SANTIAGO AYECHU GÁRRIZ

Práctica 4ª y 5ª: Llamadas al Sistema Operativo y Subrutinas

ÍNDICE

Introducción

El objetivo de esta práctica es ver algunas de las diferentes formas de realizar las llamadas al sistema operativo.

El programador de ensamblador no puede acceder directamente al HW de la máquina. Es por ello por lo que accede, indirectamente, a través del Kernel solicitando operaciones de entrada/salida al Sistema Operativo.

Esta llamada tiene diferentes propósitos ya que, por ejemplo, si deseamos imprimir algo por pantalla necesitamos realizar una llamada al sistema operativo.

Por otro lado, también se llevará a cabo, a lo largo de esta práctica, la llamada a subrutinas (como es el caso de sumMtoN), siguiendo el convenio de llamada (pase de parámetros, valor de retorno, dirección de retorno..), así como la generación de un nuevo frame.

Desarrollo

Módulo `syscall_write_puts.c`

Para este primer caso lo que se pide es la construcción de un programa en lenguaje de programación C. Este programa debe imprimir por pantalla el mensaje de bienvenida "Hola" usando las funciones `puts()`, `write()` y `syscall()`.

Se adjunta la documentación de cada una de las funciones a continuación:

```
int *puts(const char *s);

ssize_t write(int fd, const void *buf, size_t count);

long syscall(long number, ...);
```

Para utilizar la función `puts()` es necesario incluir la librería `<stdio.h>`

En cambio, para la utilización de la función `write()` es necesario incluir la librería `<unistd.h>`

En el caso de la función `syscall()` es necesario tanto la librería `<unistd.h>` como `<sys/syscall.h>`, en la que se encuentran las definiciones de las constantes SYS.

```
/*
    Programa syscall_write_puts.c
    Descripción: Realiza la llamada al sistema operativo para imprimir en la
pantalla
    Realiza la llamada de tres formas diferentes: puts, write, syscall.
    Compilación: gcc -m32 -g -o syscall_write_puts syscall_write_puts.c
*/
```

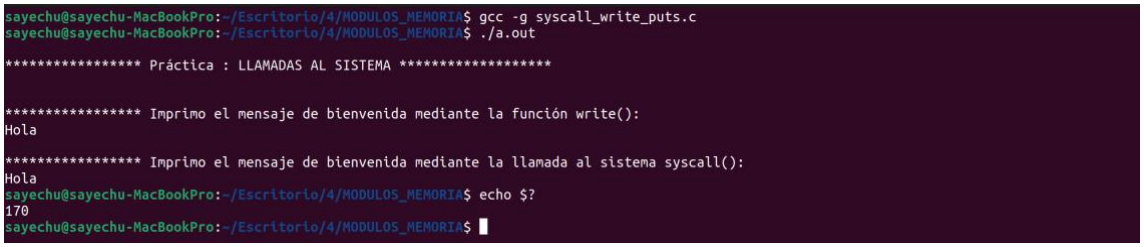
```
// Cabeceras de librerías
#include <stdio.h>        // prototipo de la función puts()
#include <unistd.h> // declaración de las macros STDOUT_FILENO, STDIN_FILENO
#include <sys/syscall.h>  // declaración de la macro __NR_write y __NR_exit
#include <stdlib.h>       // declaración de exit()

// Macros
#define LON_BUF 5         // Tamaño del string
#define __NR_write 1

int main (void)
{
    char buffer[LON_BUF] = "Hola\n";    # variable buffer de tamaño 5

    puts("\n***************** Práctica : LLAMADAS AL SISTEMA
******************\n"); // función puts() de la librería libc
    puts("\n***************** Imprimo el mensaje de bienvenida mediante la
función write(): ");
    write(STDOUT_FILENO, buffer, LON_BUF); // wrapper de la llamada al sistema
write.
    // ya que write() incluye un syscall(), llama indirectamente al sistema

    puts("\n***************** Imprimo el mensaje de bienvenida mediante la
llamada al sistema syscall(): ");
    syscall(__NR_write, STDOUT_FILENO, buffer, LON_BUF); // función syscall de
llamada directa al sistema.
    exit(0xAA); //llamada al s.o para salir envíando el código 0xAA.
}
```

Como se observa en el programa, a la función `puts()` se le pasa como argumento una string. Esta lo que hace es escribir por pantalla dicha string.

En el caso de la función `write()` se le pasa como argumentos el descriptor de fichero (`STDOUT_FILENO`), que es la salida estándar por pantalla, así como el mensaje almacenado en una array `buffer` y, por último, el número de bytes a escribir en la salida estándar.

Por último, en el caso de syscall() se le pasa el interface `__NR_write` y entonces se hace una llamada pasándole los argumentos (salida estándar, mensaje a escribir, numero bytes a escribir).

Al ejecutar el programa se obtiene la siguiente salida, se incluye también el valor devuelto al sistema operativo:



Módulo `syscall_write_puts.s`

Análogamente, se ha desarrollado el módulo fuente `syscall_write_puts.s` equivalente al programa `syscall_write_puts.c`. En este caso, se ha realizado dicho programa en lenguaje ensamblador AT&T para la arquitectura i386. Se han usado las funciones `puts()` y `write()`.

Cabe mencionar que para las funciones usadas en este programa se necesita revisar la documentación de las funciones del apartado anterior.

```
### Programa: syscall_write_puts.s
### Descripción: Programa equivalente a syscall_write_puts.c en lenguaje
esamblador x86-32.
###     El programa llama a las funciones puts() y write() de la libreria libc
para imprimir mensaje.
### Compilación: (en toolchain automatico no es necesario indicar al linker el
módulo objeto libc. en toolchain manual si)
 ### TOOLCHAIN AUTOMATICO: gcc -m32 -g -o syscall_write_puts
syscall_write_puts.s
 ### TOOLCHAIN MANUAL:
    # 1. as --32 -gstabs -o syscall_write_puts.o syscall_write_puts.s
    # 2. ld -melf_i386 -dynamic-linker /lib32/ld-linux.so.2 -o
syscall_write_puts syscall_write_puts.o -lc


    ## MACROS
    .equ LON_BUF, 5        # LON_BUF : valor 5 (longitud de mensaje3)
    .equ WRITE, 4          # __NR_write : valor 4
    .equ STDOUT, 1         # STDOUT_FILENO : valor 1
    .equ SYS_EXIT, 1       # ARGUMENTO REGISTRO EAX
    .equ SUCCESS, 0        # ARGUMENTO REGISTRO EBX

    .section .rodata       # section ReadOnlyDATA -> RODATA

mensaje1:
    .asciz "Imprimo el mensaje de bienvenida mediante la funcion puts()"

mensaje2:
    .asciz "Imprimo el mensaje de bienvenida mediante la función write()"

mensaje3:
    .asciz "Hola\n"
```

```
    .global _start          # definición punto entrada programa: _start
    .section .text          # seccion de instrucciones

_start:                     # etiqueta _start
    # imprimir "cabecera"   # imprimir cabecera programa -> mensaje1
    push $mensaje1          # apilar dirección mensaje1
    call puts               # llamada función puts() con argumento en pila

    # llamada a la función puts de la librería libc. Es necesario linkar con
libc.
    push $mensaje3          # apilar dirección mensaje3
    call puts               # llamada función puts() con argumento en pila

    # imprimir "cabecera"   # imprimimos cabecera -> mensaje2
    push $mensaje2          # apilar dirección mensaje2
    call puts               # llamada función puts() con argumento en pila

    # llamada a la función write de la librería libc. Es necesario linkar con
libc.
    push $LON_BUF           # apilar número de bytes a escribir
    push $mensaje3          # apilar mensaje a escribir -> mensaje3
    push $WRITE             # apilar descriptor de fichero
    call write              # llamada a función write() con argumentos de pila

    # llamada al sistema operativo para ejecutar la operación write
    mov $WRITE, %eax        # mover macro a registro EAX
    mov $STDOUT, %ebx       # mover descriptor de fichero a registro EBX
    mov $mensaje3, %ecx     # mover mensaje a registro ECX
    mov $LON_BUF, %edx      # mover nº bytes a registro EDX
    int $0x80               # llamada al s.o para ejecutar subr. según valor EAX

    # llamada al sistema operativo para ejecutar la operación exit
    mov $SYS_EXIT, %eax     # codigo de llamada al s.o: subrutina exit
    mov $SUCCESS, %ebx      # argumento de salida al s.o a través de EBX según
convenio ABI i386
    int $0x80       # llamada al s.o para ejecutar subrutina según valor EAX

    .end
```

Al ejecutar el programa se obtiene la siguiente salida, se incluye también el valor devuelto al sistema operativo:

```
sayechu@sayechu-MacBookPro:~/Escritorio/MEMORIA P4$ gcc -m32 -g -nostartfiles -o syscall_write_puts syscall_write_puts.s
/usr/bin/ld: /tmp/ccxd2g6u.o: warning: relocation in read-only section `.text'
/usr/bin/ld: warning: creating DT_TEXTREL in a PIE
sayechu@sayechu-MacBookPro:~/Escritorio/MEMORIA P4$ ./syscall_write_puts
Imprimo el mensaje de bienvenida mediante la funcion puts()
Hola

Imprimo el mensaje de bienvenida mediante la función write()
Hola
sayechu@sayechu-MacBookPro:~/Escritorio/MEMORIA P4$ echo $?
0
sayechu@sayechu-MacBookPro:~/Escritorio/MEMORIA P4$
```

Módulo `sumMtoN_aviso.c`

Dado un programa en lenguaje ensamblador AT&T, se pide programar el equivalente en lenguaje C (`sumMtoN_aviso.c`). Como su nombre indica, este programa realiza la suma de la serie `M, M+1, M+2, M+3, .., N`, siendo estos valores (N, M) dos valores fijos (5 y 10, respectivamente). La diferencia entre este programa y el equivalente es que se le ha añadido una "comprobación" de que se cumple la propiedad M<N.

Cabe destacar que, estos dos valores, deben cumplir la propiedad M < N.

Se adjunta código del programa `sumMtoN_aviso.c` a continuación:

```
// Program: sumMtoN_aviso.c
/*
   Descripción: suma de serie M, M+1, M+2, M+3, .., N
       función sumMtoN (1° arg = m, 2° arg = n);
   Compilación: gcc -o sumMtoN_aviso sumMtoN_aviso.c
*/
#include <stdio.h>
#include <stdlib.h>

int sumMtoN (int, int);    # declaración / definición función sumMtoN

int main(void)
{
    int m = 5;             # declaración e inicialización, M = 5
    int n = 10;            # declaración e inicialización, N = 5
    int ebx; # declaración EBX -> donde se almacena return subrutina sumMtoN

    if (m > n)      # si no cumple condición m <= n, imprimir error, salir.
    {
        perror("Los argumentos no cumplen la propiedad m <= n");
        exit(0);
    }
    ebx = sumMtoN(m, n);   # llamada a subrutina sumMtoN con parámetros m y n
    exit(ebx);             # devolver al sistema resultado de la suma de serie
}
int sumMtoN (int a, int b)
{
    int edx = 0;           # variable donde se guarda suma iterativa
    int eax = b-a;         # numero de iteraciones a realizar
    while (eax >= 0)
    {
        edx = edx + a;     # EDX <- EDX + A;
        a = a + 1;         # M+1, M+2 hasta N
        eax = eax - 1;     # Avanzar a la terminación del bucle
    }
    return edx;            # devolver suma de serie
}
```

Tras compilar y ejecutar este último programa se observa que se obtiene el mismo resultado que en el programa equivalente en lenguaje ensamblador. Se adjunta foto de compilación, ejecución y print del valor devuelto al sistema operativo tras su ejecución:

Como se puede comprobar realizando los cálculos correspondientes, el resultado es correcto ya que, habiendo fijado M en 5 y N en 10, la serie queda de la siguiente manera: 5, 6, 7, 8, 9, 10. Y, por tanto, realizando la suma de esta serie queda como resultado 45, que coincide con el calculado por el programa `sumMtoN_aviso.c`.

Módulo `sumMtoN_aviso.s`

A continuación, se pide añadir, al programa `sumMtoN.s`, la comprobación de la condición, mencionada ya anteriormente, que los parámetros deben cumplir: M < N. Por tanto, tras realizar dichos cambios, el programa `sumMtoN_aviso.s` queda de la siguiente manera:

```
/*                                  CABECERA
    Programa: sumMtoN.s
    Descripción: realiza la suma de números enteros de la serie
M,M+1,M+2,M+3,...N
        función : sumMtoN(1° arg=M, 2° arg=N) donde M < N
    Ejecución: Editar los valores M y N y compilar el programa.
        Ejecutar $./sumMtoN
        El resultado de la suma se captura del sistema operativo con el
comando linux: echo $?
        gcc -nostartfiles -m32 -g -o sumMtoN sumMtoN.s
        Ensamblaje as --32 --gstabs sumMtoN.s -o sumMtoN.o
        linker -> ld -melf_i386 -o sumMtoN sumMtoN.o
*/
    ## MACROS
    .equ SYS_EXIT, 1        # definición de MACROS

    ## DATOS                # sección de datos
    .section .data
M:                          # declaración variable M -> valor 5
    .int 5
N:                          # declaración variable N -> valor 10
    .int 10

ERROR:                      # mensaje de error en caso de no cumplir condición
    .asciz "PARAMETROS NO INTRODUCIDOS CORRECTAMENTE, M < N"

    ## INSTRUCCIONES
    .section .text          # seccion de instrucciones
```

```
    .globl _start          # definición punto entrada programa->etiqueta _start
_start:                    # punto entrada programa -> etiqueta _start
    mov N, %eax            # mover N a registro EAX
    cmp M, %eax            # comparar N y M -> N-M
    js msg                 # si negativo -> ERROR -> N < M -> salto a MSG

    ## Paso los dos argumentos M y N a la subrutina a través de la pila
    pushl N                # apilar segundo argumento -> N
    pushl M                # apilar primer argumento -> M

    ## Llamada a la subrutina sum1toN
    call sumMtoN           # llamar a subrutina sumMtoN con argumentos en pila

    ## Paso la salida de sum11toN al argumento a la llamada al sistema exit()
    mov %eax, %ebx         # resultado suma serie de sumMtoN en registro EAX
cont:
    ## Código de la llamada al sistema operativo
    movl $SYS_EXIT, %eax   # código de llamada al s.o: subrutina exit

    ## Interrumpo al S.O.
    int $0x80       # llamada al s.o para ejecutar subrutina según valor EAX

msg:                # codigo a ejecutar en caso de no cumplirse condicion M < N
    push $ERROR     # apilar mensaje de ERROR
    call puts       # llamar función puts() para imprimir mensaje ERROR en pila
    movl $0, %ebx   # devolver al Sistema operativo valor 0
    jmp cont        # salto a cont para finalizar programa

/*                          CODIGO DE SUBRUTINA sumMtoN
    Subrutina: sumMtoN
    Descripción: calcula la suma de números enteros en secuencia desde el 1°
sumando hasta el 2° sumando
        Argumentos de entrada: 1° sumando y 2° sumando
        los argumentos los pasa la rutina principal a través de la pila:
        1° se apila el último argumento y finalmente se apila el 1° argumento.
        Argumento de salida: es el resultado de la suma y se pasa a la rutina
principal a través del registro EAX.
        Variables locales: se implementa una variable local en la pila pero no
se utiliza
*/
    .type sumMtoN, @function     # declara la etiqueta sumMtoN

sumMtoN:
    ## Prólogo: Crea el nuevo frame del stack
    pushl %ebp             # salvar el frame pointer antiguo
    movl %esp, %ebp        # actualizar el frame pointer nuevo

    ## Reserva una palabra en la pila como variable local
    ## Variable local en memoria externa: suma
    subl $4, %esp          # ESP apunte 4 direcciones de memoria menos
```

```
    ## Captura de argumentos
    movl 8(%ebp), %ebx          # 1° argumento copiado en %ebx
    movl 12(%ebp), %ecx         # 2° argumento copiado en %ecx

    ## suma la secuencia entre el valor del 1°arg y el valor del 2°arg
    ## 1° arg < 2°arg
    ## utilizo como variable local EDX en lugar de la reserva externa para
variable local: optimiza velocidad

    ## Inicializo la variable local suma
    movl $0,%edx          # inicializar a 0 la variable que guarda la suma

    ## Número de iteracciones
    mov %ecx,%eax         # mover ECX -> 2° argumento (M) -> registro EAX
    sub %ebx,%eax         # EBX(N)-EAX(M) = 10-5 = 5 -> 0…5 -> 6 iteraciones

bucle:
    add %ebx,%edx         # EDX <- EDX + EBX | EDX <- EDX + M (M+1)…
    inc %ebx              # M -> M+1 -> M+2 … N
    sub $1,%eax           # número iteraciones = n. iteraciones - 1
    jns bucle             # si número iteraciones >= 0 -> salto a bucle

    ## Salvo el resultado de la suma como el valor de retorno
    movl %edx, %eax       # guardar valor de suma M a N en registro EAX

    ## Epílogo: Recupera el frame antiguo
    movl %ebp, %esp       # restauro el stack pointer
    popl %ebp             # restauro el frame pointer

    ## Retorno a la rutina principal
    ret                   # volver a rutina principal
    .end            # FIN DE PROGRAMA
```

El funcionamiento de este último programa es igual que el programa sumMtoN.s, solo que se le ha añadido la comprobación de que se cumpla la condición M < N. La explicación de la pila y como se almacenan las variables en la pila se encuentra en el apartado del programa sumMtoN.s.

Módulos Fuente Comentados

A continuación, se muestran los módulos fuente llevados a cabo en esta práctica con sus respectivos comentarios en rojo. Más adelante en la sección de 'Historial Comandos GDB + Salida' se verá la depuración de estos programas.

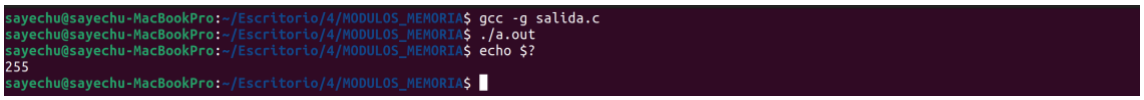
`salida.c`

En este primer programa se incluye la librería STDLIB, la cual es necesaria para la llamada a la función `exit()`.

```
#include <stdlib.h>       #include de librería STDLIB -> declaración exit()
```

```
void main (void)
{
    exit (0xFF);    # Salir al sistema enviando el código 0xFF.
}
```

Cuando se ejecuta la instrucción `exit(0xFF)` acaba el programa y se devuelve al sistema operativo el valor 0xFF, que equivale al 255 en decimal. De esta forma, se muestra a continuación lo que el programa devuelve al sistema operativo al ejecutar dicho programa a continuación:



salida1.c

En este segundo programa se realiza la llamada al sistema operativo con la función syscall(). Equivale al programa de salida.c pero en vez de usar la función exit(), usamos la llamada a syscall(). Se muestra la documentación de ambas funciones.

> The `exit()` function causes normal process termination and the least significant byte of status (i.e., status & 0xFF) is returned to the parent.

> `syscall()` is a small library function that invokes the system call whose assembly language interface has the specified number with the specified arguments. Employing `syscall()` is useful, for example, when invoking a system call that has no wrapper function in the C library.
>
> `syscall()` saves CPU registers before making the system call, restores the registers upon return from the system call.

```
/*
    Llamada al sistema desde C
    Prototipo: int syscall(int number, ...);
    man syscall
*/

#define _GNU_SOURCE

#include <unistd.h>
#include <sys/syscall.h> // declaración de la macro __NR_write y __NR_exit

void main (void)
{
    syscall(__NR_exit, 0xFF);    #llamada al sistema con syscall
}
```

Al igual que en el caso del programa que usaba la función `exit()` en este caso se hace una llamada a la subrutina especificada en los parámetros y además se le pasa el código

a devolver al sistema operativo `0xFF`. Una vez ejecutado el programa si se ejecuta el comando `echo $?` se podrá el valor devuelto por dicho programa al sistema operativo.

```
sayechu@sayechu-MacBookPro:~/Escritorio/4/MODULOS_MEMORIA$ gcc -g salida1.c
sayechu@sayechu-MacBookPro:~/Escritorio/4/MODULOS_MEMORIA$ ./a.out
sayechu@sayechu-MacBookPro:~/Escritorio/4/MODULOS_MEMORIA$ echo $?
255
sayechu@sayechu-MacBookPro:~/Escritorio/4/MODULOS_MEMORIA$
```

salida.s

Una vez visto las llamadas al sistema operativo en el lenguaje de programación de alto nivel C, se procede a realizar diferentes programas equivalentes en el lenguaje de programación de bajo nivel AT&T.

```
        .global _start      # declaración punto entrada -> etiqueta _start
        .section .text      # sección de instrucciones:
_start:
        push $0xFF          # código a devolver cuando se hace llamada al s.o.
        call exit           # llamada al s.o. devuelve código 0xFF

        .end                # fin del programa
```

Lo que hacemos es apilar los argumentos en la Stack de tal forma que cuando hacemos la llamada a la subrutina exit, se ejecuta dicha subrutina con los valores de la Stack. Al igual que en casos anteriores, si imprimimos el valor devuelto al sistema operativo tras ejecutar el programa se obtiene el valor apilado inicialmente.

```
sayechu@sayechu-MacBookPro:~/Escritorio/4/MODULOS_MEMORIA$ gcc -m32 -g -nostartfiles -o salida salida.s
/usr/bin/ld: /tmp/cc4MjpS8.o: warning: relocation against `exit@@GLIBC_2.0' in read-only section `.text'
/usr/bin/ld: warning: creating DT_TEXTREL in a PIE
sayechu@sayechu-MacBookPro:~/Escritorio/4/MODULOS_MEMORIA$ ./salida
sayechu@sayechu-MacBookPro:~/Escritorio/4/MODULOS_MEMORIA$ echo $?
255
sayechu@sayechu-MacBookPro:~/Escritorio/4/MODULOS_MEMORIA$
```

salida1.s

`salida1.s` es un programa equivalente a `salida.s` pero en vez de usar la llamada a la función `exit`, se usa la llamada a la función `syscall`. En este caso es necesario especificar un parámetro más que en el caso anterior ya que necesitamos especificar a qué subrutina (`exit`) deseamos llamar, cuyo código es 1.

```
        .global _start      # declaración punto entrada -> etiqueta _start
        .section .text      # sección de instrucciones:
_start:
        push $0xFF          # codigo a devolver cuando se hace llamada al s.o.
        push $1             # exit syscall code
        call syscall        # llamada al sistema con códigos apilados.

        .end                # fin del programa
```

Al ejecutar la llamada al sistema para ejecutar la subrutina con código 1, subrutina exit, el programa devuelve el siguiente valor, que es que es se ha apilado inicialmente en la Stack (pila).

```
sayechu@sayechu-MacBookPro:~/Escritorio/4/MODULOS_MEMORIA$ gcc -m32 -g -nostartfiles -o salida1 salida1.s
/usr/bin/ld: /tmp/ccnV17Jn.o: warning: relocation against `syscall@@GLIBC_2.0' in read-only section `.text'
/usr/bin/ld: warning: creating DT_TEXTREL in a PIE
sayechu@sayechu-MacBookPro:~/Escritorio/4/MODULOS_MEMORIA$ ./salida1
sayechu@sayechu-MacBookPro:~/Escritorio/4/MODULOS_MEMORIA$ echo $?
255
sayechu@sayechu-MacBookPro:~/Escritorio/4/MODULOS_MEMORIA$
```

salida2.s

En este caso lo hacemos es hacer una llamada al sistema operativo siguiendo el convenio ABI i386. Lo que hacemos es mover el código de la subrutina a ejecutar al registro EAX mientras que el valor que devuelve al sistema operativo se mueve al registro EBX. Seguidamente se hace una llamada al sistema operativo para ejecutar la subrutina según el valor del registro EAX. En este caso hemos indicado que queremos llamar a la subrutina exit, por lo que se mueve el 1 al registro EAX.

```
        .global _start      # declaración punto entrada -> etiqueta _start
        .section .text      # sección de instrucciones:
_start:
        mov $1, %eax        # codigo de llamada al s.o: subrutina exit
        mov $0xFF, %ebx     # argumento de salida al s.o a través de EBX según
convenio ABI i386
        int $0x80     # llamada al s.o para ejecutar subrutina segun valor EAX

        .end                # fin del programa
```

Como ya se ha indicado anteriormente, en el registro EBX se encuentra el contenido que se devuelve al sistema operativo. Por tanto, cuando ejecutamos el programa e imprimimos el valor devuelto al sistema operativo debería darnos el contenido en el registro EBX (0xFF en hexadecimal, 255 en decimal). Como se observa en la captura, adjuntada a continuación, el programa devuelve dicho código al sistema operativo.

```
sayechu@sayechu-MacBookPro:~/Escritorio/4/MODULOS_MEMORIA$ gcc -m32 -g -nostartfiles -o salida2 salida2.s
sayechu@sayechu-MacBookPro:~/Escritorio/4/MODULOS_MEMORIA$ ./salida2
sayechu@sayechu-MacBookPro:~/Escritorio/4/MODULOS_MEMORIA$ echo $?
255
sayechu@sayechu-MacBookPro:~/Escritorio/4/MODULOS_MEMORIA$
```

imprimir.s

```
        .section .data      # seccion de datos
planet:
  .long 9                   # long = 4 bytes -> variable planet
        .section .rodata    # seccion ReadOnlyDATA
mensaje:
  .asciz "El número de planetas es %d \n" # string con formato printf

        .global _start      # definicion etiqueta _start -> punto entrada prog.
        .section .text      # seccion de instrucciones
_start:
        ## imprimir en la pantalla
        push planet    # 2° argumento de la función printf
        push $mensaje # 1° argumento de la función printf: dirección del string
        call printf  # llamada a rutina printf con argumentos anteriores

        ## salir al sistema
        push $0       # argumento de la función exit
        call exit     # llamada a rutina exit con argumentos
```

En este programa lo que se hace es definir una serie de variables con las que luego se hace la llamada a la función printf() para imprimir por pantalla el contenido de dichas variables.

```
printf("El número de planetas es %d \n ", planet);
```

Así, sabiendo el formato de la función `printf()`, es necesario apilar primero la variable planet y después el mensaje. De tal forma que al desapilar se desapile primeramente el mensaje y después la variable planet, siguiendo el formato de la función `printf()`.

Cabe destacar que cuando hacemos un push de las variables, no estamos apilando el valor de las variables si no la dirección de memoria donde se encuentran.

Finalmente se apila el valor 0 en la Stack y se llama a la función `exit()`, devolviendo este valor (0) al sistema operativo. Se muestra captura de pantalla de la ejecución de dicho programa a continuación:

```
sayechu@sayechu-MacBookPro:~/Escritorio/4/MODULOS_MEMORIA$ gcc -m32 -g -nostartfiles -o imprimir imprimir.s
/usr/bin/ld: /tmp/ccrRi5Du.o: warning: relocation in read-only section `.text'
/usr/bin/ld: warning: creating DT_TEXTREL in a PIE
sayechu@sayechu-MacBookPro:~/Escritorio/4/MODULOS_MEMORIA$ ./imprimir
El número de planetas es 9
sayechu@sayechu-MacBookPro:~/Escritorio/4/MODULOS_MEMORIA$ echo $?
0
sayechu@sayechu-MacBookPro:~/Escritorio/4/MODULOS_MEMORIA$
```

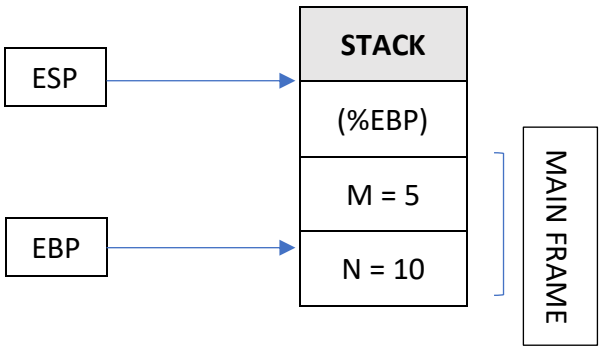
sumMtoN.s

```
/*
    Programa: sumMtoN.s
    Descripción: realiza la suma de números enteros de la serie
M,M+1,M+2,M+3,...N
        función : sumMtoN(1° arg=M, 2° arg=N) donde M < N
    Ejecución: Editar los valores M y N y compilar el programa.
        Ejecutar $./sumMtoN
        El resultado de la suma se captura del sistema operativo con el
comando linux: echo $?
        gcc -nostartfiles -m32 -g -o sumMtoN sumMtoN.s
        Ensamblaje as --32 --gstabs sumMtoN.s -o sumMtoN.o
        linker -> ld -melf_i386 -o sumMtoN sumMtoN.o
*/
    ## MACROS              # definición de MACROS
    .equ SYS_EXIT, 1       # código subrutina exit

    ## DATOS
    .section .data         # seccion de datos

    ## INSTRUCCIONES
    .section .text         # seccion de instrucciones
    .globl _start          # definición punto entrada programa -> _start

_start:                    # etiqueta _start -> punto entrada programa
```

```
    ## Paso los dos argumentos M y N a la subrutina a través de la pila
    pushl $10              # apilar segundo argumento -> N
    pushl $5               # apilar primer argumento -> M

    ## Llamada a la subrutina sum1toN
    call sumMtoN           # llamada a subrutina sumMtoN

    ## Paso la salida de sum11toN al argumento a la llamada al sistema exit()
    mov %eax, %ebx         # registro EAX <- return de sumMtoN
                           # devolver al Sistema en registro EBX, solución suma
    ## Código de la llamada al sistema operativo
    movl $SYS_EXIT, %eax   # código de subrutina exit

    ## Interrumpo al S.O.
    int $0x80 # llamada sistema operativo ejecutar subrutina valor reg. EAX

/*
    Subrutina: sumMtoN
    Descripción: calcula la suma de números enteros en secuencia desde el 1°
sumando hasta el 2° sumando
        Argumentos de entrada: 1° sumando y 2° sumando
        los argumentos los pasa la rutina principal a través de la pila:
        1° se apila el último argumento y finalmente se apila el 1° argumento.
        Argumento de salida: es el resultado de la suma y se pasa a la rutina
principal a través del registro EAX.
        Variables locales: se implementa una variable local en la pila pero no
se utiliza
*/
    .type sumMtoN, @function     # declara la etiqueta sumMtoN

sumMtoN:                         # etiqueta sumMtoN
    ## PróÍogo: Crea el nuevo frame del stack
    pushl %ebp                   # guardamos dirección a la que apunta EBP
    movl %esp, %ebp              # actualizar el frame pointer nuevo

    ## Reserva una palabra en la pila como variable local
    ## Variable local en memoria externa: suma
    subl $4, %esp                # reservar espacio para guardar contenido

    ## Captura de argumentos
    movl 8(%ebp), %ebx           # 1° argumento copiado en %ebx
    movl 12(%ebp), %ecx          # 2° argumento copiado en %ecx

    ## suma la secuencia entre el valor del 1°arg y el valor del 2°arg
    ## 1° arg < 2°arg
    ## utilizo como variable local EDX en lugar de la reserva externa para
variable local: optimiza velocidad

    ## Inicializo la variable local suma
    movl $0, %edx          # registro EDX donde se guarda las sucesivas sumas
```

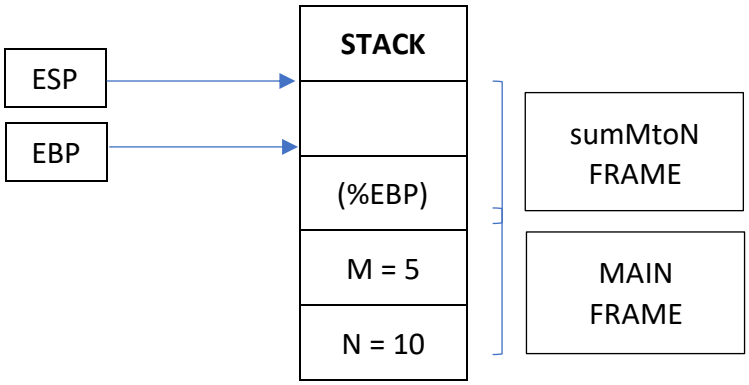
```
    ## Número de iteracciones
    mov %ecx,%eax
    sub %ebx,%eax          # 10-5 = 5 -> 0..5 -> 6 iteraciones = 5,6,7,8,9,10

bucle:
    add %ebx,%edx          # en registro ebx M, M+1, M+2 -> EDX se guarda suma
    inc %ebx               # pasar de M a M+1, de M+1 a M+2.. hasta N
    sub $1,%eax            # decrementar el numero de iteraciones pendientes
    jns bucle              # salto a etiqueta bucle si EAX >= 0

    ## Salvo el resultado de la suma como el valor de retorno
    movl %edx, %eax        # guardar el resultado de suma en registro EAX, ret

    ## Epílogo: Recupera el frame antiguo
    movl %ebp, %esp        # restauro el stack pointer
    popl %ebp              # restauro el frame pointer

    ## Retorno a la rutina principal
    ret                    # return a rutina principal
    .end            # fin del programa
```

Para poder entender el programa anterior es fundamental entender previamente cómo funciona la Stack. Cuando se apilan los dos argumentos (N y M) y se apila la dirección a la que apunta el registro EBP al comenzar la subrutina `sumMtoN`, la pila estaría tal que así:



Cuando se actualiza el frame pointer la pila queda se la siguiente manera:

Cabe destacar que justo antes hacer la llamada a la subrutina sumMtoN, también se apila la dirección de retorno. No se muestra en las representaciones para entenderlos mejor a simple vista.



De tal forma que para acceder a los argumentos (N y M), se usa direccionamiento relativo a base partiendo de la dirección a la que apunte el EBP. Por tanto para acceder al argumento M, partiendo de la dirección de EBP, tendríamos que "saltar" la celda donde se encuentra guardada la dirección original del EBP y la celda donde almacena la Return Address. Es por ello que en el direccionamiento se especifica

```
movl 8(%ebp), %ebx
```

En el caso de querer obtener el segundo argumento (N=10), es necesario saltar las celdas anteriores mas la celda del primer argumento (M=5) por tanto sería necesario añadir otras 4 direcciones de memoria en el direccionamiento. La dirección efectiva del operando fuente es la EBP+12.

Cuando se ejecuta el programa se obtiene la siguiente salida:

```
sayechu@sayechu-MacBookPro:~/Escritorio$ gcc -m32 -g -nostartfiles -o sumMtoN sumMtoN.s
sayechu@sayechu-MacBookPro:~/Escritorio$ ./sumMtoN
sayechu@sayechu-MacBookPro:~/Escritorio$ echo $?
45
sayechu@sayechu-MacBookPro:~/Escritorio$
```

Finalmente, cabe recordar que el salto al bucle (jns bucle) se hace dependiendo del valor de los registros de estado EFLAGS, modificados en la última operación (sub $1, %eax)

Se muestra el contenido del registro EFLAGS antes de hacer el salto.

```
(gdb) p $eflags
$6 = [ CF PF AF SF IF ]
(gdb)
```

Como el Sign Flag (SF) está activado, no se cumple la condición del salto por lo tanto en la última iteración no se realiza el salto a la etiqueta bucle y, por tanto, el bucle finaliza.

Comandos de Compilación

A continuación, se muestran los comandos usados para la compilación de los programas sobre los que trata esta práctica.

- Los programas en el lenguaje C (`salida.c`, `salida1.c`), se compilaron haciendo uso de los siguientes comandos:

Para el módulo fuente `salida.c`:

```
gcc -m32 -g -o salida salida.c
```

Para el módulo fuente `salida1.c`:

```
gcc -m32 -g -o salida1 salida1.c
```

Añadiendo los siguientes argumentos:

-m32: módulos fuente y objeto para la arquitectura i386.
-g para cargar tabla de símbolos

Estos comandos nos generan el módulo binario ejecutable (`salida` y `salida1`) listos para ser cargados en memoria.

- En cuanto a los programas codificados en lenguaje ensamblador AT&T para la arquitectura i386, se han usado los siguientes comandos de compilación (haciendo uso del Toolchain automático).

  Cabe destacar que, al no tener un punto de entrada `main`, en todos los comandos de compilación de los programas, se ha añadido `-nostartfiles`.

  Para el módulo fuente `salida.s`:

  ```
  gcc -m32 -g -nostartfiles -o salida salida.s
  ```

  Para el módulo fuente `salida1.s`:

  ```
  gcc -m32 -g -nostartfiles -o salida1 salida1.s
  ```

  Para el módulo fuente `salida2.s`:

  ```
  gcc -m32 -g -nostartfiles -o salida2 salida2.s
  ```

  Para el módulo fuente `imprimir.s`:

  ```
  gcc -m32 -g -nostartfiles -o imprimir imprimir.s
  ```

  Cabe destacar que, para este último módulo fuente imprimir.s, en la compilación no es necesario indicar al linker el módulo objeto libc ya que es enlazada por defecto.

  En caso de querer compilar haciendo uso del Toolchain manual, sí que sería necesario indicar al linker el módulo objeto libc, y, por tanto, se deberían de usar los siguientes comandos de compilación:

```
as --32 -gstabs -o imprimir imprimir.s

ld -melf_i386 -dynamic-linker /lib32/ld-linux.so.2 -o
imprimir imprimir.s -lc
```

Historial Comandos GDB + Salida

A continuación, se muestra los `.txt` generados a partir de la depuración de cada uno de los módulos fuente, ya vistos anteriormente. Cabe recordar que se muestra los comandos de la depuración en blanco mientras que los comentarios añadidos se encuentran en la parte de la derecha en color rojo.

No se muestran la depuración de los primeros programas en lenguaje C ya que dichos programas solo tienen una instrucción que al ejecutar dicha instrucción el programa acaba. Se muestra la depuración de los programas que tengan algo más de contenido.

Se comienza mostrando la depuración del programa `salida.s`, cuyo código se encuentra en la sección de módulos fuente comentados.

```
+file salida                # CARGAR MODULO SALIDA
Reading symbols from salida...
+b _start                   # punto interrupcion etiqueta _start
Punto de interrupción 1 at 0x1020: file salida.s, line 4.
+run                        # empezar depuracion
Starting program: /home/sayechu/Escritorio/4/MODULOS_MEMORIA/salida
[Depuración de hilo usando libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".

Breakpoint 1, _start () at salida.s:4
+n                          # ejecutar siguiente instruccion (push $0xFF)
+x /1xw $esp                # volcar 1 palabra en hexadecimal a partir de ESP
0xffffd1cc:  0x000000ff
+n                          # ejecutar siguiente instruccion (call exit)
0xf7db71c0 in exit () from /lib32/libc.so.6
[Inferior 1 (process 18788) exited with code 0377]
+quit                       # salir depuración programa
```

Depuración `salida1.s`, cuyo código se encuentra en la sección de módulos fuente comentados.

```
+file salida                # cargar modulo salida
Reading symbols from salida...
+b _start                   # punto interrupción etiqueta _start
Punto de interrupción 1 at 0x1020: file salida1.s, line 4.
+run                        # empezar depuración a partir de punto inter.
Starting program: /home/sayechu/Escritorio/4/MODULOS_MEMORIA/salida
[Depuración de hilo usando libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".

Breakpoint 1, _start () at salida1.s:4
+n                          # ejecutar siguiente instrucción (push $0xFF)
```

```
+x /1xw $esp               # volcar 1 word en hexadecimal a partir de ESP
0xffffd1cc:  0x000000ff
+n                         # ejecutar siguiente instrucción (push $1)
+x /2xw $esp               # volcar 2 words en hexadecimal a partir de ESP
0xffffd1c8:  0x00000001    0x000000ff
+n                         # ejecutar siguiente instrucción (call syscall)
0xf7e96bf0 in syscall () from /lib32/libc.so.6
[Inferior 1 (process 18962) exited with code 0377]
+quit                      # salir depuración programa
```

Se muestra depuración `salida2.s`, cuyo código se encuentra en la sección de módulos fuente comentados.

```
+file salida               # cargar modulo salida
Reading symbols from salida...
+b _start                  # punto interrupción en etiqueta _start
Punto de interrupción 1 at 0x1000: file salida2.s, line 4.
+run                       # empezar depuración a paritr de punto interrupcion
Starting program: /home/sayechu/Escritorio/4/MODULOS_MEMORIA/salida

Breakpoint 1, _start () at salida2.s:4
+n                         # ejecutar siguiente instrucción (mov $1, %eax)
+p /x $eax                 # imprimir contenido registro EAX en formato hex.
$1 = 0x1
+n                         # ejecutar siguiente instrucción (mov $0xFF, %ebx)
+p /x $ebx                 # imprimir contenido registro EBX en formato hex.
$2 = 0xff
+n                         # ejecutar siguiente instrucción (int $0x80)
[Inferior 1 (process 19089) exited with code 0377]
+quit                      # salir de depuración
```

Depuración programa `imprimir.s`, disponible código en su respectivo apartado.

```
+file imprimir      #cargar modulo imprimir, compilado incluyendo tabla simbolos
Reading symbols from imprimir...
+b _start           # punto interrupción en etiqueta _start
Punto de interrupción 1 at 0x1030: file imprimir.s, line 12.
+run                # ejecutar depuración de programa imprimir
Starting program: /home/sayechu/Escritorio/4/MODULOS_MEMORIA/imprimir
[Depuración de hilo usando libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".

Breakpoint 1, _start () at imprimir.s:12
+x /1xw &planet     # volcar 1 palabra en hexadecimal a partir de planet
0x56559000:  0x00000009
+p /s (char *) &mensaje    # imprimir en formato string contenido de mensaje
$1 = 0x56557000 "El número de planetas es %d \n"
+n                         # ejecutar siguiente instrucción (push planet)
+x /1xw $esp               # volcar contenido (1w) a partir de Stack Pointer
0xffffd1cc:  0x00000009
+n                         # ejecutar siguiente instrucción (push $mensaje)
+p /a &mensaje             # imprimir dirección de memoria de mensaje
$2 = 0x56557000
+x /2xw $esp               # volcar contenido (2w) a partir de Stack Pointer
0xffffd1c8:  0x56557000    0x00000009
+n                         # ejecutar siguiente instrucción (call printf)
0xf7dd44f0 in printf () from /lib32/libc.so.6
+up                        # volver a programa principal
#1  0x56556040 in _start () at imprimir.s:14
+n
Single stepping until exit from function printf,
which has no line number information.
```

```
_start () at imprimir.s:17
+n                          # ejecutar siguiente instrucción (push $0)
+x /3xw $esp                # volcar contenido (3w) a partir de Stack Pointer
0xffffd1c4:  0x00000000   0x56557000   0x00000009
+n                          # ejecutar siguiente instrucción (call exit)
0xf7db71c0 in exit () from /lib32/libc.so.6
+up                         # Volver a programa principal
#1  0x56556047 in _start () at imprimir.s:18
+n
Single stepping until exit from function exit,
which has no line number information.
[Inferior 1 (process 19367) exited normally]
+quit                       # Salir de depuración
```

Se procede a mostrar la depuración del módulo fuente `syscall_write_puts.c`.

```
+file syscall_write_puts          # cargar modulo syscall_write_puts
Reading symbols from syscall_write_puts...
+b 20                             # breakpoint en linea 20 -> programa main
Punto de interrupción 1 at 0x11c4: file syscall_write_puts.c, line 20.
+run                              # ejecutar depuración programa
Starting program: /home/sayechu/Escritorio/4/MODULOS_MEMORIA/syscall_write_puts
[Depuración de hilo usando libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".

Breakpoint 1, main () at syscall_write_puts.c:20
+n           # ejecutar siguiente instrucción (definición variable buffer)
+p buffer    # imprimir variable buffer
$1 = "Hola\n"
+n           # ejecutar siguiente instrucción (puts())
***************** Práctica : LLAMADAS AL SISTEMA*******************
+n           # ejecutar siguiente instrucción (puts())
***************** Imprimo el mensaje de bienvenida mediante la función write():
+n           # ejecutar siguiente instrucción (write())
Hola\n
+n           # ejecutar siguiente instrucción (puts())
****************** Imprimo el mensaje de bienvenida mediante la llamada al sistema
syscall():
+n           # ejecutar siguiente instrucción (syscall())
Hola\n
+n           # ejecutar siguiente instrucción (exit (OxAA);)
[Inferior 1 (process 19617) exited with code 0252]
+quit        # salir de depuración
```

A continuación, se muestra depuración de programa equivalente a syscall_write_puts.c en lenguaje ensamblador.

```
+file syscall_write_puts  # cargar a la depuración modulo syscall_write_puts
Reading symbols from syscall_write_puts...
+b _start                 # punto de interrupción etiqueta _start
Punto de interrupción 1 at 0x1020: file syscall_write_puts.s, line 35.
+run                      # ejecutar depuración
Starting program: /home/sayechu/Escritorio/MEMORIA P4/syscall_write_puts
[Depuración de hilo usando libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".

Breakpoint 1, _start () at syscall_write_puts.s:35
+p /s (char *) &mensaje1  # imprimir contenido mensaje1 formato string
$1 = 0x56557000 "Imprimo el mensaje de bienvenida mediante la funcion puts()"
+n                        # ejecutar siguiente instrucción (push $mensajel)
+n                        # ejecutar siguiente instrucción (call puts)
0xf7def830 in puts () from /lib32/libc.so.6
+up                       # Volver programa principal
#1  0x5655602a in _start () at syscall_write_puts.s:36
```

```
+p /s (char *) &mensaje3  # imprimir en formato string contenido mensaje3
$2 = 0x5655707a "Hola\n"
+n                        # ejecutar siguiente instrucción (push $mensaje3)
+n                        # ejecutar siguiente instrucción (call puts)
0xf7def830 in puts () from /lib32/libc.so.6
+up                       # Volver programa principal
#1  0x56556034 in _start () at syscall_write_puts.s:41
+p /s (char *) &mensaje2
$3 = 0x5655703c "Imprimo el mensaje de bienvenida mediante la función write()"
+n                  # ejecutar siguiente instrucción (push $mensaje2)
+n                  # ejecutar siguiente instrucción (call puts)
0xf7def830 in puts () from /lib32/libc.so.6
+up                 # Volver a programa principal
#1  0x5655603e in _start () at syscall_write_puts.s:45
+n                  # ejecutar siguiente instrucción (push $LON_BUF)
+x /1xw $esp        # volcar contenido (1W) a partir de Stack Pointer
0xffffd1a0:  0x00000005
+n                  # ejecutar siguiente instrucción (push $mensaje3)
+x /2xw $esp        # volcar contenido (2W) a partir de Stack Pointer
0xffffd19c:  0x5655707a   0x00000005
+p /a &mensaje3     # imprimir dirección de memoria variable mensaje3
$4 = 0x5655707a
+n                  # ejecutar siguiente instrucción (push $WRTTE)
+x /3xw $esp        # volcar contenido (3w) a partir del Stack Pointer
0xffffd198:  0x00000004   0x5655707a   0x00000005
+n                  # ejecutar siguiente instrucción (call write)
+n                  # ejecutar siguiente instrucción (mov $WRITE, %eax)
+p /x $eax          # imprimir en formato hexadecimal contenido registro EAX
$5 = 0x4
+n                  # ejecutar siguiente instrucción (mov $STDOUT, %ebx)
+p /x $ebx          # imprimir formato hexadecimal contenido registro EBX
$6 = 0x1
+n                  # ejecutar siguiente instrucción (mov $mensaje3, %ecx)
+p /x $ecx          # imprimir en formato hexadecimal contenido registro ECX
$7 = 0x5655707a
+p /a &mensaje3     # imprimir dirección de memoria de mensaje3
$8 = 0x5655707a
+n                  # ejecutar siguiente instrucción (mov $LON_BUF, %edx)
+p /x $edx          # imprimir en formato hex. El contenido registro EDX
$9 = 0x5
+n                  # ejecutar siguiente instrucción (int $0x80)
+n                  # ejecutar siguiente instrucción (mov $SYS_EXIT, %eax)
+p /x $eax          # imprimir en formato hexadecimal contenido registro EAX
$10 = 0x1
+n                  # ejecutar siguiente instrucción (mov $SUCCESS, %ebx)
+p /x $ebx          # imprimir en formato hexadecimal contenido registro EBX
$11 = 0x0
+n                  # ejecutar siguiente instrucción (int $0x80)
[Inferior 1 (process 21895) exited normally]
+quit               # salir depuración de programa
```

Finalmente, se muestra la depuración del programa `sumMtoN.s`, cuyo programa está disponible en la sección respectiva de esta memoria.

```
+file sumMtoN       # cargar a la depuración modulo sumMtoN
Reading symbols from sumMtoN...
aviso: El archivo fuente es más reciente que el ejecutable.
+layout regs        # cargar Ventana de registros
+b _start           # punto de interrupción en etiqueta _start
Punto de interrupción 1 at 0x1000: file sumMtoN.s, line 25.
+focus cmd          # focus en Ventana de comandos
Focus set to cmd window.
+run                # ejecutar depuración programa
Starting program: /home/sayechu/Escritorio/sumMtoN

Breakpoint 1, _start () at sumMtoN.s:25
```

```
+n                  # ejecutar siguiente instrucción (pushl $10) -> 2° arg.
+x /1xw $esp        # volcar 1 word a partir de Stack Pointer
0xffffd2ac:  0x0000000a
+n                  # ejecutar siguiente instrucción (pushl $5)
+x /2xw $esp        # volcar 2 word a partir de Stack Pointer
0xffffd2a8:  0x00000005   0x0000000a
+step               # STEP a subrutina sumMtoN
+p $ebp             # Imprimir dirección a la que apunta EBP
$1 = (void *) 0x0
+n                  # ejecutar siguiente instrucción (pushl %ebp)
+x /3xw $esp        # volcar 3 word a partir de Stack Pointer
0xffffd2a0:  0x00000000   0x56556009   0x00000005
+p /a $esp          # imprimir dirección de memoria a la que apunta ESP
$2 = 0xffffd2a0
+p /a $ebp          # imprimir dirección de memoria a la que apunta EBP
$3 = 0x0
+n                  # ejecutar siguiente instrucción (movl %esp, %ebp)
sumMtoN () at sumMtoN.s:58
+p /a $ebp          # imprimir dirección de memoria a la que apunta EBP
$4 = 0xffffd2a0
+p /a $esp          # imprimir dirección de memoria a la que apunta ESP
$5 = 0xffffd2a0
+n                  # ejecutar siguiente isntrucción (subl $4, %esp)
+p /a $esp          # imprimir dirección de memoria a la que apunta ESP
$6 = 0xffffd29c
+x /1xw ($ebp+8)    # volcar contenido de dir. Mem. A la que apunta EBP + 8
0xffffd2a8:  0x00000005
+n                  # ejecutar siguiente instrucción (movl 8 (%ebp), %ebx)
+p $ebx             # imprimir contenido registro EBX
$7 = 5
+x /1xw ($ebp+12)   # volcar contenido de dir. Mem. A la que apunta EBP + 12
0xffffd2ac:  0x0000000a
+n                  # ejecutar siguiente instrucción (movl 12 (%ebp), %ecx)
+p $ecx             # imprimir contenido registro ECX
$8 = 10
+n                  # ejecutar siguiente instrucción (movl $0, %edx)
+n                  # ejecutar siguiente instrucción (mov %ecx, %eax)
+n                  # ejecutar siguiente instrucción (sub %ebx, %eax)
bucle () at sumMtoN.s:76
+p $eax             # imprimir contenido registro EAX
$9 = 5
+p $ebx             # imprimir contenido registro EBX
$10 = 5
+p $edx             # imprimir contenido registro EDX
$11 = 0
+n                  # ejecutar siguiente instrucción (add %ebx, %edx)
+p $edx             # imprimir contenido registro EDX
$12 = 5
+n                  # ejecutar siguiente instrucción (inc %ebx) -> M -> M+1
+p $ebx             # imprimir contenido registro EBX
$13 = 6
+p $eax             # imprimir contenido registro EAX
$14 = 5
+n                  # ejecutar siguiente instrucción (sub $1, %eax)
+p $eax             # imprimir contenido registro EAX
$15 = 4
+n                  # ejecutar siguiente instrucción (jns bucle)
+n                  # ejecutar siguiente instrucción (add %ebx, %edx)
+p $edx             # imprimir contenido registro EDX
$16 = 11
+p $ebx             # imprimir contenido registro EBX
$17 = 6
+n                  # ejecutar siguiente instrucción (inc %ebx) -> M -> M+1
+p $ebx             # imprimir contenido registro EBX
$18 = 7
+p $eax             # imprimir contenido registro EAX
$19 = 4
+n                  # ejecutar siguiente instrucción (sub $1, %eax)
```

```
+p $eax              # imprimir contenido registro EAX
$20 = 3
+n                   # ejecutar siguiente instrucción (jns bucle)
+p $ebx              # imprimir contenido registro EBX
$21 = 7
+p $edx              # imprimir contenido registro EDX -> resultado sumas
$22 = 11
+n                   # ejecutar siguiente instrucción (add %ebx, %edx)
+p $edx              # imprimir contenido registro EDX
$23 = 18
+n                   # ejecutar siguiente instrucción (inc %ebx) -> M -> M+1
+p $ebx              # imprimir contenido registro EBX
$24 = 8
+p $eax              # imprimir contenido registro EAX -> iteraciones bucle
$25 = 3
+n                   # ejecutar siguiente instrucción (sub $1, %eax)
+p $eax              # imprimir contenido registro EAX
$26 = 2
+n                   # ejecutar siguiente instrucción (jns bucle)
+p $ebx              # imprimir contenido registro EBX
$27 = 8
+p $edx              # imprimir contenido registro EDX -> resultado sumas
$28 = 18
+n                   # ejecutar siguiente instrucción (add %ebx, %edx)
+p $edx              # imprimir contenido registro EDX
$29 = 26
+n                   # ejecutar siguiente instrucción (inc %ebx) -> M -> M+1
+p $ebx
$30 = 9
+p $eax              # imprimir contenido registro EAX
$31 = 2
+n                   # ejecutar siguiente instrucción (sub $1, %eax)
+p $eax              # imprimir contenido registro EAX
$32 = 1
+n                   # ejecutar siguiente instrucción (jns bucle)
+p $ebx              # imprimir contenido registro EBX
$33 = 9
+p $edx              # imprimir contenido registro EDX -> resultado sumas
$34 = 26
+n                   # ejecutar siguiente instrucción (add %ebx, %edx)
+p $edx              # imprimir contenido registro EDX -> queda una iteración
$35 = 35
+p $ebx              # imprimir contenido registro EBX -> queda una iteración
$36 = 9
+n                   # ejecutar siguiente instrucción (inc %ebx)
+p $ebx              # imprimir contenido registro EBX -> hasta 10
$37 = 10
+p $eax              # imprimir contenido registro EAX antes de instr sub
$38 = 1
+n                   # ejecutar siguiente instrucción (sub $1, %eax)
+p $eax              # imprimir contenido registro EAX -> r. contador iteraciones
$39 = 0
+n                   # ejecutar siguiente instrucción (jns bucle)
+p $ebx              # imprimir contenido registro EBX -> antes de intr. add
$40 = 10
+p $edx              # imprimir contenido registro EDX -> antes de intr. add
$41 = 35
+n                   # ejecutar siguiente instrucción (add %ebx, %edx)
+n                   # ejecutar siguiente instrucción (inc %ebx)
+p $ebx              # imprimir contenido de registro EBX
$42 = 11
+n                   # ejecutar siguiente instrucción (sub $1, %eax) -> SF = 1
+p $eax              # imprimir contenido registro EAX
$43 = -1
+n                   # ejecutar siguiente isntrucción (jns bucle) -> SF=1,NOSALTA
+n                   # ejecutar siguiente instrucción (movl %edx,%eax) -> ret
+p $eax              # registro EAX -> devuelve resultado suma M a N
$44 = 45
```

```
+n                  # ejecutar siguiente instrucción (movl %ebp,%esp)
bucle () at sumMtoN.s:86
+p /a $ebp          # imprimir direccion de memoria a la que apunta EBP
$45 = 0xffffd2a0
+x /1xw $esp        # volcar dirección de memoria a la que ESP apuntaba incialm
0xffffd2a0:  0x00000000
+n                  # ejecutar siguiente instrucción (popl %ebp)
bucle () at sumMtoN.s:89
+p /a $ebp          # imprimir nueva dirección de EBP -> original
$46 = 0x0
+n                  # ejecutar siguiente instrucción (ret)
_start () at sumMtoN.s:32
+p $eax             # imprimir valor devuelto por subrutina
$47 = 45
+n                  # ejecutar siguiente instrucción (mov %eax, %ebx)
+p $ebx             # imprimir valor de registro EBX tras ultimo mov
$48 = 45
+n                  # ejecutar siguiente instrucción (movl $SYS_EXIT, %eax)
+p $eax             # imprimir contenido registro EAX tras ultima instr. mov
$49 = 1
+n                  # ejecutar siguiente instrucción (int $0x80) -> llamada s.o
[Inferior 1 (process 3610) exited with code 055]
+quit               # salir depuración
```

Conclusiones

Tras comenzar viendo las diferentes maneras de realizar una llamada al sistema operativo, tanto en el lenguaje C como en lenguaje ensamblador AT&T x86-32, se ha construido un programa equivalente a `syscall_write_puts.c` en lenguaje ensamblador. En este se ha visto diferentes maneras de imprimir un mensaje dado llamando al sistema con diferentes funciones.

En lo que se refiere a Subrutinas, se ha visto como se apilan las variables en la Stack (pila) y los diferentes punteros necesarios para acceder a dichas variables apiladas. Se han visto la estructura de la pila, instrucciones de la pila (push y pop), así como los argumentos necesarios para estas instrucciones.

Finalmente, gracias a la depuración se ha visto como realmente se almacenan las variables en la Stack, cómo se realizan las llamadas a subrutinas y cómo se devuelven los resultados de esta subrutina. En el caso del programa `sumMtoN`, se devolvía el resultado de la suma de la serie.

Gracias a esta práctica se puede lograr entender cómo funcionan realmente las funciones de programación de alto nivel a un nivel bajo. Es decir, el funcionamiento que hay detrás de una llamada a `printf()` en el lenguaje de programación de C.