Herramienta de depuración GNU Debugger (GDB)

Programación para Ingeniería Eléctrica

April 30, 2014

1 Introducción

Llamamos *depurador*, *debugger*, a la herramienta o conjunto de herramientas que nos permiten analizar el comportamiento de un programa en tiempo de ejecución, de modo de poder detectar fallas que de otro modo son difíciles de diagnosticar.

En general, y el GDB no es la excepción, el debugger carga en memoria al programa a analizar. Luego permite establecer puntos de inspección, llamados *breakpoints*, y condiciones iniciales (por ejemplo, argumentos en la línea de comandos). Acto seguido, el debugger es capaz de ejecutar el programa de manera incremental, ya sea paso a paso (sentencia por sentencia, instrucción por instrucción), o hasta llegar a los breakpoints. En todo momento que no se esté ejecutando el programa, uno puede interactuar con el depurador para consultar el estado del programa (memoria, registros, etc.) e incluso evaluar expresiones, de modo de poder hacer verificaciones más complejas, antes de seguir adelante.

El depurador también puede utilizarse como herramienta de análisis *forense*, permitiendo analizar las causas de la terminación inesperada de un programa a partir de la información que éste dejó al sistema antes de cerrarse (lo que suele denominarse *volcado de nucleo* o core dump). Un caso típico es averiguar por qué se dió un *segmentation fault*.

Este pequeño tutorial mostrará cómo realizar las tareas básicas de depuración con el GDB. EL GDB es un programa muy complejo y rico, con muchísimos comandos muy sofisticados que no están cubiertos aquí. Para ver la documentación oficial completa por favor dirigirse a `http://www.sourceware.org/gdb/documentation/`.

2 Corriendo un programa con el GDB

Consideremos el siguiente programa:

```
#include <stdio.h>

int sumatoria(int a) {
  int i = 0;
  int s = 0;
  for (i = 0; i < a; i++) {
    s += i;
  }
  return s;
}

int main() {
  int x = 5;
  x += sumatoria(10);
  x -= sumatoria(3);
  return x;
```

```
}
```

Lo primero que debemos hacer es compilar nuestro programa con *información de depuración*, de modo que el GDB pueda saber los nombres de las variables y las funciones, tal cual las escribimos nosotros (de otro modo, dichos nombres se pierden al compilar el código, tornando el resultado ilegible):

```
gcc -O0 -ggdb -o debug debug.c
```

Una vez compilado el programa, GDB puede ser invocado para depurar dicho programa. La linea de comandos:

```
gdb ./debug
```

invoca a GDB y le instruye que cargue al programa a depurar, `debug`. Esto nos coloca en la consola de GDB, que comienza con una carátula similar a la siguiente:

```
GNU gdb (Ubuntu 7.7-0ubuntu3) 7.7
Copyright (C) 2014 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type ‘‘show copying’’
and ‘‘show warranty’’ for details.
This GDB was configured as ‘‘x86_64-linux-gnu’’.
Type ‘‘show configuration’’ for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type ‘‘help’’.
Type ‘‘apropos word’’ to search for commands related to ‘‘word’’...
Reading symbols from ./debug...done.
(gdb)
```

La última linea es un *prompt*, tal como el del terminal común, indicando que se está a la espera de comandos. En principio la interacción con GDB es a través de líneas de comando especiales, pero también tiene la capacidad de ser controlado desde interfaces gráficas, algo que muchos entornos de desarrollo utilizan para depurar programas, usando GDB como *back-end*. En este tutorial veremos cómo utilizar GDB desde esta línea de comandos, algo que es muy sencillo y muy versátil.

3 Uso básico

Puntos de parada o breakpoints El depurador puede usarse de dos maneras: hacer una ejecución controlada de un program, viendo la evolución de su estado paso a paso, o bien correr el programa y analizar el estado en caso de que se termine inesperadamente (excepción). En el primer caso, lo primero que se hace es establecer puntos de inspección o de parada (breakpoints). Estos son lugares en el código en donde queremos detener la ejecución y, recién a partir de allí, hacer una ejecución paso a paso; claramente, seria muy poco práctico correr *todo* el programa paso a paso, ya que muy posiblemente demoraríamos una eternidad en llegar al lugar de interés. El siguiente comando pone un punto de parada en la linea 6 de `debug.c`.

```
(gdb) break 6
Breakpoint 1 at 0x400502: file debug.c, line 6
```

Podemos poner todos los breakpoints que deseemos. En el caso anterior, ya que el programa consiste en un sólo archivo fuente, alcanza con especificar la línea de código en donde queremos parar. Si hay más de un archivo fuente involucrado, podemos especificarla de la siguiente manera. Ya que estamos, además, agregamos una *condición de parada*, es decir, el breakpoint tendrá efecto sólo si la condición especificada se cumple al pasar por esa linea:

```
(gdb) break debug.c:7 if a == 5
Breakpoint 1 at 0x400502: file debug.c, line 6
```

En el caso anterior, nos detendremos en el interior del `for` cuando el valor de la variable `i` alcance el valor `5`. Notar que esto puede pasar o no según el parámetro que se le pase a la función.

Ejecución incremental Una vez marcados los breakpoints estamos en condiciones de iniciar nuestra ejecución controlada. La depuración comienza con el comando `run`, que toma como argumentos opcionales los argumentos de línea de comandos que uno desee pasarle al programa. El comando y su salida se muestran a continuación:

```
(gdb)run
Starting program: /home/nacho/workspace/pie/teorico/debug

Breakpoint 1, sumatoria (a=10) at debug.c:6
6  for (i = 0; i < a; i++) {
(gdb)
```

El GDB nos muestra aquí que ha llegado al primer breakpoint que marcamos. Además de eso nos da algo de información básica sobre el estado actual del programa, como ser la línea de código correspondiente al breakpoint (que aún no se ha ejecutado!), el nombre de la función (sumatoria), y los valores de los parámetros con que fuera llamada la función al llegar al breakpoint en esta ocasión. Un breakpoint permanece activo hasta que uno lo desactive o se salga de GDB, y durante una misma sesión de depuración es posible pasar muchas veces por el mismo lugar.

En este punto, podemos por ejemplo ejecutar el programa paso a paso. Esto puede hacerse básicamente de dos maneras, con los comandos `step` (abreviado `s`) o `next` (abreviado `n`). El comando step, sin argumentos, ejecuta una línea del programa. Opcionalmente recibe un argumento con la cantidad de líneas que se desea avanzar. Avancemos pues a la primera linea del for

```
(gdb) step
7     s += i;
```

Análisis del estado Una de las cosas básicas que uno puede hacer es analizar el estado actual de las variables locales y los parámetros de la función actual. Para esto (y mucho más) existe el comando `print`. Veamos cómo usarla para imprimir unas cuantas cosas:

```
(gdb) print i
$2 = 0
(gdb) print a
$3 = 10
(gdb) print i+a
$4 = 10
(gdb) print i < a
$5 = 1
(gdb) print atoi("123")
$6 = 123
(gdb)
```

De hecho, con print podemos evaluar e imprimir el resultado de *cualquier expresión válida de C* con las funciones que estén disponibles al momento de la ejecución del programa siendo depurado. No tenemos por ejemplo las funciones matemáticas aquí, porque no incluímos la biblioteca matemática, pero sí tenemos todas las funciones de la biblioteca estándar, como `atoi`. Esto anterior es extremadamente útil para ensayar los cálculos que están siendo depurados con parámetros alternativos por ejemplo.

4 Análisis de la pila

Muy comunmente nos sucederá que nuestro programa termina inceremoniosamente con una violación de segmento o *segmentation fault*. Esto suele deberse a que, de alguna manera, intentamos acceder o escribir en lugares de memoria que no eran de nuestro programa. Una opción para detectar tales situaciones es hacer una depuración del programa paso a paso y tratar de ver a dónde podría estar ocurriendo esto (muy comunmente, esto ocurre al acceder arreglos o punteros). Otra forma es ejecutar el programa en cuestión dentro de GDB y luego analizar el estado del programa al producirse el error de ejecución. Veamos el siguiente caso:

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

int reventarum(int a) {
  int **pp;
  int *p;
  pp = malloc(sizeof(int*)*a);
  p = pp[0];
  p[0] = 5; // !!
  return p[1];
  free(p);
}

int explotarum(int a) {
  int b = 2*a;
  return reventarum(b)/2;
}

int destrozarum(int c, int d) {
  return explotarum((c + d)/2);
}

int main() {
  int x = 5;
  x -= destrozarum(5,3);
  x += explotarum(10);
  return x;
}
```

Si ejecutamos este programa nos dará una violación de segmento, ya que si bien inicializamos el arreglo de punteros a int `pp`, no inicializamos cada uno de sus punteros, por lo que el valor de `pp[0]` está indefinido y su acceso a él es inválido. Veamos entonces cómo nos ayuda GDB en este caso. Depuramos el programa con

```
$gdb ./debug2
```

Luego corremos el programa desde GDB. Esto es lo que sucede:

```
(gdb) run
Starting program: /home/nacho/workspace/pie/teorico/debug2

Program received signal SIGSEGV, Segmentation fault.
0x000000000040055c in reventarum (a=8) at debug2.c:9
9  p[0] = 5; // !!
(gdb)
```

En este caso es bastante claro qué es lo que está sucediendo, pero muchas veces ésto dependerá de, por ejemplo, qué parámetros se pasaron a la función en donde ocurrió el problema. Por eso, es fundamental disponer de la *traza de ejecución*, que nos muestra el estado del stack del sistema, con toda la lista de funciones que fueron invocadas para llegar al punto en el que ocurrió el problema. Podemos ver la traza de ejecución al momento de ocurrir el problema con el comando `backtrace` (o su abreviación `bt`), e incluso *volver* hacia atrás en el stack hacia las funciones precedentes mediante el comando `frame`. Veamos esto en acción:

```
(gdb) bt
#0  0x000000000040055c in reventarum (a=8) at debug2.c:9
#1  0x0000000000400588 in explotarum (a=4) at debug2.c:16
#2  0x00000000004005b9 in destrozarum (c=5, d=3) at debug2.c:20
#3  0x00000000004005d9 in main () at debug2.c:25
(gdb) list
  4int reventarum(int a) {
    5  int **pp;
    6  int *p;
    7  pp = malloc(sizeof(int*)*a);
    8  p = pp[0];
    9  p[0] = 5; // !!
    10  return p[1];
    11  free(p);
    12}
  13
(gdb) frame 1
#1  0x0000000000400588 in explotarum (a=4) at debug2.c:16
(gdb) frame 2
#2  0x00000000004005b9 in destrozarum (c=5, d=3) at debug2.c:20
  20  return explotarum((c + d)/2);
(gdb) frame 0
#0  0x000000000040055c in reventarum (a=8) at debug2.c:9
(gdb) print pp
$3 = (int **) 0x602010
(gdb) print p
$4 = (int *) 0x0
```

En las últimas dos líneas vemos por fin qué es lo que sucedió: el puntero `p` tiene valor `0x0` (0 en hexadecimal), lo cual es un valor inválido de puntero, y por lo tanto intentar acceder a una posición de memoria utilizándolo (en este caso `p[0]`) resulta en una violación de segmento.

5 Errores oscuros

El lenguaje C no tiene mecanismos para protegerse automáticamente contra accesos indebidos a memoria por parte del programa. Esto puede ocasionar errores fáciles de detectar, como el que mostramos

anteriormente, donde simplemente no se inicializó un puntero y su valor era obviamente inválido, pero también puede suceder que intentemos modificar lugares de memoria válidos de nuestro programa, pero que no son los que deberíamos estar accediendo. Más aún, es posible romper la estructura interna del propio programa (por ejemplo, el stack o el heap), ante lo cual los síntomas pueden ser muy extraños a primera vista. Puede suceder por ejemplo que se den errores de cálculo en funciones que están bien implementadas, y que esto dependa de si más adelante se invoca o no a otra función que no tiene nada que ver, si acaso esa función destruye las variables locales involucradas en ese cálculo durante su ejecución. También pueden darse violaciones de segmento u otro tipo de errores de ejecución serios que no tienen nada que ver con lo que aparentemente estamos queriendo hacer. En estos casos, no queda otra que hacer una ejecución paso a paso del programa y ver en qué punto se corrompe. Veamos un ejemplo diabólico:

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

int reventarum(int a) {
  int x[4]; // jo jo jo
  x[13] = 5; // acá está b (en linux 64 bits!)
}

int explotarum(int a) {
  int b;
  b = 2*a;
  reventarum(b);
  return b;
}

int machacarum(int k) {
  int *p;
  int i;
  if (k == 0) {
    p = &k;
    for (i = 0; i < 100; i++) {
      *p++ = i; // destroza el stack!
    }
  } else { machacarum(k-1); }
}

int destrozarum(int c, int d) {
  return explotarum((c + d)/2);
}

int main() {
  int x = 5;
  x -= destrozarum(5,3);
  machacarum(5);
  return x;
}
```

Veamos qué pasa al depurar este programa

```
(gdb) break 11
```

```
Breakpoint 1 at 0x400508: file debug3.c, line 11.
(gdb) break 13
Breakpoint 2 at 0x40051a: file debug3.c, line 13.
(gdb) run
Starting program: /home/nacho/workspace/pie/teorico/debug3

Breakpoint 1, explotarum (a=4) at debug3.c:11
11  b = 2*a;
(gdb) s
12  reventarum(b);
(gdb) print b
$1 = 8
(gdb) s
reventarum (a=8) at debug3.c:6
6  x[13] = 5; // acá está b (en linux 64 bits!)
(gdb) n
7}
(gdb) n

Breakpoint 2, explotarum (a=4) at debug3.c:13
13  return b;
(gdb) print b
$2 = 5
(gdb)
```

Como podemos ver, el valor de `b` luego de llamar a `reventarum` es... `5` en lugar de `8`. Lo que sucedió es que la posición de memoria del elemento (inválido) `x[13]` en `reventarum` se corresponde con `b` en el stack, y por lo tanto estamos de hecho cambiando su valor inadvertidamente. Más aún, el valor 13 es particular de GCC 4.6.4, Linux 64 bits. En otras plataformas, el daño será posiblemente otro.

Ya en la función `machacarum` la cosa es más seria. Ahí directamente estamos sobreescribiendo todo un bloque del stack con valores sin sentido. Veamos cómo está todo justo antes de la hecatombe:

```
(gdb) break 20
Breakpoint 3 at 0x400531: file debug3.c, line 20.
(gdb) continue
Continuing.

Breakpoint 3, machacarum (k=0) at debug3.c:20
20    p = &k;
(gdb) bt
#0  machacarum (k=0) at debug3.c:20
#1  0x0000000000400564 in machacarum (k=1) at debug3.c:24
#2  0x0000000000400564 in machacarum (k=2) at debug3.c:24
#3  0x0000000000400564 in machacarum (k=3) at debug3.c:24
#4  0x0000000000400564 in machacarum (k=4) at debug3.c:24
#5  0x0000000000400564 in machacarum (k=5) at debug3.c:24
#6  0x00000000004005b9 in main () at debug3.c:34
```

El backtrace refleja el hecho de que `machacarum` es llamada recursivamente. El apocalipsis sólo sobrecaerá sobre nosotros al llegar k a `0`, que es precisamente donde estamos ahora. Apretamos el botón rojo y voilá! Violación de segmento.