PRÁCTICA 5: LLAMADAS A UNA SUBRUTINA:

ÍNDICE

A continuación vamos a realizar una serie de comandos en el gdb para ver cuáles son los contenidos de la pila:

- Sin ejecutar ninguna instrucción del programa
  - Estado de la pila
    - Top del stack: `x $esp` ó `x $sp` : stack pointer
    - Bottom del frame: `x $ebp` ó `x $fp` : frame pointer
    - Contenido del top de la pila (dirección sp): argc: número de argumentos string de la línea de comandos en ejecución
      - `x /xw $sp`
    - Contenido una posición anterior al top de la pila (dirección sp+4): argv[0]: dirección del 1º string de la línea de comandos en ejecución
      - `p /s *(char **)($sp+4)`

```
(gdb) x $esp
+x $esp
0xffffd150:     0x00000001
(gdb) x $ebp
+x $ebp
0x0:    No se puede acceder a la memoria en la dirección 0x0
(gdb) x /xw $esp
+x /xw $esp
0xffffd150:     0x00000001
```

```
(gdb) p /s *(char**) ($esp+4)
+p /s *(char**) ($esp+4)
$1 = 0xffffd2ff "/home/eduardo/Escritorio/Eduardo Ezponda/ESTRUCTURA DE COMPUTADORES/P5/sumMtoN"
(gdb)
```

La única diferencia entre el primer y el tercer comando es que se pide que se imprima en formato hexadecimal y en tamaño word (2 bytes).

Además, al no realizar ninguna instrucción del programa ni ningún push, se estará accediendo a una pila de la computadora distinta a la del programa.

- Ejecutar las líneas necesarias hasta entrar en la subrutina:
  - Comando step: `s` ya que el comando `n` no entra en la subrutina sino que la ejecuta completamente.
  - ¿A dónde apunta el stack pointer sp? ¿Qué información contiene a donde apunta el sp?
    - `x /i *(int *)$sp` : ¿qué instrucción es?

```
(gdb) x /i *(int*)$sp
+x /i *(int*)$sp
   0x56556009 <_start+9>:       mov    %eax,%ebx
(gdb)
```

Como se puede observar, realizamos un examine en formato instrucción del stack pointer sp. La instrucción resultante es mov %eax, %ebx.

A través del comando step (s), se accede directamente a la subrutina.

- Ejecutar el prólogo de la subrutina
  - Nuevo frame
    - Nuevo valor del frame pointer: `p $fp`
    - Valor del stack pointer: `p $sp`
    - Acceso a la dirección de retorno tomando como refefencia el nuevo frame pointer: `x /i *(int *)($fp+4)`

```
(gdb) p $ebp
+p $ebp
$2 = (void *) 0xffffd140
(gdb) p $esp
+p $esp
$3 = (void *) 0xffffd140
(gdb) x /i *(int*)($ebp+4)
+x /i *(int*)($ebp+4)
   0x56556009 <_start+9>:       mov    %eax,%ebx
(gdb)
```

Tras ejecutar el prólogo de la subrutina, en el que apilamos la antigua dirección del frame pointer para tenerla guardada al final de la subrutina y así por mantener la pila de la misma forma que al principio. Además hacemos que apunte el frame pointer al stack pointer para "crear" una nueva pila encima de la anterior.
Realmente no se crea una nueva pila porque únicamente existe una.

En las imágenes se puede observar el resultado de los dos prints y el examine.

- Ejecutar la subrutina hasta obtener el valor de retorno
  - Imprimir el valor de retorno: `p $eax`

```
+p $eax
$4 = 45
(gdb)
```

Tras realizar toda la subrutina sumMtoN, se devuelve el resultado de la función a través del registro eax. Con lo cual, en este caso, siendo la M 5 y la N 10, el resultado de la operación será 5 + 6 + 7 + 8 + 9 + 10 = 45.

- Ejecutar el epílogo de la subrutina
  - Valor del frame pointer: `p $fp`
  - Valor del stack pointer: `p $sp`
  - Dirección de retorno: `x *(int *)$sp`

```
(gdb) p $ebp
+p $ebp
$5 = (void *) 0x0
(gdb) p $esp
+p $esp
$6 = (void *) 0xffffd144
(gdb) x * (int*)$esp
+x * (int*)$esp
   0x56556009 <_start+9>:       mov    %eax,%ebx
```

Tras ejecutar el epílogo de la subrutina, dejaremos el stack de la misma manera que antes de la ejecución de la llamada a la subrutina.

Como se puede observar en la imagen, los resultados de los comandos se encuentran justo debajo.

- Ejecutar la instrucción de retorno
  - Dirección del stack pointer: `p $sp`
    - ¿Por qué ha cambiado la dirección del stack pointer?

La dirección del stack pointer varía porque vuelve a la rutina inicial, y por lo tanto apuntará a donde estaba inicialmente.

PROGRAMA: 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(1o arg=M, 2o 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
	## DATOS
	.section .data

	## INSTRUCCIONES
	.section .text
	.global _start
_start:
	## Paso los dos argumentos M y N a la subrutina a través de la pila
	pushl $10	#push second argument -> N
	pushl $5	#push first argument -> M

	## Llamada a la subrutina sum1toN
	call sumMtoN

	## Paso la salida de sum11toN al argumento a la llamada al sistema exit()
	mov %eax, %ebx # ( %ebx is returned)
	## Código de la llamada al sistema operativo
	movl $SYS_EXIT, %eax # llamada exit
	## Interrumpo al S.O.
	int $0x80

##Subrutina: sumMtoN
##Descripción: calcula la suma de números enteros en secuencia desde el 1o sumando hasta el 2o sumando
##	Argumentos de entrada: 1o sumando y 2o sumando los argumentos los pasa la rutina principal a través de la pila:
##	1o se apila el último argumento y finalmente se apila el 1o 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
	## Captura de argumentos
	movl 8( %ebp), %ebx #1o argumento copiado en %ebx
	movl 12( %ebp), %ecx #2o argumento copiado en %ecx

	## suma la secuencia entre el valor del 1oarg y el valor del 2oarg
	## 1o arg < 2oarg
	## utilizo como variable local EDX en lugar de la reserva externa para variable local: optimiza velocidad
	## Inicializo la variable local suma
	movl $0, %edx

	## Número de iteracciones
	mov %ecx, %eax
	sub %ebx, %eax

bucle:
	add %ebx, %edx
	inc %ebx
	sub $1, %eax
	jns bucle

	## Salvo el resultado de la suma como el valor de retorno
	movl %edx, %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
	.end
```

FUNCIONAMIENTO PROGRAMA:

En primer lugar, se declara la macro SYS_EXIT como 1 para luego llamar posteriormente a exit con el registro eax.

Para llamar a la subrutina sumMtoN, previamente se necesita pasar los valores de M y N. Para ello, se utiliza una pila y por ello se apilan los valores 5 y 10. La pila es una estructura LIFO, con lo que se apilarán los valores continuamente por encima del anterior y se desapilarán los valores introducidos por último lugar. Además, la función se declara como .type sumMtoN, @function.

Se llama a sumMtoN a través de call. A continuación comenzaría el prólogo debido a que al final de la subrutina se tiene que devolver la pila en el mismo estado en el que se encontraba previamente. El contenido de la pila tiene que ser el mismo, y el stack pointer (esp) y frame pointer (ebp) tienen que apuntar a la misma zona de memoria previa.

Por lo tanto, apilamos la dirección del antiguo frame pointer para tenerla guardada para el final y apuntamos el “nuevo” frame pointer a donde apuntaba el stack pointer para “crear” una nueva pila en la zona superior. Este proceso se denomina prólogo.

A partir de ello, capturamos los elementos con una dirección relativa a base. Observando el estado de la pila en la siguiente página, sabiendo que cada tamaño es de 4 bytes, únicamente sumando esos 4 bytes de donde apunta el esp accederemos al segundo y tercer argumento de la pila. Una vez capturados, nos dispondremos a través de la etiqueta bucle a realizar la suma en cada una de las iteraciones hasta que eax sea 0.

Cuando eax sea 0, significa que el bucle ha concluido y por tanto el resultado de la operación en este caso se guarda en el registro edx.

Como es una función, es necesario pasar el valor de retorno al registro eax y realizar el epílogo para dejar la pila de la misma manera previa.

Volvemos a la rutina principal a través de ret. Ya únicamente quedaría la parte final del programa en el que se devuelve al sistema operativo el resultado de la operación (45), y se llama despueś al sistema operativo para que tome el control e interrumpa el programa.

ESTADOS DE LA PILA/STACK :

esp ->10 y ebp ->10

10

```
## Paso los dos argumentos M y N a la subrutina a través de la pila
pushl $10»  #push second argument -> N
```

Esp -> 5 y ebp -> 10

5
10

```
## Paso los dos argumentos M y N a la subrutina a través de la pila
pushl $10»  #push second argument -> N
pushl $5»   #push first argument -> M
```

EPÍLOGO:

esp -> ebp y ebp -> ebp

ebp
5
10

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

esp -> variable local en memoria externa y ebp -> ebp

ebp
5
10

```
»   ## Variable local en memoria externa: suma
»   subl $4, %esp
```

PRÓLOGO:

esp -> ebp y ebp -> ebp

ebp
5
10

esp -> 5 y ebp -> 10

5
10

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

20.5. Práctica 5: LLamadas a una Subrutina

20.5.1. Módulo sumMtoN_aviso.c

- Desarrollar el programa *sumMtoN_aviso.c* equivalente al módulo en lenguaje asm *sumMtoN.s* y añadiendo un mensaje de aviso en caso de error indicando la relación correcta entre los parámetros 1º sumando y 2º sumando.

20.5.2. Módulo sumMtoN_aviso.s

- Añadir al programa fuente *sumMtoN.s* un mensaje de aviso en caso de error indicando la relación correcta entre los parámetros 1º sumando y 2º sumando.

SUMMTON_AVISO.C:

```c
/*Programa: sumMtoN_aviso.c
## Descripción: realiza la suma de números enteros de la serie M,M+1,M+2,M+3,...N función : sumMtoN(1o arg=M, 2o arg=N) donde M < N
## Ejecución: Editar los valores M y N y compilar el programa.
##      Ejecutar $./sumMtoN_aviso
##      El resultado de la suma se captura del sistema operativo con el comando linux: echo $?

##      gcc -m32 -g -o sumMtoN_aviso sumMtoN_aviso.c
##      Ensamblaje as --32 --gstabs sumMtoN_aviso.c -o sumMtoN_aviso.o
##      linker -> ld -melf_i386 -o sumMtoN_aviso sumMtoN_aviso.c
*/

#include <stdio.h>
#include <stdlib.h>

int sumMtoN (int m, int n);

int main (void)
{
        int m, n, ebx;
        char* mensaje = "ERROR, se debe de cumplir la relación M < N\n";

        m = 5;
        n = 10;

        if (m < n)
        {
                ebx = sumMtoN (m, n);
        }
        else
        {
                printf("%s",mensaje);
                ebx = 0; //realmente no haría falta esto ya que si entra a la función sumMtoN no daría ninguna vuelta en el bucle y el resultado sería 0
        }
        exit(ebx);
}
int sumMtoN (int m, int n)
{
        int suma = 0;

        while (n - m >= 0)
        {
                suma = suma + m;
                m = m + 1;
        }

        return suma;
}
```

El funcionamiento del programa sumMtoN_aviso.c es similar al sumMtoN solo que se le añade un comentario en el caso de que no se cumpla la condición y devolviendo con el exit el valor 0.

En el caso de que se cumpla la condición, no se imprimirá dicho mensaje por pantalla y se devolverá al sistema operativo el valor de la suma.

SUMMTON_AVISO.S:

```
## Programa: sumMtoN_aviso.s
## Descripción: realiza la suma de números enteros de la serie M,M+1,M+2,M+3,...N función : sumMtoN(1o arg=M, 2o 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
»   ## DATOS
»   .section .data
n:
»   .int 10
m:
»   .int 5
mensaje:
»   .asciz "ERROR, se debe de cumplir la relación M < N"
»   ## INSTRUCCIONES
»   .section .text
»   .global _start
_start:
»   mov n, %eax
»   cmp m, %eax #compara realizando un sub -> eax - m

»   jl string

»   ## Paso los dos argumentos M y N a la subrutina a través de la pila
»   pushl n»#push second argument -> N ->no apilar dirección sino contenido
»   pushl m»#push first argument -> M ->no apilar dirección sino contenido

»   ## Llamada a la subrutina sum1toN
»   call sumMtoN
»   jmp fin
»   ## Paso la salida de sum11toN al argumento a la llamada al sistema exit()
string:
»   push $mensaje
»   call puts
»   mov $0, %eax  #paso el valor de 0 al registro eax porque luego le va a pasar ese valor a ebx
fin:
»   mov %eax, %ebx # ( %ebx is returned)
»   ## Código de la llamada al sistema operativo
»   movl $SYS_EXIT, %eax # llamada exit
»   ## Interrumpo al S.O.
»   int $0x80

##Subrutina: sumMtoN
##Descripción: calcula la suma de números enteros en secuencia desde el 1o sumando hasta el 2o sumando
##» Argumentos de entrada: 1o sumando y 2o sumando los argumentos los pasa la rutina principal a través de la pila:
##» 1o se apila el último argumento y finalmente se apila el 1o 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
»   ## Captura de argumentos
»   movl 8( %ebp), %ebx #1o argumento copiado en %ebx
»   movl 12( %ebp), %ecx #2o argumento copiado en %ecx

»   ## suma la secuencia entre el valor del 1oarg y el valor del 2oarg
»   ## 1o arg < 2oarg
»   ## utilizo como variable local EDX en lugar de la reserva externa para variable local: optimiza velocidad
»   ## Inicializo la variable local suma
»   movl $0, %edx

»   ## Número de iteracciones
»   mov %ecx, %eax
»   sub %ebx, %eax

bucle:
»   add %ebx, %edx
»   inc %ebx
»   sub $1, %eax
»   jns bucle

»   ## Salvo el resultado de la suma como el valor de retorno
»   movl %edx, %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
»   .end
```

En el siguiente programa se realizan una serie de variaciones del programa sumMtoN.s para conseguir el objetivo de mostrar por pantalla el contenido de una string avisando al usuario de que ha introducido los datos incorrectamente al no cumplir la relación M < N.
Además, se le pasa al sistema operativo el valor de 0 en ese caso que implica que ha habido un error.

Para incluir estos objetivos, es necesario pasar a un registro, en este caso eax, el valor de N o M y compararlo con el otro a través de la instrucción cmp. En el caso que M sea estrictamente menor que N, el programa continuará con su correcto funcionamiento, similar al de sumMtoN.s

En el caso de que no se cumpla la condición, se mostrará por pantalla la cadena de caracteres mensaje, para que el usuario conozca el error en la introducción de los datos.

COMPILACIÓN:

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

gcc -m32 -nostartfiles -g -o  sumMtoN_aviso sumMtoN_aviso.s

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

Cabe destacar el uso de -m32 para usar una máquina de 32 bits, y -g para cargar la tabla de símbolos. Además se ha añadido el -nostartfiles porque en este caso se utiliza la etiqueta _start.

CONCLUSIÓN:

Durante esta práctica se han realizado llamadas a subrutinas, en este caso sumMtoN. Para ello es necesario realizar un call y una declaración previa de la función pasando al final de la función el valor de retorno con el registro eax. Además, la única forma de pasar los argumentos necesarios a la subrutina es a través de la stack. Se apilan los argumentos con la instrucción push, y se desapilan con la instrucción pop (se podrá añadir un sufijo a cada uno de ellos para determinar el tamaño).

Por último, como es necesario dejar la pila de la misma manera que se ha encontrado previamente a la llamada de la subrutina, se realizará un prólogo y un epílogo para completar dicho fin.

En el prólogo se apilará la dirección antigua del frame pointer para tenerla guardada, y se creará una nueva pila en la parte superior de la anterior pila.

En el epílogo se dejará la pila como se ha encontrado previamente realizando las instrucciones oportunas dependiendo del uso que se haya llevado a cabo de la pila en la subrutina.

Por último, se crean dos programas tanto en asm como en lenguaje c en los que se imprime un mensaje de error en el caso de que no se cumpla la condición M < N.