Práctica 2.- Programación ensamblador x86-64 Linux

1 Resumen de objetivos

Al finalizar esta práctica, se debería ser capaz de:

- Usar las herramientas `gcc`, `as`, `ld`, `objdump` y `nm` para compilar código C, ensamblar y enlazar código ensamblador, y localizar y examinar el código generado por el compilador.
- Reconocer la estructura del código generado por `gcc` para rutinas aritméticas muy sencillas, relacionando las instrucciones del procesador con la construcción C de la que provienen.
- Describir la estructura general de un programa ensamblador en `gas` (GNU assembler).
- Escribir un programa ensamblador sencillo.
- Hacer llamadas al sistema operativo (*kernel* Linux) desde ensamblador x86-64.
- Enumerar los registros e instrucciones más usuales de los procesadores de la línea x86-64.
- Usar con efectividad un depurador como `gdb`/`ddd` para ver los registros, ejecutar paso a paso y con puntos de ruptura, desensamblar el código, y volcar el contenido de la pila y los datos.
- Reconocer la utilidad de entornos integrados de desarrollo como Eclipse para editar código fuente, compilar programas C, ensamblar programas ASM y depurar los respectivos ejecutables.
- Argumentar la utilidad de los depuradores para ahorrar tiempo de depuración, y reconocer cómo estas herramientas permiten familiarizarse con la arquitectura del computador.
- Explicar la gestión de pila en procesadores x86-64.
- Recordar y practicar en una plataforma mixta de 32-64 bits la representación de distintos tipos de datos (caracteres, números naturales, enteros en complemento a dos), y el funcionamiento de diversas operaciones (incluyendo suma entera en doble precisión y división entera).

2 Herramientas de prácticas

Las prácticas se realizarán en Linux utilizando las herramientas GNU. Opcionalmente podrán usarse ddd y/o el entorno Eclipse, aunque la configuración avanzada del mismo no es objetivo de esta asignatura, y los guiones asumirán que se usa algún editor (`gedit`, `vi`, …) y el interfaz usuario texto de gdb. Usaremos `gcc` para compilar (traducir fuente C a ensamblador, objeto o ejecutable), `as` para ensamblar (traducir fuente ensamblador a objeto), `ld` para enlazar (combinar varios ficheros objeto en un único fichero ejecutable), y para depurar usaremos `gdb` (con `-tui` o a través del *front-end* ddd o del entorno Eclipse).

En la Figura 1 se pueden ver las herramientas que se deben ejecutar para obtener un fichero ejecutable a partir de un fichero fuente en lenguaje C: compilador, ensamblador y enlazador. En la práctica, el programador en lenguaje de alto nivel no tiene que ejecutar los tres programas por separado.

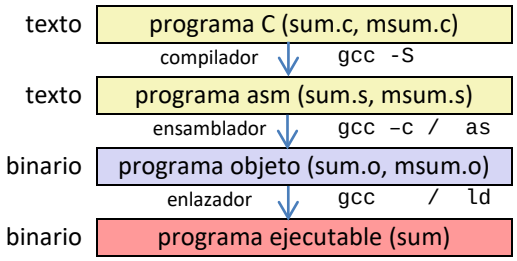

Figura 1: proceso de compilación

El código fuente se puede crear con cualquier editor de texto, como por ejemplo `gedit`, procurando que la extensión coincida con el tipo de lenguaje usado (.c/.s). El procesamiento del compilador `gcc` puede detenerse tras las etapas de traducción a ensamblador o a objeto con los modificadores (*switches*) `-S`/`-c`. El proceso puede continuarse con el propio `gcc` o con el ensamblador `as` y el enlazador `ld`.

Por ejemplo, a partir de estos dos ficheros fuente en lenguaje C…

```
long plus(long, long);

void sumstore(long x, long y,
              long *dest) {
  long t = plus(x, y);
  *dest = t;
}
```

```
void sumstore(long x, long y,
              long *dest);
int main() {
    long d;
    sumstore(2, 3, &d);
    return d;
}
```

Figura 2: fichero fuente sum.c

```
long plus(long a, long b) {
    long s = a + b;
    return s;
}
```

Figura 3: fichero fuente msum.c

…usando gcc se podría producir el ejecutable `sum` con una sola orden, o generar el código ensamblador u objeto para cada fichero, o retomar esos ficheros ensamblador u objeto para producir el ejecutable. Teniendo los ficheros ensamblador, también se podría continuar con las otras herramientas `as`/`ld`.

Los modificadores necesarios se pueden consultar en los manuales (`man gcc`, `man as`, `man ld`…) aunque los más habituales son:

gcc			
switch	**argumento**	**significado**	**explicación**
-c		Compile	Compila o ensambla los fuentes, pero no enlaza. Se obtiene un objeto por cada fuente. Por defecto, los objetos se llaman como el fuente.o
-S		aSsembly	Compila los fuentes pero no ensambla. Se obtiene un fuente ensamblador por cada fuente C. Por defecto, se llaman como el fuente.s
-o	fichero.ext	Output	Cambiar el nombre del fichero producido. Por defecto, ejecutable a.out, objeto fuente.o, ensamblador fuente.s
-g	1…3	debuG	Genera información de depuración (por defecto, nivel 2; Eclipse usa nivel 3)
-O	0…3, s, g		Optimizar para velocidad (0 no optimización…3 agresivo) o tamaño (s, size). Poner –O = –O1. No poner nada = –O0. Para depurar usar –Og mejor.
-m32 **-m64**		Machine	Genera código para entorno de 32/64 bits, aunque no sea el configurado por defecto.
-L	dir	LibraryDir	Añadir dir a la lista de búsqueda de librerías (preferible sin espacio: -Ldir)
-l	name	LibraryName	Enlazar contra libname.so / libname.a (preferible usar sin espacio: -lname)
-print-file-name=lib			Imprime el pathname que se usaría para lib a la hora de enlazar
-v		Verbose	Imprime los comandos ejecutados para las diversas etapas de compilación
-###			Como –v, pero no ejecuta los comandos. Es más fácil de leer.
-fno-stack-protector **-fno-asynchronous-unwind-tables** **-fno-omit-frame-pointer -no-pie** **-fno-if-conversion -fno-tree-ch** **-fno-reorder-blocks -fno-tree-ter**			Versiones recientes de gcc añaden protecciones o info.debug y dificultan leer el código ASM generado. Estos switches quitan dos de esas opciones. Versiones recientes de gcc optimizan demasiado y generan código distinto al mostrado en transparencias en clase. Estos switches permiten quitar la optimización concreta y reproducir el código explicado en clase.
as			
-g		debuG	mismo sentido
-o	fichero.o	Output	mismo sentido
--32 **--64**			mismo sentido
ld			
-L	dir	LibraryDir	mismo sentido
-l	name	LibraryName	mismo sentido
-m	elf_i386 elf_x86_64	Machine	mismo sentido
-M		printImpMap	Imprimir mapa de enlazado, posición en memoria de objetos, símbolos, etc.

Tabla 1: modificadores para gcc, as, ld

Ejercicio 1: gcc

Crear los ficheros sum.c y msum.c mostrados anteriormente (Figura 2, Figura 3), y reproducir con ellos la siguiente sesión en línea de comandos Linux. Notar cómo los modificadores `–fno-stack-protector` y `-fno-asynchronous-unwind-tables` se anotan en variables de entorno para ahorrar tecleado.

```
gcc         sum.c msum.c -o sum  # compilar de una vez
./sum ; echo $?                  # muestra cód. ret. 5 = 2+3
file sum                         # ELF 64-bit LSB shared object

gcc -no-pie sum.c msum.c -o sum  # position-indep. añadido recientemente
./sum ; echo $?                  # quitar para generar ejecutable normal
file sum                         # ELF 64-bit LSB executable
```

```
FNSP=-fno-stack-protector           # ahorrarse teclear switches tan largos
FNAUT=-fno-asynchronous-unwind-tables

gcc -Og -S  sum.c $FNAUT          # crea  sum.s (-CallFrameInfo)
       cat  sum.s                 # rax res, rbx s-invcado, rdi/si/dx args
gcc -Og -S msum.c $FNAUT $FNSP    # crea msum.s (-StackProtector)
       cat msum.s                 # 8(%rsp) es var.local d

gcc -c      sum.s msum.c          # crea sum.o, msum.o desde ASM/C
ls; file *.o                      # ELF 64-bit LSB relocatable

gcc -no-pie sum.o msum.o -o sum   # crea exe sum(no a.out) desde objetos
file sum; ./sum; echo $?          # ELF 64-bit LSB executable - funciona
```

Figura 4: compilación, ensamblado y enlazado, usando gcc

Observar que el código ensamblador x86-64 menciona tanto registros de 32 bits (p.ej. en `main` se usan ESI, EDI, EAX en la secuencia `movl 3,%esi / movl $2, %edi / call sumstore / movl 8(%rsp), %eax`) como registros de 64 bits (p.ej. en `sumstore` se usan RBX, RDX, RAX en la secuencia `pushq %rbx / movq %rdx, %rbx / call plus / movq %rax, (%rbx)`). En los comandos del *shell bash*, $? representa el código de estado retornado por el último programa ejecutado, y la almohadilla # introduce un comentario hasta final de línea. El punto y coma "`;`" separa dos comandos en la misma línea.

Ejercicio 2: as y ld

El mismo resultado se puede obtener con las distintas herramientas separadamente (`as` y `ld`, limitando el uso de `gcc` a compilar C→ASM). Reproducir la siguiente sesión de comandos Linux. Notar que el último comando `ld` es tan largo que se ha fraccionado en dos líneas, usando el *escape* \↵ (*backslash*-<Enter>), que indica al *shell bash* que se debe ignorar el salto de línea y considerar la siguiente línea una continuación de la actual. En caso de duda, teclear todo el comando en una sola línea, en lugar de aplicar *escape* al salto de línea.

```
rm    sum *sum.[os]; ls               # limpiar ficheros producidos
gcc -Og -S sum.c msum.c $FNAUT $FNSP  # compilar: ya no necesitamos gcc
ls        *sum.?                      # seguir trabajando con [m]sum.s

as  sum.s -o  sum.o       # ensamblar creando objeto sum.o (no a.out)
as msum.s -o msum.o
ls    *.o; file *.o

ld       sum.o msum.o     # warn: falta _start
                          # _start está definido en crt1.o, verlo con nm
gcc -### sum.o msum.o     # una forma de ver cómo enlaza gcc
gcc -### -no-pie  *.o     # reproducir último paso collect2 usando ld:
ls /lib64                 # : /usr/lib/x86*/crt?.o -lc –dynamic-linker …
ls /usr/lib/x86_64-linux-gnu/*crt*  # otros progs pudieran necesitar crtbegin/end
                          # backslash \ es "escape" para <Enter> en bash
ld sum.o msum.o -o sum    -dynamic-linker /lib64/ld-linux-x86-64.so.2 \↵
                          /usr/lib/x86_64-linux-gnu/crt?.o -lc
file sum; ./sum; echo $?  #funciona
```

Figura 5: compilación usando gcc, ensamblado y enlazado usando as y ld

Dependiendo de la version y configuración del compilador en la distribución que se use, el proceso de enlazado indicado por `gcc -###` será más o menos complicado. En este caso (gcc 7.3.0 Ubuntu 18.04.1) bastó con enlazar contra *libC* (*switches* `-lc` y `–dynamic-linker`) y añadir tres ficheros de *runtime* presentes en `/usr/lib/x86_64-linux-gnu` (`crt1.o`, `crti.o` y `crtn.o`). Otros programas podrían necesitar los *runtime* `crtbegin.o` y `crtend.o` (presentes en donde indica `gcc –print-file-name=`). El *backslash* "\" continúa un comando que se desea prolongar a la siguiente línea. Si se desea, se puede teclear el comando `ld` en una sola línea sin pulsar <Enter>, haciendo innecesario el *backslash*.

Si se desea, se puede modificar el programa msum.c para sustituir la última sentencia `return` de `main` por un `printf("2 * 3 --> %ld\n", d);` (en cuyo caso también convendría empezar con `#include <stdio.h>` para declarar el prototipo de la función `printf`). De esta forma, el programa imprime el resultado, en lugar de retornarlo como código de estado. La sintaxis del formato de `printf` se puede consultar en los manuales (`man 3 printf`).

2.1 Código ensamblador y código máquina

La traducción del ejemplo a lenguaje ensamblador ya la vimos en la sesión de la Figura 4:

```
// gcc -Og -S sum.c
// … -fno-asynchronous-unwind-tables

long plus(long, long);

void sumstore(long x, long y,
              long *dest) {
  long t = plus(x, y);
  *dest = t;
}
```

Figura 6: fichero fuente C sum.c

```
        .file  "sum.c"
        .text
        .globl sumstore
        .type  sumstore, @function
sumstore:
        pushq  %rbx
        movq   %rdx, %rbx
        call   plus@PLT
        movq   %rax, (%rbx)
        popq   %rbx
        ret
        .size  sumstore, .-sumstore
        .ident "GCC: (Ubuntu 7.3.0-16u3) 7.3.0"
        .section .note.GNU-stack,"",@progbits
```

Figura 7: fichero ensamblador sum.s

La versión ensamblador es una representación legible de las instrucciones máquina en que se convierte el programa, como las comentadas anteriormente `pushq %rbx` / `movq %rdx, %rbx`. Contiene también directivas, como `.text` (iniciar sección de código) y `.size` (tamaño de un objeto). En este caso, se define el tamaño del objeto `sumstore` (función global, ver `.global` y `.type`) como `".-sumstore"`, esto es, la diferencia entre el contador de posiciones `"."` y la propia etiqueta `sumstore`.

El ensamblador *emite* código máquina conforme traduce ensamblador, ocupando posiciones (bytes) de memoria. El símbolo `"."` es la posición por donde se va ensamblando, y cada etiqueta toma el valor del contador cuando se emite.

El fichero objeto sum.o no es legible, ya que contiene código máquina, pero se puede desensamblar y consultar los símbolos que define con otras utilidades del paquete GNU *binutils*, como `objdump` y `nm`. Los modificadores necesarios se pueden consultar en los manuales (`man objdump`, `man nm`) aunque los más habituales son:

objdump	fich.obj.		
switch	**argumento**	**significado**	**explicación**
-d		Disassemble	Muestra los mnemotécnicos ensamblador correspondientes a las instrucciones máquina en las secciones de código del fichero objeto
-S		Source	Intercala código fuente con desensamblado. Implica –d. Requiere compilar con –g.
-h		Headers	Resumen de las cabeceras de sección presentes
-r		Reloc	Muestra las reubicaciones
-t		table	Muestra las entradas de la tabla de símbolos (similar a nm)
-T		Table	Muestra tabla de símbolos dinámicos (similar a nm -D). Para librerías compartidas.
-j / --section=	name	Just	Seleccionar información sólo de la sección mencionada
-s / --full-contents			Mostrar contenidos completos de todas las secciones (o sólo de las indicadas con -j)
nm	fich.obj.		
-D		Dynamic	Mostrar símbolos dinámicos (p.ej. en librerías compartidas)

Tabla 2: modificadores para objdump, nm

Ejercicio 3: objdump y nm

Reproducir la siguiente sesión en línea de comandos Linux

```
objdump -d   sum.o # mostrado al lado->
objdump -t   sum.o
nm           sum.o # sumstore en .text

objdump -S    sum.o      # falta -g
```

```
sum.o:     file format elf64-x86-64


Disassembly of section .text:
```

```
gcc -g -Og -c sum.c
objdump -S    sum.o      # ahora sí

objdump -t    sum.o      # secciones -g
objdump -h    sum.o      # ver.text=11B
gcc    -Og -c sum.c      # quitar -g
objdump -S    sum.o      # ahora no
objdump -h    sum.o
```

Figura 8: sesión Linux

```
0000000000000000 <sumstore>:
  0: 53          push   %rbx
  1: 48 89 d3    mov    %rdx,%rbx
  4: e8 00 00
        00 00    callq  9 <sumstore+0x9>
  9: 48 89 03    mov    %rax,(%rbx)
  c: 5b          pop    %rbx
  d: c3          retq
```

Figura 9: desensamblado de sum.o

Como vemos en la Figura 9, el ensamblador emitió 14 bytes, el contador iba por 0 cuando se definió `sumstore`, irá por 0xe tras emitir `ret`, y por tanto "`.size sumstore, .-sumstore`" calculará el tamaño de `sumstore` como 14. La primera instrucción, "`push %rbx`", se codifica en lenguaje máquina como 0x53 y ocupa 1B, la posición 0. La segunda instrucción, "`mov %rdx,%rbx`", empieza en 0x1, ocupa 3B y acaba por tanto en 0x3, dejando el contador de posiciones en 0x4.

Se puede comprobar (con `nm`) que el símbolo `_start` que nos impedía enlazar nuestros dos objetos con `ld` a secas (Figura 5) está en uno de los objetos del *runtime* de gcc. En el Apéndice 2 hay un resumen de las instrucciones y modos de direccionamiento x86-64 y de las directivas del ensamblador GNU, que puede resultar útil para entender tanto este desensamblado como el siguiente programa completo.

3 Primer programa completo en ASM: llamadas al sistema

Teclear (o copiar-pegar, o descargar del sitio web de la asignatura) el código de la Figura 10 en un fichero llamado `saludo.s`. Si se opta por reescribirlo, tener en cuenta que la almohadilla # indica que el resto de la línea es comentario, con lo cual no es necesario copiarlo.

En el código se pueden distinguir: instrucciones del procesador, directivas del ensamblador, etiquetas, expresiones y comentarios. Las instrucciones usadas en este caso han sido SYSCALL y MOV, para realizar las dos llamadas al sistema requeridas (*`WRITE`* escribir mensaje y *`EXIT`* terminar programa). Las directivas son comandos que entiende el ensamblador (no instrucciones del procesador), y se han usado para declarar las secciones de datos y código (`.data` y `.text`), para emitir un *string* y un entero (en `.data`) y para declarar como global el punto de entrada (en `.text`). Las etiquetas se han usado para nombrar esos tres elementos (`saludo`, `longsaludo` y `_start`), y poder referirse a ellos posteriormente (en *`WRITE`* o en `.global`), ya que representan su dirección de comienzo. Notar el uso del contador de posiciones y aritmética de etiquetas (`.-saludo`) para calcular la longitud del *string*. La otra expresión de inicialización es el valor del *string*. Los comentarios se indican con # ó `/**/`. Los valores inmediatos se prefijan con $, y los registros con %.

```
# saludo.s:  Imprimir por pantalla
#            Hola a todos!
#            Hello, World!
# retorna:   código retorno 0, programado en la penúltima línea
#            comprobar desde línea de comandos bash con echo $?

# SECCIÓN DE DATOS (.data, variables globales inicializadas)
#     datos hex, octal, binario, decimal, char, string:
#            0x, 0,     0b,      díg<>0,  ‘,    ""
#     ejs: 0x41, 0101, 0b01000001,  65,  ‘A,   "AAA"

.section .data      # directivas comienzan por .
            # no son instrucciones máquina, son indicaciones para as
            # etiquetas recuerdan valor contador posiciones (bytes)
saludo:
      .ascii "Hola a todos!\nHello, World!\n"       # \n salto de línea

longsaludo:
      .quad   .-saludo    # . = contador posic. Aritmética de etiquetas.

# SECCIÓN DE CÓDIGO (.text, instrucciones máquina)

.section .text      # cambiamos de sección, ahora emitimos código
.global _start      # muestra punto de entrada a ld (como main en C)
```

```
_start:                     # punto de entrada ASM (como main en C)
#       Llamada al sistema WRITE, consultar "man 2 write"
#       ssize_t write(int fd, const void *buf, size_t count);
        mov $1, %rax        # write: servicio 1 kernel Linux
        mov         $1,%rdi #    fd: descriptor de fichero para stdout
        mov    $saludo,%rsi #   buf: dirección del texto a escribir
        mov longsaludo,%rdx # count: número de bytes a escribir
        syscall             # llamar write(stdout, &saludo, longsaludo);

#       Llamada al sistema EXIT, consultar "man 2 exit"
#       void _exit(int status);
        mov $60, %rax       #   exit: servicio 60 kernel Linux
        mov  $0, %rdi       # status: código a retornar (0=OK)
        syscall             # llamar exit(0);
```

Figura 10: saludo.s: ejemplo de llamadas al sistema WRITE y EXIT

En arquitectura x86-64 cada posición de memoria es un byte. Ya vimos en la Figura 7 cómo se usó aritmética de etiquetas para calcular el tamaño ocupado por la función `sumstore`. En este caso, podríamos modificar (alargar o acortar) el *string* en el código fuente ensamblador, y la variable `longsaludo` tomaría automáticamente el valor correcto (longitud) para la posterior llamada a *`WRITE`*.

Dado que los Sistemas Operativos se ejecutan en un nivel de privilegio elevado (*espacio kernel* vs. *espacio usuario*) se debe utilizar algún mecanismo proporcionado por la arquitectura para elevar el privilegio de un proceso y/o permitirle ejecutar una llamada al sistema (*syscall*). Tradicionalmente se han usado las interrupciones software a tal efecto, especialmente en procesadores donde dicho mecanismo era el único disponible para conmutar entre niveles de privilegio, e incluso en procesadores donde no había espacio *kernel* protegido en oposición al espacio de usuario. El programador utiliza un vector concreto (`0x80` en el caso de Linux i386, ver `man 2 syscall`) cuando desea realizar la llamada. La subrutina de servicio espera encontrar el número de servicio y hasta 6 argumentos en registros del procesador (EAX y EBX, ECX, EDX, ESI, EDI, EBP en Linux i386) de manera que el programador debe fijar estos valores antes de realizar la interrupción `int 0x80`. Si la llamada al sistema produce un valor de retorno, lo devuelve en otro registro (EAX en Linux i386). Los números de servicio (llamada) pueden encontrarse en `/usr/include/x86_64-linux-gnu/asm/unistd_32.h`.

La demanda de mayores prestaciones ha llevado a los fabricantes a proporcionar interfaces más rápidos para conmutar a espacio *kernel*, y así Linux x86-64 usa la instrucción `syscall` (ver `man 2 syscall`), pasando el número de servicio en RAX y los argumentos en RDI, RSI, RDX, R10, R8, R9 (una variante de la *SystemV AMD64 ABI*). El programador debe fijar dichos valores antes de ejecutar la instrucción `syscall`. Si la llamada al sistema produce un valor de retorno, lo devuelve en RAX. Los números de servicio (llamada) pueden encontrarse en `/usr/include/x86_64-linux-gnu/asm/unistd_64.h`.

Los argumentos de cada llamada pueden conocerse leyendo la correspondiente página de manual de la sección 2 (Llamadas al Sistema). En nuestro caso, nos interesa consultar `man 2 write` y `man 2 exit` para saber que tienen 3 (RDI, RSI, RDX) y 1 argumentos, respectivamente. El argumento de `exit(status)` es un código de retorno (se puede probar a cambiarlo en el fuente y comprobarlo con `echo $?`) mientras que `write(fd, buf, count)` escribe `count` bytes a partir de `buffer` en el descriptor de fichero `fd`. En concreto, el descriptor para la salida estándar (STDOUT_FILENO) está definido en `/usr/include/unistd.h` (también se puede comprobar listando `ls -la /dev/stdout`).

Ejercicio 4: gdb -tui

Ensamblar y enlazar el programa `saludo.s`, incluyendo información de depuración, y reproducir la siguiente sesión gdb. Aunque permitiremos usar el *frontend* ddd o el entorno Eclipse en modo gráfico si estuvieran disponibles, es conveniente aprender también los comandos gdb en modo texto, y de hecho alguna funcionalidad sólo puede accederse en dicho modo. Hay una lista de comandos en el enlace [6].

```
as  -g   saludo.s -o saludo.o    # ensamblar incluyendo info. depuración
ld       saludo.o -o saludo      # enlazar
gdb -tui saludo                  # sesión gdb en modo text-user-interface
```

```
   list              # localizar línea "mov $1,%rdi" y ponerle breakpoint
   break 9           # equivalente gráfico ddd: cursor a izq.línea y stop
   info break        # equiv.gráf: Source->Brkpts. Notar address 0x4000b7

   run               # eq.gr: Program->Run /(View->CmdTool->)botonera->Run
   disassemble       # eq.gr: View->MachCodeWin
   print $rip              # Notar dirección break = RIP=0x4000b7
   print $rax              # Notar RAX=1, pero RDI=0 aún
   info registers          # eq.gr: Status->Registers
   stepi                   #
   p $rip                  # EIP sigue avanzando (p=print)
   p $rdi                  # Notar RDI=1 ahora
   si                      #                   (si=stepi)
   p/x $rsi                # Notar RSI=0x6000df > RIP
   disas _start            # Notar traducción ASM->LM $saludo, longsaludo

   x/32cb &saludo    # eq.gr: Data->Memory->Examine 32 char bytes &saludo
   x/32xb &saludo    # Print p/probar (cambiar a hex bytes)/Display p/fijo
   x/s    &saludo    # eq.gr: Data->Memory->Examine 1 string bytes &saludo
   p (char*) &saludo
   p(long)longsaludo
   x/1dg &longsaludo       # eXamine para ver dirección de inicio y valor
   x/1xg &longsaludo       # comprobar ordenamiento little-endian
   x/8xb &longsaludo
   si                # Comprobar regs EAX,RDI,RSI,RDX = 1,1,0x6000df,28
   info reg          #                (write,stdout,&saludo,longsaludo)
   disas             # $saludo=$0x6000df(inm), longs=0x6000fb(dir), %rdx=28(reg)
   si                # Se escribe mensaje en pantalla (View->GDB Console)
3x si / cont         # eq.gr: Pulsar cont o clickar 3 stepi para exit(0)

   clear 9                 # otra ejecución: parar justo antes del final
   break 16                # localizar la 2a syscall y ponerle bkpt
   info break
   run
   set $rdi=1              # y cambiar código de retorno sobre la marcha
   stepi / cont

   cl 16                   # otra ejecución: parar antes de imprimir
   br 12                   # localizar la primera syscall y ponerle bkpt
   info br
   run
   print      saludo             # 'saludo' has unknown type; cast it to
   p    (int) saludo             # interpreta 4B "Hola" (4 códigos ASCII)
   p /x (int) saludo             # como un entero 0x616c6f48, ver Apénd.1
   p /x(char) saludo             # typecast a char 1B 'H'
   p   (char) saludo             # hex 0x48, decimal 72

   p         &saludo             # dirección de memoria
   p  (char*)&saludo             # typecast a char* = string
   p  (char*)&saludo+13          # saltarse 13 letras, localizar \n
   p*((char*)&saludo+13)         # cambiarlo por '-'
   set var *((char*)&saludo+13)='-'
   print     (char*)&saludo      # comprobar cambio en memoria
   cont                          # comprobar cambio en ejecución
   quit
```

Figura 11: ensamblado, enlazado, y sesión de depuración usando gbd –tui

Como vemos, la forma general de usar un depurador es escoger un punto de parada (o varios), lanzar la ejecución, comprobar valores de variables y registros cuando el programa se detenga (al encontrar algún punto de parada), y seguir ejecutando (paso a paso, continuar ejecución normal, o volver a empezar desde el principio). Observar qué fácilmente se modifica el *string* para que sea una línea, no 2.

Los *frontends* como ddd o entornos de desarrollo como Eclipse presentan al usuario un interfaz gráfico más intuitivo, como se ve en las siguientes figuras, aunque internamente usen gdb como depurador. Pueden usarse para la asignatura, pero el profesorado no es responsable de su funcionamiento.

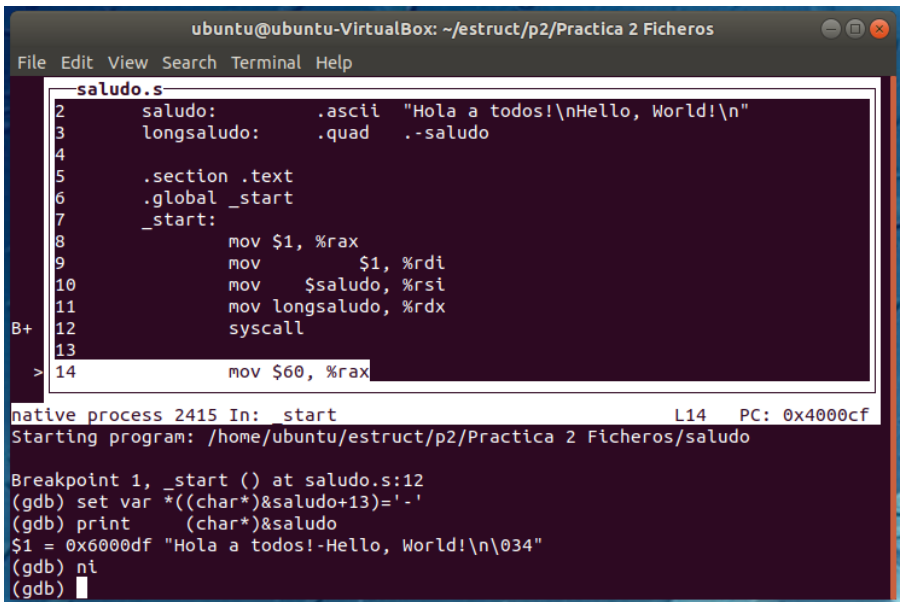

Figura 12: gbd –tui saludo. Observar qué fácilmente se modifica el string para que sea una línea, no 2.

En la Figura 12 se observa que ha sido necesario pulsar <Ctrl>-L para refrescar la pantalla, que quedó un poco trastocada tras ejecutar `syscall`. Al refrescar la pantalla se recupera el marco de decoración del código fuente, pero se pierde el texto escrito por la propia llamada al sistema.

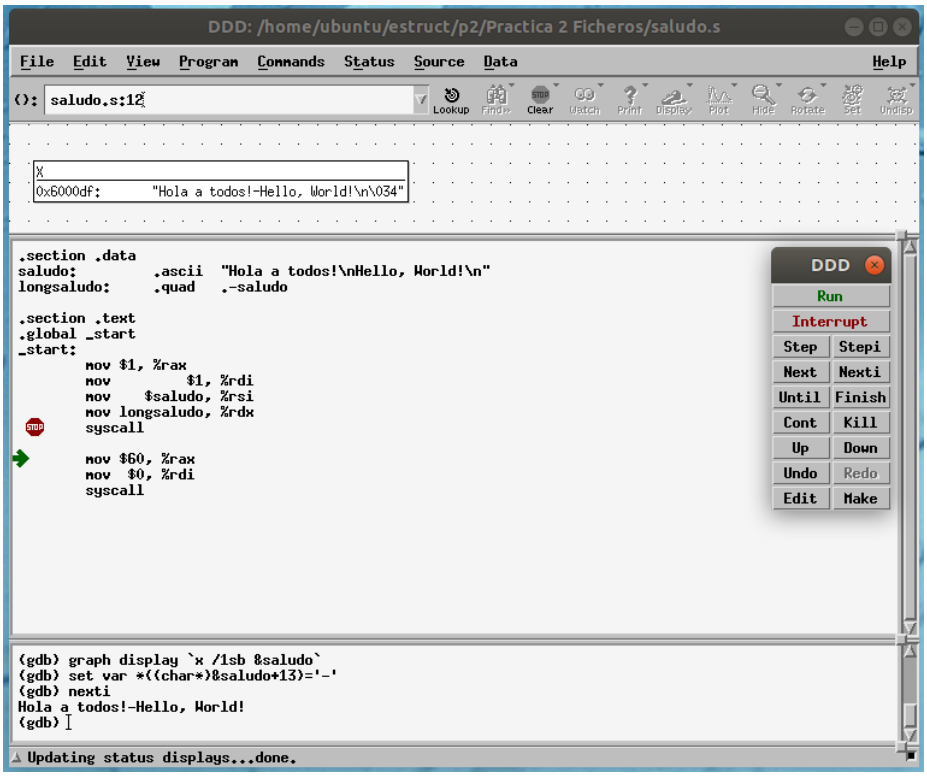

Figura 13: ddd saludo. Sigue siendo necesario usar el mismo comando en modo texto para modificar el string.

En la Figura 13 se ha utilizado *Data>Memory>Examine 1 string bytes from &saludo* para mostrar el *string*, aunque para modificarlo siga siendo necesario usar el mismo comando en modo texto. Para continuar la ejecución se ha usado el botón *Nexti*, que ha sido traducido al correspondiente comando en modo texto. La botonera agrupa los comandos más habituales. También son de uso frecuente las opciones de menú *View>**, *Program>** (*Run in Execution Window* es útil para obtener la salida en un `xterm` separado), *Status>Registers* y *Data>Memory*. Con *Edit>Preferences>Helpers* puede cambiarse el `xterm` a otro programa preferido.

Con el entorno Eclipse querremos utilizar *File>Import>C/C++ Executable* para depurar el ejecutable `saludo` que ya habíamos ensamblado con información de depuración usando `as -g`. Conviene configurar el lanzador en la pestaña *Debugger>Stop on startup at:* `main`, cambiándolo a `_start`. El *Wizard* nos permite cambiar automáticamente a la ***Perspectiva*** de *Depuración* (tal vez tengamos que cerrar la perspectiva *Java*), en donde disponemos de los botones habituales (*Debug*, *Run*, *Resume*, *Terminate*, *Step Into*, *Step Over*, etc). El modo *Instruction Stepping Mode* es útil cuando se depura código desensamblado para el cual no se dispone de fuente.

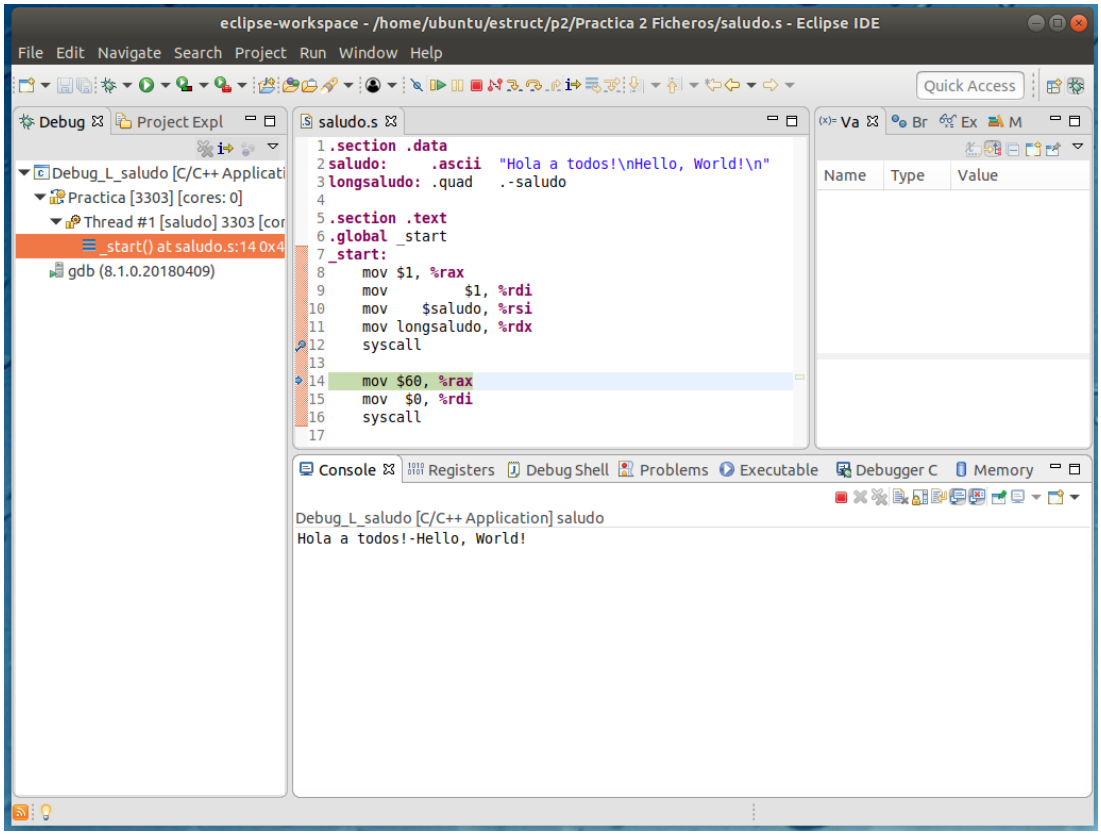

Figura 14: eclipse. Seguimos usando el mismo comando para cambiar el string.

En la Figura 14 se ha añadido un breakpoint en la primera `syscall`, se ha continuado con *Resume*, se ha añadido un *Memory>Memory Monitor* para `&saludo`, se ha vuelto a utilizar el mismo comando texto en la consola *Debugger C*, y al pulsar *Step Over* se ha obtenido el mismo resultado. Seguramente interesa consultar las ***Vistas*** *Console*, *Registers*, *Memory*, *Breakpoints*, etc.

Por experiencia con promociones anteriores (léase bomba digital), y viendo las versiones de los paquetes oficiales de ambos entornos en Ubuntu (quieta en ddd-3.3.12-5, atrasada por eclipse-3.8.1-11, sólo como paquete *snap* eclipse-4.8), el depurador que utilizaremos oficialmente en estas prácticas será `gdb -tui`, aunque no prohibimos que se usen ddd o Eclipse, pero en ningún caso será preocupación de esta asignatura resolver bloqueos del ddd al depurar la bomba digital (si es que los hay), o configurar Eclipse para enlazar o ensamblar o incluso compilar nuestros programas. Editaremos nuestros programas fuente con `vim` o `gedit`, los pasaremos a ejecutable con `as`, `ld` y/o `gcc`, y los depuraremos con `gdb -tui`, *permitiendo* el uso de ddd o Eclipse si el estudiante conoce y desea usar esos *frontends*.

4 Segundo programa ASM: Llamadas a funciones (libC, usuario)

Es conveniente dividir el código de un programa entre varias funciones, de manera que al ser estas más cortas y estar centradas en una tarea concreta, se facilita su legibilidad y comprensión, además de poder ser reutilizadas si hacen falta en otras partes del programa. En la Figura 15 se muestra un programa ensamblador con una función (subrutina) que calcula la suma de una lista de enteros de 32 bits. La dirección de inicio de la lista y su tamaño se le pasa a la función a través de los registros RBX y ECX. El resultado se devuelve al programa principal a través del registro EAX. Se preservan los demás registros.

Conocer el funcionamiento de la pila (*stack*) es fundamental para comprender cómo se implementan a bajo nivel las funciones. La pila se utiliza (en llamadas a subrutinas) para guardar la dirección de retorno, para almacenar las variables locales (si el compilador no puede optimizar el uso de registros para todas ellas), y para pasar argumentos (según la convención de llamada: por ejemplo SysV AMD64 usa pila a partir del 7º argumento). Las instrucciones PUSH y POP (Apéndice 2, Tabla 3) y las llamadas y retornos de subrutinas (CALL, RET, INT, IRET, Tabla 7) utilizan de forma implícita la pila, que es la zona de memoria adonde apunta el puntero de pila RSP. La pila crece hacia direcciones inferiores de memoria, y RSP apunta al último elemento insertado (tope de pila), de manera que PUSH primero decrementa RSP en el número de posiciones de memoria que ocupe el dato a insertar (8B o 2B, aunque nosotros sólo apilaremos palabras *quad* de 8B), y luego escribe ese dato en las posiciones reservadas, a partir de donde apunta RSP ahora. Similarmente POP primero lee del tope de pila, guardando el valor en donde indique su argumento, y luego incrementa RSP. Por su parte, CALL guarda la dirección de retorno en pila antes de saltar a la subrutina indicada como argumento, y RET recupera de pila la dirección de retorno.

Ejecutar el programa de la Figura 15 paso a paso con gdb `-tui` (o ddd o Eclipse) y comprobar que la pila funciona como se ha explicado; en particular, que CALL guarda la dirección de retorno, RET recupera el contador de programa, PUSH almacena temporalmente un valor, y POP lo recupera posteriormente.

```
# suma.s:     Sumar los elementos de una lista
#             llamando a función, pasando argumentos mediante registros
#             comprobar con "./suma; echo $?" o con depurador gdb/ddd

# SECCIÓN DE DATOS (.data, variables globales inicializadas)
.section .data
lista:        .int 1,2,10, 1,2,0b10, 1,2,0x10  # ejs. binario 0b / hex 0x
longlista:    .int (.-lista)/4    # . = contador posiciones. Aritm.etiq.
resultado:    .int  0

# formato:    .asciz "suma = %u = 0x%x hex\n"  # fmt para printf() libC
# el string "formato" sirve como argumento a la llamada printf opcional
# opción: 1) no usar printf, 2)3) usar printf/fmt/exit, 4) usar tb main
# 1) as  suma.s -o suma.o
#    ld  suma.o -o suma                                      1232 B
# 2) as  suma.s -o suma.o                                    6520 B
#    ld  suma.o -o suma -lc -dynamic-linker /lib64/ld-linux-x86-64.so.2
# 3) gcc suma.s -o suma –no-pie –nostartfiles                6544 B
# 4) gcc suma.s -o suma –no-pie                              8664 B

# SECCIÓN DE CÓDIGO (.text, instrucciones máquina)
.section .text                  # PROGRAMA PRINCIPAL
_start: .global _start          # se puede abreviar de esta forma
# main: .global   main          # Programa principal si se usa C runtime

      call  trabajar            # subrutina de usuario
#     call  imprim_C            # printf()  de libC
      call  acabar_L            # exit()   del kernel Linux
#     call  acabar_C            # exit()    de libC
      ret

trabajar:
      mov     $lista, %rbx      # dirección del array lista
      mov  longlista, %ecx      # número de elementos a sumar
      call suma                 # llamar suma(&lista, longlista);
      mov  %eax, resultado      # salvar resultado
      ret

# SUBRUTINA: int suma(int* lista, int longlista);
# entrada:   1) %rbx = dirección inicio array
#            2) %ecx = número de elementos a sumar
# salida:       %eax = resultado de la suma
suma:
      push     %rdx             # preservar %rdx (se usa como índice)
      mov  $0, %eax             # poner a 0 acumulador
      mov  $0, %rdx             # poner a 0 índice
bucle:
```

```
bucle:
       add  (%rbx,%rdx,4), %eax   # acumular i-ésimo elemento
       inc   %rdx                 # incrementar índice
       cmp   %rdx,%rcx            # comparar con longitud
       jne    bucle               # si no iguales, seguir acumulando

       pop   %rdx                 # recuperar %rdx antiguo
       ret

#imprim_C:
#      si se usa esta subrutina, usar también la línea que define formato
#      se puede linkar con ld -lc -dyn ó gcc -nostartfiles, o usar main
#      mov  $formato,  %rdi       # traduce resultado a decimal/hex
#      mov  resultado, %esi       # versión libC de syscall __NR_write
#      mov  resultado, %edx       # ventaja: printf() con fmt "%u" / "%x"
#      mov         $0, %eax       # varargin sin xmm
#      call printf                # == printf(formato,resultado,resultado)
#      ret

acabar_L:                         #   void _exit(int status);
       mov       $60,  %rax       #   exit: servicio 60 kernel Linux
       mov  resultado, %edi       # status: código a retornar (la suma)
       syscall                    # == _exit(resultado);
       ret

#acabar_C:                        #   void  exit(int status);
#      mov  resultado, %edi       # status: código a retornar (la suma)
#      call exit                  # ==  exit(resultado)
#      ret
```

Figura 15: suma.s: ejemplo de llamada a subrutina (paso de parámetros por registros RBX/RCX)

Con esto concluye la sección de Seminario de esta práctica. Para la parte de trabajo personal se sugerirá mejorar este programa para que no pierda bits al ir sumando números (si son muchos, o grandes). También se pedirá dividir por el tamaño de la lista para calcular la media. Se pedirá realizarlo sobre registros de 32 bits (`.int` normales de 4B como los que hemos usado hasta ahora), usando aritmética multi-precisión, para recordar la diferencia entre pensar que los datos tienen signo o que no lo tienen. Por último, como la plataforma que usamos dispone de registros de 64 bits, podemos repetir el cálculo en uno de ellos y comprobar si nuestro programa en doble precisión produce el mismo resultado.

Desarrollo de las Prácticas en [Windows + VirtualBox +] Ubuntu 18.04.LTS

Como ya se ha comentado en clase de Teoría [1] (Presentación p.34 y Tema 1 p.64), en el laboratorio estamos usando Ubuntu 18.04.LTS 64bit. En un portátil con Windows se puede optar por instalar Ubuntu en una partición separada, o instalar algún software de virtualización como por ejemplo VirtualBox (si no se tenía previamente) y crear una máquina virtual con dicho Ubuntu.

A la instalación por defecto de Ubuntu se le podrían añadir (usando el comando `apt`, o tal vez instalándose el *frontend* gráfico *Synaptic*) los siguientes paquetes cuya presencia asumimos: `g++` (la *suite* del compilador), `ghex` (editor hexadecimal), `make` (para recompilar las *Guest Additions* de VirtualBox). Si se desea, también puede interesar instalar `gcc-multilib` (para recompilar aplicaciones de 32 bits que necesiten estas librerías de compatibilidad), `ddd` (*frontend* depurador gráfico), `eclipse-4.8` (paquete *snap* en *Ubuntu Software*, evitar el paquete ubuntu *Development (universe) eclipse-3.8*). Probablemente Eclipse requerirá instalar previamente `default-jre`, y posteriormente, entrando en el propio entorno Eclipse, *Help>Eclipse MarketPlace>Find CDT>Eclipse C/C++ IDE CDT 9.4 (Oxygen.2)*. Similarmente, para la opción *Run in Execution Window* de `ddd` convendrá instalar `xterm`. Para la práctica de cache también querremos instalar `gnuplot-x11`. El *firewall* se puede comprobar/activar con el comando "`sudo ufw status/enable`".

De esta forma se pueden repetir los tutoriales de prácticas, como este que acabamos de completar, de forma independiente. Al estar las sesiones de tutorial transcritas en su totalidad (incluso se han transcrito los equivalentes en modo texto de los comandos gráficos ejecutados en `ddd`), se pueden probar antes de venir al laboratorio, se pueden repetir después de haber asistido al tutorial, y se pueden repasar en cualquier momento, independientemente de las sesiones presenciales de laboratorio.