Laboratorio Sesión 02: Uso del debugger

Objetivo

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

Conocimientos Previos

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, dado que no se puede usar siempre implicará un 0 en estas prácticas :-)

Así pues, una de las primeras decisiones que hay que tomar es qué 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* (podéis usarlos indistintamente y os animamos a probar los dos). El primero es un programa en l´ınea 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 será mayor. Así pues, para depurar nuestra orden básica de compilación será:

```
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".

Esta documentación puede encontrarse en la web de GNU. 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:

Comando	Efecto
quit	Salir de gdb.
run	Ejecutar el programa, los argumentos del programa se escribirían aquí.
kill	Parar el programa.
break sum	Activar un breakpoint al inicio de la rutina sum (o una etiqueta, o el main).
break 23	Activar un breakpoint en la linea 23 del codigo fuente.
break *0x80483c3	Activar un breakpoint en la dirección 0x80483c3.
delete 1	Eliminar el breakpoint 1.
delete	Eliminar todos los breakpoints.
stepi	Ejecutar una instrucción.
stepi 5	Ejecutar 5 instrucciones.
nexti	Idem que stepi, pero si la instrucción es una llamada a subrutina continúa hasta que retorna
continue	Ejecuta hasta el siguiente breakpoint.
finish	Ejecuta hasta que la rutina actual retorna.
info frame	Da información del bloque de activación actual
help	Nos da información general de gdb.
help comando	Nos da información específica del comando indicado.

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

Comando	Efecto
disas	Desensambla la función actual.
disas sum	Desensambla la función sum.
disas 0x80483e8	Desensambla alrededor de la dirección especificada.
disas 0x80 0x90	Desensambla en el rango de direcciones especificado.
print /x $eax	Muestra el valor del registro especificado en hexadecimal.
print $eax	Muestra el valor del registro especificado en decimal.
print /t $eax	Muestra el valor del registro especificado en binario.
print 0x100	Muestra el número 0x100 en decimal.
print /x 456	Muestra el número 456 en hexadecimal.
print /x *(int *) ($ebp +8)	Muestra el contenido de ebp + 8 en hexadecimal.
print *(int *) 0xbfff890	Muestra el entero contenido en la dirección especificada.
print *(char *) ($ebx)	Muestra el caracter contenido en la dirección de memoria apuntada por ebx.
x/2w 0xbfff890	Examinar 8 bytes (2 longs) a partir de la dirección especificada.
x/2h ($ebp)	Examinar 4 bytes (2 words) a partir de la dirección apuntada por ebp.
x/20b sum	Examinar los primeros 20 bytes de la función sum (puede ser una etiqueta).
info registers	Muestra el valor de TODOS los registros.

El *ddd* por su lado es un programa muy visual e intuitivo. Su manual de referencia también se puede encontrar en la página web de GNU. El comando *info* también os puede servir de consulta rápida.

Estudio Previo

Como trabajo previo de la sesión hay que realizar las siguientes tareas:

1. Explica en detalle qué hacen las siguientes instrucciones. Si están sintácticamente mal escritas, indica el motivo:
```
movl $1, %eax
movl 1, %eax
movl $1, eax
movl 1, eax
```

2. Dada la siguiente secuencia de instrucciones:
```
cmpl %eax, %ebx
jge fin
```
¿en qué condiciones se efectua el salto?

3. Dada la siguiente secuencia de instrucciones:
```
cmpl %ebx, %eax
jge fin
```
¿en qué condiciones se efectua el salto?

4. Dado el siguiente código en C donde `a`, `b` y `i` son variables globales enteras:
```
for (i=0;i<100;i++)
  if (a<b)
    a=a+(b-a)/2;
```
Realiza la traducción a código ensamblador.

5. Dado el siguiente segmento de código ensamblador y las mismas variables del apartado anterior:
```
 80483fd:       jmp    8048433
 80483ff:       mov    a,%edx
 8048405:       mov    b,%eax
 804840a:       cmp    %eax,%edx
 804840c:       je     8048426
 804840e:       mov    b,%edx
```

```
8048414:        mov     a,%eax
8048419:        sub     %eax,%edx
804841b:        mov     %edx,%eax
804841d:        add     %eax,%eax
804841f:        add     %edx,%eax
8048421:        mov     %eax,a
8048426:        mov     i,%eax
804842b:        sub     $0x1,%eax
804842e:        mov     %eax,i
8048433:        mov     i,%eax
8048438:        test    %eax,%eax
804843a:        jg      80483ff
```

Realiza la traducción a código C.

6. Explica cuál es la diferencia entre un *breakpoint* y un *watchpoint*.
7. Explica cómo se puede hacer en *ddd* que un *breakpoint* sólo se pare cuando se ha ejecutado 15 veces.
8. Explica cómo se puede hacer en *ddd* para modificar el contenido de una dirección de memoria.

Trabajo a realizar durante la Práctica

Para hacer esta práctica deberéis trabajar con el debugger sobre los dos códigos que os proporcionamos compilados en un solo programa. Cuando lo tengáis listo haced las siguientes tareas teniendo en cuenta que, si no se os dice lo contrario, cada apartado se realiza sin volver a ejecutar el programa desde el principio.

1. Poned un *breakpoint* en la primera l´ınea en alto nivel de la rutina `"Color"`. Ejecutad el programa de ejemplo hasta entrar por primera vez dentro de dicha rutina. Averiguad cuánto vale el elemento 4 del vector variable local `mano` en ese momento (el elemento `mano[4]` es una estructura y por tanto tiene dos valores).
2. Ejecutad 45 instrucciones más en ensamblador sin entrar en ninguna subrutina. Averiguad cuál es el valor del registro `edx` en ese punto.
3. Seguid ejecutando el programa hasta la tercera vez en que se entra en la rutina `"Color"` (contando desde el principio). ¿Cuánto vale ahora el elemento `mano[4]`?
4. Ejecutad el programa desde el principio hasta justo antes de ejecutar la primera instrucción en alto nivel de la rutina `"Pareja"`. ¿A qué dirección de memoria apunta en este momento el registro `ebp`? ¿Qué valor hay en la posición de memoria apuntada por el registro `ebp+8`? ¿Y dentro de la posición de memoria apuntada por el contenido de la posición de memoria anterior? Cambiad dicho valor por un `3`. ¿Varía la salida de la función `"Pareja"` en ese caso?
5. Ejecutad el programa desde el principio y averiguad cuál es el valor del elemento `mano[4]` la 25 vez que se entra en la función `Ordenar` (mano ahora es variable local de la función `Ordenar`).
6. Averiguad cuál es el valor del elemento `manita[4]` (manita es una variable global y sus elementos también son estructuras con dos valores) la primera vez que la variable global `m` vale 5.
7. Estudiad la rutina `PierdeTiempo` que, en función de una comparación, realiza una suma en la variable que hay almacenada en la posición `ebp-12`. Modificad su código ensamblador, compilad de nuevo y averiguad el nuevo resultado en los siguientes supuestos:
   - Cuando la suma se ejecuta con la condición contraria a la que hay programada.
   - Cuando la suma se ejecuta siempre (sin condición).

Nombre: Liang Liang Chen Xu Grupo: 12

Nombre:_____

Hoja de respuesta al Estudio Previo

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

```
movl $1, %eax
```

Muevo el valor del op1(1) al contenido del op2(registro eax), datos de 32 bits long.

```
movl 1, %eax
```

Muevo el contenido de M[1] al registro %eax, datos de tipo long.

```
movl $1, eax
```

Muevo el valor del op1(1) a la dirección de memoria con la etiqueta eax, datos de tipo long.

```
movl 1, eax
```

Esta instrucción no es realizable porque no pueden haber dos operandos de la misma instrucción accediendo a memoria.

2. Dada la siguiente secuencia de instrucciones:

```
cmpl %eax, %ebx
jge fin
```

¿en qué condiciones se efectua el salto?

Si el contenido de %ebx es más grande o igual que el de %eax

3. Dada la siguiente secuencia de instrucciones:

```
cmpl %ebx, %eax
jge fin
```

¿en qué condiciones se efectua el salto?

Si el contenido de %eax es más grande o igual que el de %ebx

4. La traducción a código ensamblador del código C es:

```
     movl $0, i
for: cmpl $100, i
     jge fi_for
     movl b, %eax // eax = b
     cmpl %eax, a //a – b
     jge fi
     subl a, %eax // b-a se guarda en b
     sarl $1, %eax ; (b-a)/2 se guarda en b
     addl %eax, a
fi: incl i
    jmp for
fi_for:
```

5. La traducción a código C del código ensamblador es:

```
while(i > 0) {
  int edx = a;
  int eax = b;

if (eax == edx) {
   eax = i;
  --eax;
  i = eax;
 eax = i;
}

else {
  edx = b;
 eax = a
 edx = edx – eax;
 eax = edx;
 eax += eax;
 eax += edx;
 a = eax;
 eax = i;
  --eax;
  i = eax;
 eax = i;
}
}
```

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

El breakpoint indica un punto en e código en donde el programa debe detenerse al ejecutarse. En cambio, el watchpoint se activa cuando el valor de una variable especifica cambia o su ubicación en memoria.

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

Se establece el breakpoint en la línea deseada, click derecho en el icono del breakpoint -> properties -> ignore count -> 15 -> Apply.

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

Hay que tener en cuenta que el programa tiene que estar en un breakpoint o en un punto del programa en el que se pueda acceder a memoria. Existen dos maneras : Desde la GUI de ddd, ir a la variable que queremos modificar en el código, darle a display de esa variable, ir a la ventana emergente y darle a set value y cambiarle el valor(o a partir de la dirección de memoria).

Nombre:_____ Grupo:_____

Nombre:_____

Hoja de respuestas de la práctica

1. La primera vez que se entra en la rutina "`Color`" el primer campo de la variable `mano[4]` vale: [ ] y el segundo: [ ]
2. Después de ejecutar 45 instrucciones en ensamblador más sin entrar en ninguna sub-rutina el registro `edx` vale: [ ]
3. La tercera vez (contando desde el principio) que se entra en la rutina "`Color`" el primer campo de la variable `mano[4]` vale: [ ] y el segundo vale: [ ]
4. Justo antes de la primera instrucción en alto nivel de la rutina "`Pareja`" en su primera ejecución:
   - `ebp` apunta a: [ ]
   - En la posición de memoria apuntada por el registro `ebp+8` hay: [ ]
   - En la posición de memoria apuntada por el contenido de la posición de memoria anterior hay: [ ]
   - Con el valor original la salida de "`Pareja`" es: [ ]
   - Si el valor cambia a `3` la salida de "`Pareja`" es: [ ]
5. La 25 vez que se entra en la función `Ordenar`, `mano[4]` vale: [ ] y [ ]
6. La primera vez que la variable `m` vale 5, `manita[4]` vale: [ ] y [ ]
7. El resultado de la rutina `PierdeTiempo` es:
   - Cuando la suma se ejecuta con la condición contraria a la que hay programada: [ ]
   - Cuando la suma se ejecuta siempre (sin condición): [ ]