contenido del registro EBX. El estado de la pila queda entonces:

```
0xffffcccc:     0x00000004      0x56558fc8      0x00000064      0xffffcd18
0xffffccdc:     0x56556289      0x00000003      0xffffcd08      0x00000000
```

```
ESP = 0xffffcccc
0xffffcccc → 0x00000004      ; Espacio reservado por `enter`
0xffffccd0 → EBX (valor temporal)
0xffffccd4 → antiguo EBP
0xffffccd8 → nuevo EBP
0xffffccdc → dirección de retorno
0xffffcce0 → valor (3)
0xffffcce4 → puntero a ivalue
```

3. Acceso a los parámetros en la función ensamblador

Desde el nuevo marco de pila, los parámetros pueden accederse como:

```
[EBP+8]  → primer argumento (int valor)
[EBP+12] → segundo argumento (int *destino)
```

Las instrucciones ejecutadas son:

```
mov eax, [ebp+8]
mov ebx, [ebp+12]
inc eax
mov [ebx], eax
```

Estas instrucciones incrementan el valor original y lo escriben en la dirección apuntada por ebx.

4. Estado de la pila antes de retornar (breakpoint 5)

Antes de ejecutar leave y ret, se restaura el valor de EBX y se ve el siguiente contenido:

```
(gdb) x/32xw $esp
0xffffccd0:     0x56558fc8      0x00000064      0xffffcd18      0x56556289
0xffffcce0:     0x00000003      0xffffcd08      0x00000000      0x56556218
0xffffccf0:     0x00000000      0xffffcfbb      0x00000003      0x037f0f7f
0xffffcd00:     0x00000020      0x00000000      0x00000004      0xada26f00
0xffffcd10:     0xf7fc7570      0xf7fa2e34      0xffffcd48      0x565561fc
0xffffcd20:     0x40600000      0x00000000      0x00000000      0x565561e3
0xffffcd30:     0x00000000      0x00000000      0x00000000      0x40600000
0xffffcd40:     0x00000000      0xffffcd60      0x00000000      0xf7d96cb9
(gdb) next
46              leave                   ; Restaura el frame de pila (equ
(gdb) x/32xw $esp
0xffffccd4:     0x00000064      0xffffcd18      0x56556289      0x00000003
0xffffcce4:     0xffffcd08      0x00000000      0x56556218      0x00000000
0xffffccf4:     0xffffcfbb      0x00000003      0x037f0f7f      0x00000020
0xffffcd04:     0x00000000      0x00000004      0xada26f00      0xf7fc7570
0xffffcd14:     0xf7fa2e34      0xffffcd48      0x565561fc      0x40600000
0xffffcd24:     0x00000000      0x00000000      0x565561e3      0x00000000
0xffffcd34:     0x00000000      0x00000000      0x40600000      0x00000000
0xffffcd44:     0xffffcd60      0x00000000      0xf7d96cb9      0x00000000
(gdb) next
47              ret
(gdb) x/32xw $esp
0xffffccdc:     0x56556289      0x00000003      0xffffcd08      0x00000000
0xffffccec:     0x56556218      0x00000000      0xffffcfbb      0x00000003
0xffffccfc:     0x037f0f7f      0x00000020      0x00000000      0x00000004
0xffffcd0c:     0xada26f00      0xf7fc7570      0xf7fa2e34      0xffffcd48
0xffffcd1c:     0x565561fc      0x40600000      0x00000000      0x00000000
0xffffcd2c:     0x565561e3      0x00000000      0x00000000      0x00000000
0xffffcd3c:     0x40600000      0x00000000      0xffffcd60      0x00000000
0xffffcd4c:     0xf7d96cb9      0x00000000      0xffffce14      0xf7db013d
```

```
ESP = 0xffffccd0
0xffffccd0 → EBX (restaurado)
0xffffccd4 → antiguo EBP
0xffffccd8 → nuevo EBP
0xffffccdc → return address
0xffffcce0 → parámetro 1
```

```
0xffffcce4 → parámetro 2
```

leave realiza internamente:
- mov esp, ebp
- pop ebp

ret utiliza el valor en [esp] como dirección de retorno, restaurando la ejecución al punto siguiente al call.

5. Estado de la pila al volver a convert_and_increment (breakpoint 6)

Al retornar de suma_uno, el estado de la pila es:

```
0xffffccf0:     0x00000000      0xffffcfbb      0x00000003      0x037f0f7f
```

```
ESP = 0xffffccf0
0xffffccf0 → valores anteriores al call
```

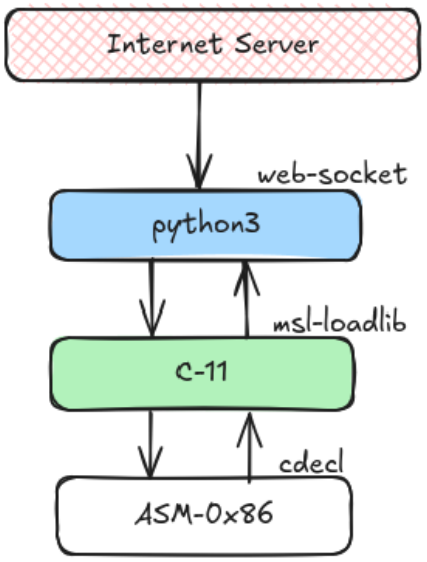
El registro EAX contiene el valor 2, que es el resultado del truncamiento del float original (3.5 → 3) y no se actualizó tras el llamado a suma_uno, ya que este modifica directamente el valor apuntado por el puntero (&ivalue). Puede observarse el valor final en memoria apuntada por EBX o directamente accediendo a ivalue.

Conclusión

Este análisis confirma el comportamiento esperado de la convención de llamada cdecl:
- Los parámetros se pasan por pila, de derecha a izquierda.
- La dirección de retorno se guarda automáticamente en la pila por la instrucción call.
- La función en ensamblador accede a sus parámetros usando offsets positivos desde EBP.
- El caller (convert_and_increment) se encarga de limpiar la pila si corresponde.

Esta estructura de pila y el uso del marco de pila (EBP) facilitan la depuración, análisis y la interoperabilidad entre C y ASM en sistemas x86 de 32 bits.

La arquitectura del servicio queda definida como una arquitectura de capa, la cual se representa de la siguiente forma:



Se comprende la integración de lenguajes con distintos niveles de abstracción trabajando en conjunto utilizando las bondades de cada uno.