Sistemas de Computación

TP2: Stack Frame

Ingeniería en Computación
Ingeniería Electrónica

Preparado por:

Argañarás, Agustín
Genero, Bruno
Ibañez Sala, Ignacio

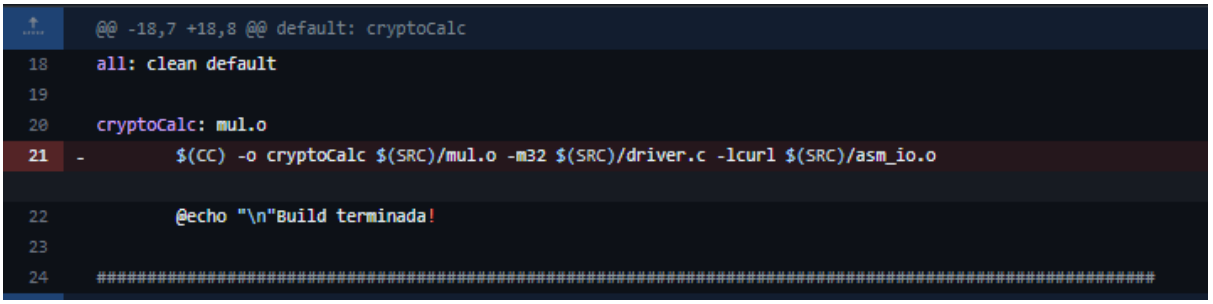
Profesor
Javier Alejandro Jorge

Objetivos

Se debe diseñar e implementar una Calculadora de Cotización de Criptomonedas. La capa superior recuperará la cotización de al menos dos criptomonedas que pueden ser obtenidas de alguna de "Las 12 mejores API del mercado de valores para crear productos financieros". Se recomienda el uso de API Rest y Python. Los datos de consulta realizados deben ser entregados a un programa en C que convocará rutinas en ensamblador para que hagan los cálculos de conversión y devuelvan las cotizaciones en pesos, u$s y euros. Luego el programa en C o Python mostrará los cálculos obtenidos.
El repositorio del proyecto está en el siguiente link:
https://github.com/generobruno/CryptoCalculator

Implementación

Para realizar el trabajo en cuestión decidimos usar C para programar los procesos que llaman a la API y hacen interfaz con otro código en ASM, para poder utilizar el stack de programa visto en clases.
En un principio habíamos intentado realizar todas las funciones en un sólo archivo .c, el cual se encargaría de obtener la información necesaria de una API, utilizando la librería “libcurl”, y luego utilizaría una convención de llamadas para utilizar el código en ASM que se encargaría de realizar el cálculo deseado. El archivo Makefile de esta implementación se veía de la siguiente manera:



Como se puede apreciar, vinculamos los 3 objetos diferentes (.c, .asm y lcurl) en un solo binario ejecutable final. La opción “-m32” se utiliza para utilizar la versión de 32 bits de GCC al compilar “driver.c”, lo cual es necesario ya que el objeto “mul.o” está codificado en ASM de 32 bits. Sin embargo, la librería utilizada para hacer las requests está hecha en 64 bits, por lo que obtenemos un error al compilar.

```
bruno@BrunoLaptop:~/Documents/Facultad/5to_anio/Sistemas de Computacion/TP2/Cryp
toCalculator/c$ gcc -o cryptoCalc src/asm/mul.o -m32 src/driver/driver.o -lcurl
src/asm/asm_io.o
/usr/bin/ld: skipping incompatible /usr/lib/x86_64-linux-gnu/libcurl.so when sea
rching for -lcurl
/usr/bin/ld: skipping incompatible /usr/lib/x86_64-linux-gnu/libcurl.a when sear
ching for -lcurl
/usr/bin/ld: cannot find -lcurl: No such file or directory
/usr/bin/ld: skipping incompatible /usr/lib/x86_64-linux-gnu/libcurl.so when sea
rching for -lcurl
collect2: error: ld returned 1 exit status
```

Por esta razón se decidió realizar 2 ejecutables distintos, uno para obtener la información de internet utilizando libcurl (requester), y otro compilado en 32 bits para realizar el cálculo con ASM (driver).

```
all: clean default

cryptoCalc: mul.o request.o driver.o
	$(CC) -o cryptoCalc -m32 $(DRIVER)/driver.o $(ASM)/asm_io.o $(ASM)/mul.o
	$(CC) -o requester $(REQS)/request.o $(UTILS)/cJSON.o -lcurl
	@echo "\n"Build terminada!
```

Al hacer esto no tenemos ningún error de compilación y los ejecutables se construyen exitosamente.
El problema ahora era cómo comunicarle al proceso que realizaba los cálculos la información necesaria obtenida de la API. Al tratarse de 2 procesos no relacionados debimos hacer uso de un mecanismo de IPC (Inter-Process Communication). Por ello,

decidimos utilizar una FIFO (o named pipe) que envíe el número obtenido desde requester para que luego driver lo lea. El flujo de trabajo es el siguiente:

1. Ejecutamos driver.c (./cryotoCalc + args). Este proceso crea la FIFO:

```c
/**
 * @brief Crea una FIFO para comunicarse con el proceso
 * que obtiene la tasa de conversión de la cryptomoneda
 */
void createFIFO(void) {
    // File descriptor y ubicación para la FIFO
    char *fifo = "/tmp/fifoRequest";

    // Estructura de control
    struct stat stats;

    // Eliminamos la FIFO si ya existía
    if(stat(fifo, &stats) < 0) {
        if(errno != ENOENT) {
            perror("Fallo en stat().");
            exit(1);
        }
    } else {
        if(unlink(fifo) < 0) {
            perror("Error eliminando la FIFO anterior.");
            exit(1);
        }
    }

    // Creamos la FIFO
    if(mkfifo(fifo, 0666) == -1) {
        fprintf(stderr,"Error creando la FIFO (B).");
        exit(1);
    }

    printf("FIFO Creada\n");
}
```

2. Con la FIFO creada, driver crea un hijo que ejecuta requester y se queda esperando a que dicho hijo envíe el mensaje (recvVal()).

```c
// Hacemos que un hijo ejecute la funcion de request
pid_t child = fork();

switch (child)
{
case -1:    // Caso de error
    perror("Error en fork");
    exit(1);
case 0:     // Caso del hijo
    execvp("./requester", argv); // TODO Revisar: De esta manera solo funca ejecutando en CryptoCalculator/c.
    exit(errno);
default:    // Caso del padre
    // Espera a que el hijo termine
    rate = recvVal();
    removeFIFO();
    wait(NULL);
    break;
}
```

3. Por su parte, requester hace el pedido a la API, parsea el json obtenido para leer el resultado, y envía dicho valor a la FIFO.

```c
/**
 * @brief Envia la información al proceso principal
 * haciendo uso de una FIFO
 *
 * @param msg Mensaje a enviar
 */
void sendVal(char *msg) {
    // File descriptor y ubicación para la FIFO
    int fd;
    char *fifo = "/tmp/fifoRequest";
    char message[MSG_MAX];

    // Abrimos el archivo
    fd = open(fifo, O_WRONLY | O_NONBLOCK);

    if(fd == -1) {
        if(errno == ENOENT || errno == ENXIO) {
            fprintf(stdout, "Server Disconnected.\n");
            exit(1);
        }
        perror("Error abriendo la FIFO.");
        exit(1);
    }

    // Formato del mensaje
    memset(message,0,sizeof(message));
    strcpy(message,msg);

    // Escribimos el mensaje en la FIFO
    if(write(fd, message, sizeof(message)) == -1) {
        if(errno == EBADF || errno == ENXIO) {
            fprintf(stdout, "Server Disconnected.\n");
            exit(1);
        } else {
            perror("Error al leer el mensaje (B).\n");
            exit(1);
        }
    }

    /* printf("Data sent: %s",message); */

    // Cerramos el archivo
    if(close(fd) == -1) {
        fprintf(stderr,"Error al cerrar el archivo (B).");
        exit(1);
    }

}
```

4. Cuando driver recibe el valor, elimina la FIFO y la ejecución de su hijo finaliza (figura del punto 2).
5. Con el valor obtenido, se invoca a la función en assembler para realizar el cálculo.

```c
printf("\n\nRate BTC/%s: %f\n", argv[1], rate);
printf("Number to calculate: ");
scanf("%f", &a);

// Hacemos la conversión
mul(a, rate, &result);

printf("\n\nThe result is: $%f \n\n", result);
```

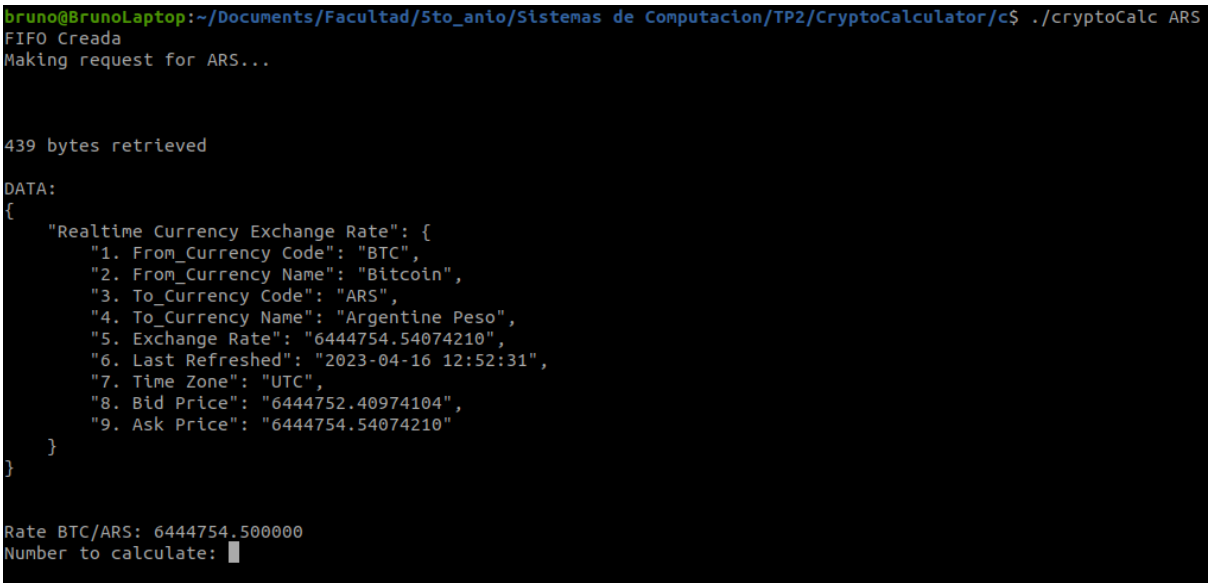
6. El código en ASM utiliza los valores pasados como argumento (a, rate, &result) para crear un stack frame y realizar el cálculo de punto flotante, utilizando el coprocesador matemático.

```
segment .text
        global  mul
mul:

;
; enter 0,0
;

        push    ebp
        mov     ebp, esp
        push    ebx


        dump_stack          1, 2, 4                     ; print out stack from ebp-8 to ebp+16
        fld                 dword [ebp+8]               ; stack: a
        fld                 dword [ebp+12]              ; stack: b, a
        fmulp               st1                         ; stack: a*b
        mov                 ebx, dword [ebp+16]         ; store in ebx direction of res
        fstp                dword [ebx]                 ; store (in res) and pop stack
        dump_regs 1                                     ; dump out register values

;
; leave
;

        pop     ebx
        mov     esp, ebp
        pop     ebp

        ret
```

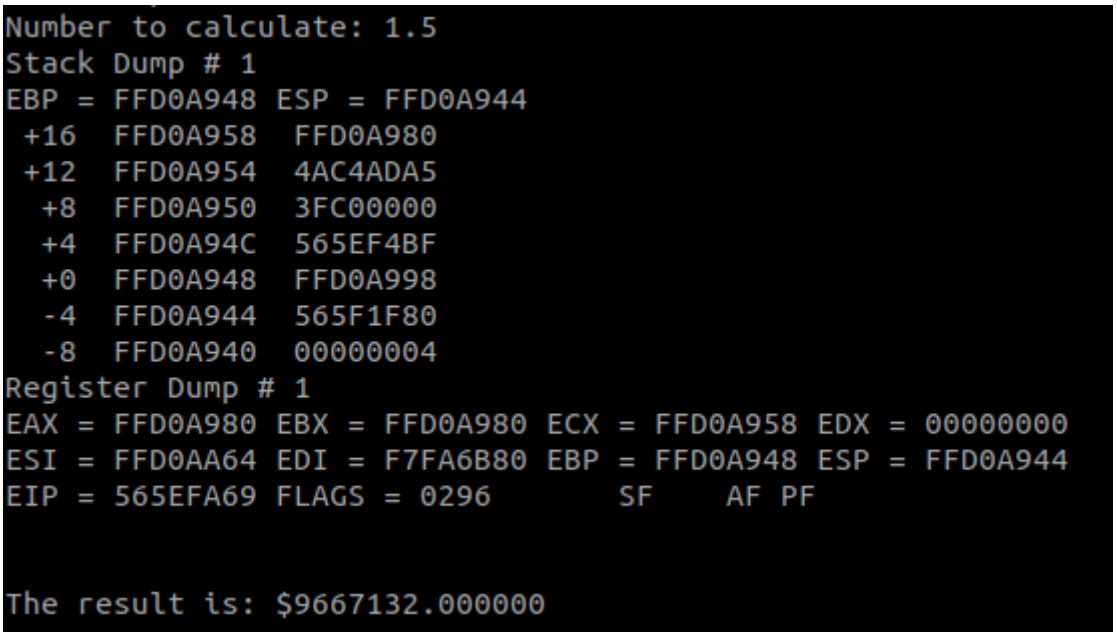
7. Finalmente, se imprime por pantalla el resultado obtenido (figura del punto 5).

Ejecución

La ejecución del programa se muestra a continuación, haciendo una conversión de pesos argentinos a BTC. Primero se obtiene este valor de conversión, y el programa espera a que le pasemos el número de criptomonedas que queremos calcular.



Escribimos con el teclado el número deseado y se realizará el cálculo.



El resto de valores que aparecen por pantalla son las direcciones de memoria asociadas al stack. Se puede apreciar como el stack frame de la función mul() en ASM crece para tener los 3 argumentos que pasamos (+8 a +16) y la dirección de memoria de retorno (+4). Los valores negativos se muestran aunque no son utilizados, ya que se usan para variables locales en la función llamada.

Análisis del Stack

Para poder analizarlo con gdb, compilamos los códigos con la bandera -g que agrega la información para el debug.
Hacemos un break point antes de ejecutar la función en assembler para poder analizar su funcionamiento. Inicialmente podemos observar que valores y que direcciones tienen las variables que serán pasadas como parámetro a la función de ensamblador, y los diferentes registros del procesador.

```
(gdb) next
Number to calculate: 1.0
68          mul(a, rate, &result);
(gdb) info registers
eax            0x1                 1
ecx            0xffffcd78          -12936
edx            0x0                 0
ebx            0x56558f80          1448447872
esp            0xffffcd80          0xffffcd80
ebp            0xffffcdb8          0xffffcdb8
esi            0xffffce84          -12668
edi            0xf7ffcb80          -134231168
eip            0x565564a6          0x565564a6 <main+441>
eflags         0x282               [ SF IF ]
cs             0x23                35
ss             0x2b                43
ds             0x2b                43
es             0x2b                43
fs             0x0                 0
gs             0x63                99
(gdb)
```

```
(gdb) print a
$1 = 1
(gdb) print rate
$2 = 27693.6504
(gdb) print &result
$3 = (float *) 0xffffcda0
(gdb) print &a
$4 = (float *) 0xffffcd9c
(gdb) print &rate
$5 = (float *) 0xffffcda8
```

Se puede observar como EBP y ESP no están aún igualados ya que no entramos a la rutina todavía. Luego de ingresar a la función en assembler y ejecutar el prólogo de la rutina veremos cómo ESP se reduce en para hacer lugar a los parámetros que pasamos y ahora EBP y ESP si están referenciados.

```
(gdb) step
47              push    ebx
(gdb) info reg
eax            0xffffcda0          -12896
ecx            0xffffcd78          -12936
edx            0x0                 0
ebx            0x56558f80          1448447872
esp            0xffffcd68          0xffffcd68
ebp            0xffffcd68          0xffffcd68
esi            0xffffce84          -12668
edi            0xf7ffcb80          -134231168
eip            0x56556a53          0x56556a53 <mul+3>
eflags         0x292               [ AF SF IF ]
cs             0x23                35
ss             0x2b                43
ds             0x2b                43
es             0x2b                43
fs             0x0                 0
gs             0x63                99
(gdb)
```