

DEPURADOR GDB

Introducción

GDB es el source debugger de GNU. Es un poderoso debugger que permite "ver" que esta sucediendo dentro de programas escritos en C, C++ y Modula-2. Entre las capacidades más notorias que este debugger posee están:

- Debugging de programas complejos con múltiples archivos.
- Capacidad para detener el programa o ejecutar un comando en un punto específico (breakpoints), según una condición (watchpoints) o al llegar un signal (catchpoints).
- Capacidad para mostrar valores de expresiones cuando el programa se detiene automáticamente (displays).
- Es posible examinar la memoria y/o variables de diversas formas y tipos, incluyendo estructuras, arreglos y objetos.
- Es posible igualmente cambiar los valores de las variables para estudiar el comportamiento del programa sin necesidad de recompilar.
- Posibilidad de realizar debugging a programas en ejecución (procesos).
- Posibilidad de realizar debugging a programas que han finalizado.
- Múltiples formas de entrar al debugger.

Invocación del Debugger

El debugger se puede ejecutar de una de las siguientes formas:

```
$ gdb
```

para entrar al modo interactivo.

```
$ gdb programa
```

para cargar el programa y entrar en el modo interactivo. El programa no comienza hasta que sea indicado con un comando.

```
$ gdb programa core
```

para realizar debugging de un programa que ha finalizado con un 'core'. El debugger carga el programa y su ambiente exactamente como terminó. Útil para verificar por que un programa termino mal o para ver donde un programa se colgó (usando CTRL-\ para cortar el programa y obtener un core).

```
$ gdb programa pid
```

para depurar un programa en ejecución con el pid indicado. El proceso se detiene y el debugger lo controla en otro terminal. Sumamente útil para depurar programas con interfaz desde otro terminal virtual. Una vez que se entra al modo interactivo, GDB acepta comandos hasta que se le indique que se desea salir con el comando 'quit'. La presión de ENTER solo en una línea siempre invoca el último comando introducido.

Comandos más frecuentemente usados:

Los comandos más frecuentemente usados son:

```
list [archivo:]funcion
list [archivo:]línea[,línea]
list
list -
```

para listar el fuente a partir de una función o una línea. 'list' solo continua el listado previo (page down). 'list -' lista las líneas anteriores (page up).

```
break [archivo:]función
break [archivo:]línea
```

para colocar un breakpoint al comienzo de la función o al comienzo de la línea indicada.

```
run [argumentos]
```

para comenzar la ejecución del programa desde el principio. Los argumentos son los pasados al ejecutable.

```
bt (backtrace)
```

para mostrar el stack del programa, indicando las funciones invocadas y en que lugares fueron llamadas.

```
print expr
```

para mostrar el valor de una expresión.

```
c
```

para continuar la ejecución del programa después de que ha sido detenido con un signal o un breakpoint.

```
next
```

ejecuta la próxima línea del programa sin entrar dentro de las funciones. Se puede aprovechar el hecho que GDB repite el último comando con ENTER para ejecutar varias líneas seguidas.

```
step
```

ejecuta la próxima línea del programa entrando en funciones. Se puede aprovechar el hecho que GDB repite el último comando con ENTER para ejecutar varias líneas seguidas.

```
jump línea
```

salta los comandos siguientes y comienza la ejecución a partir de 'línea'. Útil para continuar un programa que ha terminado con 'core', arreglando la situación y saltando las líneas defectuosas.

```
help [item]
```

ayuda en línea.

```
quit
```

sale de GDB.

Guía de uso rápida

Con los comandos mencionados anteriormente es posible hacer debugging de programas fácilmente y con buen control. Esta sección pretende mostrar brevemente como comenzar a usar gdb con solo esos comandos. Para poder usar un debugger en UNIX es necesario compilar el programa con la opción -g. Ejemplo:

```
cc -g prueba.c -o prueba
```

1. Ejecutar el programa desde el debugger, método recomendado en las primeras fases de prueba. Se obtiene control completo del programa y si 'este falla, se visualizara inmediatamente donde ocurrió la falla. Es posible cortar el programa en cualquier momento con CTRL+C y regresar al debugger, lo que permite verificar loops infinitos, etc.
2. \$ gdb prueba
3. (gdb) run [argumentos]
4. ...
5. CTRL+C
6. (gdb)

Una vez detectado funciones donde pueden haber problemas:

```
$ gdb prueba
(gdb) list función
...
(gdb) break línea
(gdb) run [argumentos]
...
break
(gdb) print expr
(gdb) next
...
(gdb) c
...
```

7. Determinar donde un programa termina con 'core'
8. \$ gdb programa core
9. #0 main () at prueba.c:100
10. 100 *(char *)0 = 10;
11. (gdb) bt
12. ...

Una vez dentro de gdb, se pueden examinar otras variables y hacer un backtrace para verificar el camino que tomo el programa para producir la excepción. Es posible arreglar la situación y ejecutar un 'jump' para continuar ignorando el error (ver comandos avanzados):

```
(gdb) arreglar la situación
(gdb) jump línea                ejecutar ignorando el
error
```

13. Programa Ejecutando Es posible depurar un programa en ejecución. Para esto hacer:
 14. \$ ps para determinar el pid
 15. \$ gdb programa pid para interceptar el programa
 16. (gdb) en este momento el programa se
detiene
 17. ...

Uso de breakpoints, watchpoints y catchpoints.

Breakpoints son puntos donde el programa se detiene cuando pasa por ellos.

Watchpoints son expresiones que detienen el programa cuando el valor cambia.

Catchpoints son breakpoints sobre signals. Los comandos son como siguen:

```
break [archivo:]función
break [archivo:]línea
```

para colocar un breakpoint al comienzo de la función o al comienzo de la línea indicada.

```
tbreak [archivo:]línea
tbreak [archivo:]función
```

igual que break, pero el breakpoint es valido por una sola vez. Útil para crear breakpoints temporales.

```
watch expr
```

se habilita un watchpoint cuando la expresión <expr> cambia.

```
catch
```

se colocan breakpoints en todos los handlers de excepciones del contexto actual.

```
info break
info watch
```

muestra los watchpoints o breakpoints habilitados.

```
info catch
```

indica si se están interceptando las excepciones.

```
clear línea
clear función
```

para eliminar un breakpoint de la línea indicada. Ver delete para eliminar breakpoints por numero.

```
delete numero
```

para eliminar un breakpoint por numero. El numero puede verse con info break.

```
disable breakpoint
enable breakpoint
```

para habilitar o deshabilitar temporalmente un breakpoint. A diferencia de delete no se pierde la referencia de la línea donde se encuentra, simplemente es ignorado.

```
condition breakpoint [expr]
```

para hacer que un breakpoint sea condicional. Es decir, solo se habilita si la expresión <expr> es cierta. Como <breakpoint> se debe pasar el numero del breakpoint. Si <expr> no se especifica se hace el breakpoint incondicional.

```
ignore breakpoint [count]
```

ignora <count> pasadas sobre el breakpoint <breakpoint>.

Examinando y cambiando los datos.

GDB ofrece comandos para manipular los datos. Entre estos están:

```
whatis variable
```

indica de que tipo es la variable.

```
p tipo tipo
```

imprime la definición del tipo indicado.

```
print [/fmt] expr
```

imprime la expresión. /fmt es un indicador de formato (ver x). Para print el formato solo puede estar compuesto por una letra de cambio de tipo.

```
set variable=expresión
```

cambia el valor de una variable al resultado de la expresión.

```
display [/fmt] expr
```

habilita un display continuo durante debugging de la expresión indicada. Cada vez que el programa se detiene se hace un print [/fmt] de la expr. Se pueden tener varios displays a la vez. Con delete se pueden eliminar displays y con enable y disable se pueden habilitar y deshabilitar igual que como si fueran breakpoints.

```
undisplay numero.
```

equivalente a delete de un display: destruye un display.

```
x [/fmt] address
```

examinar memoria. /fmt es un indicador opcional formado por '/', seguido de un número (contador), seguido de una letra de formato, seguido de una letra de tamaño. Las letras de formato son:

o	octal	f	float
x	hexadecimal	a	address
d	decimal	u	unsigned decimal
t	binary	s	string
c	char		

las letras de tamaño son:

b	byte	h	halfword
w	word	g	giant (8 bytes)

El contador indica cuantos elementos imprimir, así:

```
x /10xb arreglo
```

imprime los 10 siguientes bytes del arreglo en hexadecimal.

Manipulando el Stack.

Los comandos para manipular el stack permiten cambiar y/o examinar variables que se encuentran en otros contextos al local. Es posible verificar el contenido de todas las variables automáticas en el stack. Los comandos son:

```
frame [N]
```

selecciona el frame N y lo imprime. Sin argumento el comando indica donde se está actualmente (i.e. frame actual).

```
bt [N]
```

para ver el contenido del stack. Si se especifica un numero positivo se ven los N primeras entradas en el stack y un numero negativo se ven las ultimas N.

```
select-frame #
```

se selecciona el frame number indicado (el numero lo da el comando bt). El cambiar de frame permite ver o cambiar las variables sobre el stack. No desapila ni empila ningún nuevo frame.

```
up
```

para ir al frame inmediatamente superior (i.e. la rutina que llamo a la actual).

```
down
```

para ir al frame inmediatamente inferior.

```
return
```

para forzar el desapilamiento del frame actual.

Comandos para impresión de información de Estado

Los siguientes comandos imprimen información variada de estado del debugger y del programa depurado:

```
info files
```

muestra los archivos y procesos que se están depurando.

```
info program
```

estado del programa al momento.

```
info sources
```

muestra los archivos fuentes en debugging.

```
info types
```

muestra todos los tipos definidos.

```
info variables
```

muestra todas las variables globales definidas.

```
info functions
```

muestra todas las funciones definidas

```
info display
```

muestra todas las expresiones display en efecto.

Profesor: Javier Zofío

```
info breakpoints
```

muestra todos los breakpoints en efecto.

```
info watchpoints
```

muestra todos los watchpoints en efecto.

```
info args
```

muestra los argumentos del frame actual.

```
info locals
```

muestra las variables locales del frame actual.

Otros comandos útiles.

Otros comandos de GDB que son útiles a la hora de depurar:

```
file
```

para cargar un nuevo ejecutable y tabla de símbolos dentro del debugger.

```
cd
```

para cambiar de directorio.

```
pwd
```

para ver el directorio actual.

```
shell
```

para salir a un subshell.

```
search reg-expr
```

para buscar en el fuente por una expresión regular a partir de la línea actual hacia abajo.

```
reverse-search reg-expr
```

para buscar en el fuente por una expresión regular a partir de la línea actual hacia arriba.

```
make
```

para correr el programa make.

Conectando gdb a un proceso en ejecución (agregado por Carlos Figueira)

1. Identificar el PID del proceso al que se desea conectar
2. Lanzar gdb con el nombre del ejecutable de ese proceso (para tener la información de los símbolos)
3. En gdb, ejecutar el comando "attach proceso", donde "proceso" es el PID del proceso

Una vez completados estos pasos, gdb está en control de la aplicación y puede utilizarse como si se hubiera arrancado desde gdb. Una variante es detener el proceso primero (con Ctrl-Z), y luego de que gdb asuma el control de la aplicación, ejecutar el comando "fg" para reactivarlo de nuevo.