Bisoños Usuarios de Linux de Mallorca y Alrededores | Bergantells Usuaris de Linux de Mallorca i Afegitons

Introducción a GNU Debugger (GDB) **(3154 lectures)**

Per Carles Pina i Estany, cpina (http://pinux.info)
Creado el 29/06/2003 13:49 modificado el 29/06/2003 13:49

GDB es el debugger (depurador) de GNU. Es para línea de comandos, en modo texto, tiene soporte para varios lenguajes y es muy potente. No es difícil de usar para lo más básico y es lo que intentaremos ver aquí. ¿Cómo usarlo? ¿Cómo mandar reportes de bugs sabiendo donde falla? ¿Cómo analizar un fichero core?

Hay muchos depuradors en Hasefroch que son en modo gráfico. En Linux también hay varios en modo gráfico (o texto con menús) aunque la mayoría son frontends de GDB (Gnu DeBugger). Así que deberiamos tener una mínima idea de cómo funciona GDB para saber hasta donde podemos llegar con los otros, comprenderlos, etc.

Un debugger nos sirve para ver qué pasa dentro de un programa, ya sea en tiempo de ejecución o bien "post-mortem" (ficheros core), cuanto valen las variables, en qué función ha petado; poder parar si se cumple una condición, etc.

Evidentemente, tendremos una aplicación, por ejemplo `hola.c`. También tendremos el gdb instalado (en Debian, `apt-get install gdb` debería bastarnos). La compilaremos *con soporte para GDB* de esa forma:

```
carles@pinux:~$ gcc -ggdb hola.c -o hola
```

Con eso estamos incluyendo más información al ejecutable para que GDB pueda decirnos nombres de funciones, de variables, etc.

Podemos usar GDB de tres maneras:

- Arrancar el programa desde GDB.
- Atacharnos a un programa ya ejecutado.
- Analizar un fichero core.

Una vez tenemos la sesión del GDB, dentro de ciertas limitaciones (no podemos "seguir ejecutando" una sesión de un fichero core) tendremos las mismas opciones.

Arrancar el programa desde GDB

```
carles@pinux:~$ gdb -q hola
(gdb) run pepe
Starting program: /home/carles/hola pepe
argc: 2
argv1: pepe

Program exited normally.
(gdb)  quit
```

Notas:

- El parámetro `-q` hace que GDB no nos enseñe la licencia, etc.
- En la línea `(gdb)`, GDB está esperando una entrada nuestra. Le decimos que ejecute el programa y le pase de parámetro `pepe`.

- Las líneas argc, y argv1 son simples `printfs` del programa.
- Con `quit` salimos de GDB.

Atacharnos a un programa ya ejecutado

Suponemos que se está ejecutando en un terminal el programa `hola2` y queremos ver qué está haciendo, cuánto vale alguna variable, etc.
Desde otro terminal miraremos qué PID tiene:

```
carles@pinux:~$ ps auxw | grep -i hola2
carles   12196 17.2  0.1  1196  276 pts/10   T    13:09   0:16 ./hola2
carles   12212  0.0  0.2  1744  692 pts/7    R    13:10   0:00 grep -i hola2
carles@pinux:~$
```

Vemos que tiene el PID 12196. Entonces ejecutaremos `gdb -q hola2 12196`:

```
carles@pinux:~$ gdb -q hola2 12196
Attaching to program: /home/carles/hola2, process 12196
Reading symbols from /lib/libc.so.6...done.
Loaded symbols for /lib/libc.so.6
Reading symbols from /lib/ld-linux.so.2...done.
Loaded symbols for /lib/ld-linux.so.2
0x08048423 in main (argc=1, argv=0xbffffb64) at hola2.c:9
9               for (i=0;;i++) {}
(gdb)
```

Vemos que estaba en la función `main`, los valores que se le han pasado en la función donde estaba (argc=1, argv=0xbffffb64), el nombre del fichero (hola2.c) y la línea del fichero (línea 9). En este punto el programa queda pausado, esperando que nosotros le demos más instrucciones.

Esta forma de cogernos a un programa ya ejecutado es útil para depurar a programas con forks, etc... Sencillamente miramos el PID del proceso que queremos depurar y lo pasamos al gdb.

A partir de aquí podemos hacer las funciones que ya veremos (ver cuanto valen las variables, asignar valores, etc.)

Analizar un fichero core

Hemos ejecutado un programa y en un momento dado nos ha dicho:

```
Violación de segmento (core dumped)
```

Si no nos da el `core dumped` seguramente es que tenemos desactivado que nos vuelque los cores al disco. Para ver si lo tenemos activado hacemos:

```
carles@pinux:~$ ulimit -c
100000
carles@pinux:~$
```

Nos mostrará el tamaño máximo del fichero de core. Si es 0, no se hace fichero de core, así que le ponemos un tamaño máximo para nuestas necesidades.

Para ver qué pasó:

```
carles@pinux:~$ gdb -q mplayer core
Core was generated by `mplayer prova.asx'.
Program terminated with signal 11, Segmentation fault.
#0  0x6c6e6977 in ?? ()
(gdb)
```

Y a partir de aquí hacemos el análisis que veremos a continuación para ver la función (con backtrace), qué valian las variables, etc.
Cuando un programa es grande en memoria (muchas variables, estructuras de datos, etc.) su core correspondiente también será grande, ya que está volcando todo loque hay en memoria en un fichero.

Analizando el programa

Tanto si nos hemos atachado a un ejecutable o bien ejecutado desde el inicio en la línea de comandos podemos hacer lo típico de muchos depuradores: ejecutar una instrucción, inspeccionar variables, cambiar variables, llamar a otras funciones, poner puntos de "break", etc.

Si estamos analizando un fichero core podremos ver en qué función nos hizo el core, qué valian las variables, etc. pero no podemos ejecutar paso a paso.

Pasemos a ver un listado de varias funciones con ejemplo/idea de uso para cada una. Entre paréntesis hay la abreviatura para no escribirlo todo cada vez.

run

Con `run` ejecutamos el programa desde el inicio. Se irá ejecutando hasta encontrar algun breakpoint o watchpoint.
Podemos pasarle parámetros como si de la línea de comandos fuera: `run param1 param2`.

next (n)

Ejecuta la siguiente línea de programa. Si es una función, ejecuta la función entera (no entra en ella).

step (s)

Como next, pero si es una función entra en ella.

backtrace (bt)

Nos enseña la pila del programa. De esa forma veremos fácilmente en qué funcionas ha entrado, con qué valores, etc. Muy útil para ver donde falló algo cuando tenemos el fichero de core.

Ir con cuidado, porqué si nos peta, p. ej. en la función printf (porqué falta el \0), como no tenemos la glibc compilada con -ggdb no lo veremos muy claro. Lo mismo si tenemos problemas en funciones implementadas por GTK o otras librerías.

break (b)

Una de las funciones más usadas. Un break sirve para ir ejecutando de forma normal el programa hasta que llega a cierta línea. Podemos usarlo de las siguientes maneras:

- `break [línea]`. Pone un break en la línea que sea del fichero actual.
- `break [función]`. Hace un break cuando empieza la función que sea.
- `break [fichero]:[línea]`. En el fichero (típicamente un .c) indicado en la línea indicada, hace un break.
- `break [fichero]:[función]`. Break en la función del fichero...

Una vez hecho un break podemos hacer next, step, print, continue, etc.

until

Irá ejecutando el código hasta llegar a la siguiente línea donde nos encontremos. Muy útil para situarnos a la última línea de un while o for, y hacer until que nos dejará con todo el while/for ejecutado.

watchpoints

Con los watchpoints le decimos al gdb "parate cuando cambie esa variable" (o bien cuando leemos de esa variable). Podemos usarlo de esas maneras:

- `watch expr`. Cuando expr cambia, GDB hace como un "breakpoint". Hay veces que puede cambiar una expresión y no ser evidente (por ejemplo nos pasamos de índice en un array)
- `rwatch expr`. Cuando se lee la expresión.
- `awatch expr`. Cuando la expresión es leída o escrita
- `info watchpoints`. Igual que `info break` pero con los watchpoints.

Está muy bien explicado en el manual[(1)] del GDB.

clear

Usamos clear para eliminar breakpoints (o watchpoints). P. ej. `clear 10` elimina el breakpoint de la línia 10. También para eliminar watchpoints.

info break/watch (i)

Nos hará un listado de los breakpoints/watchpoints definidos.
usando `info` vemos varias posibilidades del comando.

continue (c)

Estamos en una sesión de GDB y queremos ejecutar hasta el siguiente breakpoint, hacemos continue y sigue hasta él. Quizás estamos aquí por un breakpoint o bien porqué hemos empezado con next y step.

list (l)

Podemos usarlo para ver el código fuente:

- `list`: nos enseña las 10 próximas líneas.
- `list -`: las 10 líneas anteriores
- `list [numlin]`: las líneas alrededor de esa. Podemos pasar un fichero también con `fichero.c:num_lin`
- `list i,j`: nos enseña de la línea i a la j.

print (p)

Con esa función podemos imprimir el contenido de las variables. P. ej.:

```
(gdb) print i
$1 = 3
(gdb) print i+2
$2 = 5
(gdb)
```

(podemos ver que se pueden realizar operaciones). Si es un puntero, etc. funciona igual que en C:

```
(gdb) print argv
$8 = (char **) 0xbffffb64
(gdb) print argv[1]
$9 = 0xbffffc5b "bulma"
(gdb) print argv[1][0]
$10 = 98 'b'
(gdb) print &i
$11 = (int *) 0xbffffb04
(gdb)
```

Si es un array lo puede mostrar todo en pantalla de golpe:

```
(gdb) print a
$12 = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
(gdb) print a[3]
$13 = 3
```

```
(gdb)
```

whatis

Nos enseña el tipo de variable que es:

```
(gdb) whatis i
type = int
(gdb)
```

set var

Mediante `set var` cambiaremos el valor de una variable. Por ejemplo:

```
(gdb) set var i=3
(gdb) p i
$19 = 3
```

jump

Salta a una línea en concreto o bien a una función (`jump [línea]`, `jump función`)

call

Se usa `call funcion(param1,param2)`. Llama a la función con los parámetros que sea.

return

Termina la función donde estamos y devuelve el valor que le pasamos: `return 3`

help

Haciendo `help comando` veremos la ayuda para un comando en concreto.

quit (q)

Sale de `gdb`

Finalmente...

gdb es MUY potente. Eso descrito aquí es una breve descripción de la potencia de verdad, pero se debería poder usar GDB con las opciones mínimas vistas aquí.

Se puede mejorar para depurar threads, multiproceso y mucho más.

Hay varios frontends para GDB, o IDEs con frontends. Por ejemplo, el xxgdb[2], ddd[3], kdbg[4], etc[5]. (todos ellos en Debian)

Más documentación

Podeis encontrar más documentación en:

- /usr/share/doc/gdb (sobretodo el fichero refcard.ps, chuleta muy buena)
- En su guía de usuario[6]
- man gdb
- Su web[7]

Lista de enlaces de este artículo:

1. http://sources.redhat.com/gdb/onlinedocs/gdb_6.html#SEC30
2. http://www.cs.uni.edu/Help/xxgdb.html
3. http://www.gnu.org/software/ddd/
4. http://members.nextra.at/johsixt//kdbg.html
5. http://www.gnu.org/software/ddd/#References
6. http://sources.redhat.com/gdb/onlinedocs/gdb_toc.html
7. http://sources.redhat.com/gdb/

E-mail del autor: carles _ARROBA_ pinux.info
Podrás encontrar este artículo e información adicional en:
http://bulma.net/body.phtml?nIdNoticia=1805