**Laboratorio Sesión 02: Uso del debugger**

El objetivo de esta sesión es introducir el uso del debugger y las principales herramientas de depuración de código.

**Compilar a código máquina de 32 bits**

En la primera práctica vimos el comando de compilación estándar de gcc:

gcc -o miprograma miprograma.c

Este comando compila para el sistema operativo en el que estamos trabajando (en este caso x86 de 64 bits o AMD64). Sin embargo, en esta asignatura estudiamos el ensamblador x86 de 32 bits (o IA32). Para poder compilar hacia código máquina de 32 bits en nuestro sistema operativo tendremos que añadir la opción -m32:

gcc -m32 -o miprograma miprograma.c

El código generado se puede ejecutar directamente ya que la arquitectura es compatible con el código IA32 y, además, cuando lo depuréis veréis las instrucciones estudiadas en clase.

**Debuggers**

Cuando se trata de depurar un programa, habitualmente se usa un programa de depuración. Hay gente que prefiere depurar a base de realizar printf en el código del programa. Aunque es una opción que a veces funciona y a veces no, por lo que no es confiable. Así pues, una de las primeras decisiones que hay que tomar es que tipo de programa de depuración vamos a utilizar. Aunque podéis usar cualquiera, en esta asignatura vamos a ver el funcionamiento de dos de estos programas: gdb y ddd. El primero es un programa en línea de comandos que permite trabajar con un terminal texto y que representa el tipo de depuradores que podremos encontrar en entornos poco desarrollados (como los entornos de desarrollo de software para procesadores específicos, por ejemplo). El segundo es un programa bajo entorno gráfico que tiene las mismas funcionalidades, pero con una aproximación más visual e intuitiva. Dentro de ddd tenéis una consola que os permite introducir las ordenes de gdb.

**Compilar con información de depurado**

El primer paso para poder depurar adecuadamente un programa consiste en compilarlo incluyendo en el ejecutable información que nos ayude a poder depurarlo (por ejemplo, que permita mostrar el código original en C además del código ensamblador). También es bueno no permitir que el compilador optimice el programa a la hora de depurar ya que la correspondencia entre el código ejecutado y el código escrito sería mayor. Así pues, para depurar nuestra orden básica de compilación sería:

gcc -m32 -gstabs+ -o miprograma miprograma.c

A continuación, invocaremos el programa de depurado deseado:

gdb miprograma / ddd miprograma

También podemos observar directamente el código ensamblador del programa compilado:

objdump -d miprograma

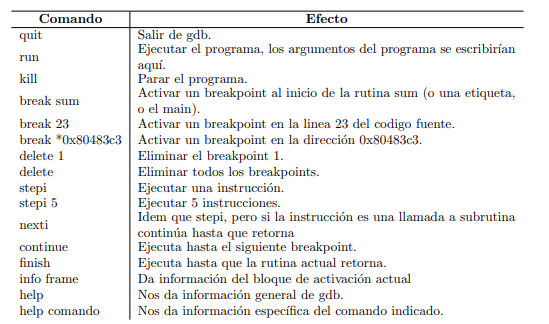
A partir de aquí la idea general es activar un conjunto de breakpoints cercanos a los puntos que nos interesen en el código. Estos puntos pueden ser las primeras instrucciones de una subrutina o una determinada dirección de memoria. Cuando el programa pasa por uno de los breakpoints, el programa se para (antes de ejecutar la sentencia asociada al breakpoint) y devuelve control al usuario. Desde ese punto, podemos examinar el contenido de los registros y el contenido de la memoria. También podemos ejecutar el programa instrucción a instrucción, por grupos de instrucciones o justo hasta el siguiente breakpoint. Lo importante es comprobar que el conjunto de instrucciones que estamos examinando realmente dan el resultado esperado.

Documentación de gdb y ddd

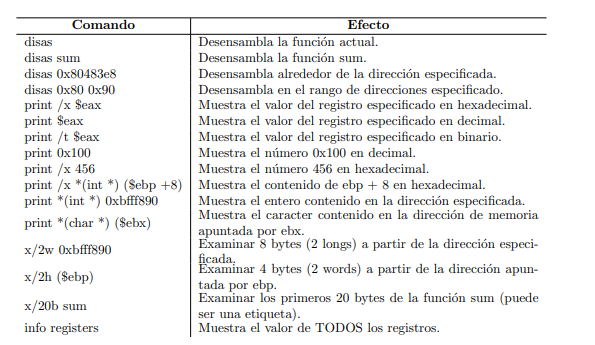
La documentación de referencia para el uso de gdb es el manual de gdb:

“Debugging with GBD”.

A continuación, la siguiente tabla muestra algunas de las ordenes más útiles de gdb relacionadas con los breakpoints y con la ejecución de instrucciones paso a paso:



En la siguiente tabla mostramos algunos de los comandos de gdb para examinar código y datos:



**Estudio Previo**

**1.Explica en detalle qué hacen las siguientes instrucciones. Si están sintácticamente mal escritas, indica el motivo:**

movl $1, %eax: mueve el literal 1 al registro %eax (de 32 bits).

movl 1, %eax: mueve el contenido de la dirección 1 de la memoria (M[1]) al registro %eax.

movl $1, eax: mueve el literal 1 a la dirección de memoria referenciada por la etiqueta eax.

movl 1, eax: mueve el contenido de la dirección 1 de la memoria (M[1]) a la dirección de memoria referenciada por la etiqueta eax.

**2. Dada la siguiente secuencia de instrucciones:**

cmpl %eax, %ebx

jge fin

¿en qué condiciones se efectúa el salto?

Si %ebx es mayor o igual (con signo) que %eax salta a la etiqueta fin.

**3. Dada la siguiente secuencia de instrucciones:**

cmpl %ebx, %eax

jge fin

¿en qué condiciones se efectúa el salto?

Si %eax es mayor o igual (con signo) que %eax salta a la etiqueta fin.

**4. Dado el siguiente código en C donde a, b y i son variables globales enteras:**

for (i = 0; i < 300; i++) {

if (a == b) {

b = b + (b - a) / 2;

}

}

Realiza la traducción a código ensamblador.

Movl $0, %eax //i = 0

For: cmpl $300, %eax

Jge Fi\_for //jump if i >= 300

Cmpl a, b

Jna End\_if //jump if a != b

Movl b, %ebx //%ebx <- b

Subl a, %ebx //%ebx <- %ebx – a

Sarl 1, %ebx //%ebx / 2

Addl, %ebx, b //%ebx + b

End\_if: incl %eax

Jmp for

Fi\_for:

**5. Dado el siguiente segmento de código ensamblador y las mismas variables del apartado anterior:**

80483fd: jmp 8048433

80483ff: mov a, %edx //%edx <- a

8048405: mov b, %eax //%eax <- b

804840a: cmp %eax, %edx

804840c: je 8048426 //jump if %edx igual a %eax

804840e: mov b, %edx //%edx <- b

8048414: mov a, %eax //%eax <- a

8048419: sub %eax, %edx //%edx <- %edx - %eax (b = b – a)

804841b: mov %edx, %eax //%eax <- %edx

804841d: add %eax, %eax //%eax <- %eax + %eax

804841f: add %edx, %eax //%eax <- %eax + %eax

8048421: mov %eax, a //a <- %eax

8048426: mov i, %eax //%eax <- i

804842b: sub $0x1, %eax //%eax = %eax – 0x1 (--i)

804842e: mov %eax, i //i <- %eax

8048433: mov i, %eax //%eax <- i

8048438: test %eax, %eax

804843a: jg 80483ff

Realiza la traducción a código C.

While (i > 0) {

if (a != b) {

a = 3(b – a);

}

--i;

}

**6. Explica cuál es la diferencia entre un breakpoint y un watchpoint.**

Los Breakpoints detienen el programa en la línea donde fue establecido el breakpoint, mientras que los Watchpoints permiten detener la ejecución de un programa cuando se cumpla determinada condición.

**7. Explica cómo se puede hacer en ddd que un breakpoint sólo se pare cuando se ha ejecutado 15 veces.**

Primero de todo, poner el breakpoint en la línea de código donde se quiera que pare, luego ir a Properties y poner el Ignore Count a 15.

**8. Explica cómo se puede hacer en ddd para modificar el contenido de una dirección de memoria.**