Aprendendo a utilizar o GNU Debugger (parte 1)

Autor: Ricardo Rodrigues Lucca <rlucca at gmail.com>
Data: 26/01/2004

Introdução

Em programação, muitas vezes nos deparamos com becos sem saída por causa de erros que não temos nem idéia de onde possam estar.

Assim sendo, um depurador (vulgo: debugador) pode ser o nosso melhor amigo. Amigo que quando sabemos como tratá-lo, nos ajudará horrores! :).

O depurador de que vou falar é o *GNU Debugger* ou simplesmente, *gdb*. Nessa série de artigos irei mostrar como podemos utilizar o gdb para problemas comuns.

[]'s e boa leitura!

Iniciando

Você pode chamar o *GNU Debugger* digitando no console o comando "**gdb**" seguido de um enter (o enter é muito importante :p).

Podemos chamá-lo também passando como parâmetro o executável do programa:

$ ls
ex ex.c

$ gdb ex
GNU gdb 5.2
Copyright 2002 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB. Type "show warranty" for details.
This GDB was configured as "i386-slackware-linux"...
(gdb) _

Assim o *gdb* já irá carregar o nosso programa. Também é permitido fazermos coisas do tipo:

$ gdb ex 1305
ou
$ gdb ex core

Respectivamente, o primeiro esta dizendo para o gdb usar o processo já existente para o programa e não criar outro. Agora, o segundo diz que iremos analisar aqueles "malditos" arquivos (core) que aparecem do nada no nosso HD. Os arquivos core contém o estado de um processo no momento de uma parada ocasionada por algum erro.

Encerrando e pedindo ajuda no gdb

Uma vez iniciado o *gdb*, o prompt muda e temos três formas de sair:

- Digitando "quit";
- Digitando "q";
- Pressionando CONTROL+D.

Você deve ter percebido que a primeira e a segunda opção são parecidas. É isso mesmo! Na verdade elas são a mesma opção, pois o gdb nos permite fazer abreviações dos comandos, desde que isso não gere ambiguidade com outro comando!

Para terminar essa página gostaria de dizer que podemos conseguir ajuda do programa em tempo de execução digitando **help**. Sendo que o help ainda aceita como parâmetro "algo" que queiramos saber mais a respeito.

gdb e gcc

Para utilizarmos o *gdb* devemos utilizar o parâmetro "-g" no *gcc*. Isso informa ao gcc que iremos utilizar futuramente um depurador e assim ele irá inserir no arquivo alguns dados que o gdb poderá utilizar.

Na verdade, sem isso não temos acesso as funções do gdb, pois ele apresentará erros. Talvez, sendo a única função que possa ser utilizada sem o gdb seja a para execução do programa.

Carregamento e execução

Se você não gosta de passar o arquivo que o *gdb* tratará pela linha de comando, pode fazer isso através do comando "**file**" seguido de um espaço e do nome do arquivo (não esquecer o enter :p).

(gdb) **file ex**
Reading symbols from ex...done.

Percebam que quando estamos no gdb o prompt se torna "(gdb)". Aqui o programa não contem erros, mas foi compilado com a opção "-g" no gcc. Assim, ele conseguiu ler corretamente o arquivo.

O comando para executar o programa é o "**run**", onde qualquer parâmetro que seja dado para ele será considerado do programa que será executado.

(gdb) **run**
Starting program: /home/rlucca/C/seila/ex
01! 02! 03! 04! 05!

Program exited normally.
(gdb) **file /bin/ls**
Load new symbol table from "/bin/ls"? (y or n) **y**
Reading symbols from /bin/ls...
(no debugging symbols found)...done.

Bem, quem for analisar o exemplo acima pelo já comentado verá que executamos um programa chamado "ex", que foi carregado anteriormente com "file". Assim, a saída "01! 02! 03! 04! 05!" é do programa que tem a única função de ser exemplo para a execução daqui e conta de 1 até 5 exibindo sempre 2 casas.

Mas e depois? Temos mais 1 exemplo do uso de "file" para carregar programas. Quando já temos um arquivo carregado o gdb nos pergunta se queremos realmente reler os "dados" novamente. Respondendo "yes" (sim, em Português), ele tentará ler os dados. Mas, o que é isso? Nenhum símbolo foi encontrado pro depurador poder usar. Sim, é isso mesmo! Isso é um exemplo do que ocorre quando tentamos carregar um executável compilado sem informações pro depurador.

Um exemplo simples

Bom, no exemplo que vou utilizar irei ensinar como, no console, podemos gerar um arquivo "core" e analisá-lo. Vamos pegar um exemplo:

```
#include <stdio.h>

int main(void)
{
	int a=1;

	while (a<5) {
		printf("%2d! ",a);
		a++;
		sleep(1);
	}
```

```
    return 0;
}
```

Muito bem, copie o exemplo e salve como "ex.c". Compile e execute. Ele não contém erros. Agora, execute ele e durante a execução pressione as teclas CONTROL+Pipe. O Pipe (|) é a barrinha reta usada normalmente para indicar que a saída de um programa vai ser direcionado para a entrada de outro.

Logo após isso será gerado um "core" e o programa será encerrado com a mensagem: "Quit (core dumped)" ou ainda somente "Quit" para quem tem o "ulimit -c 0" configurado.

Agora, abra o gdb e vamos para a próxima página!

List, BreakPoints e Next

Agora que temos o *gdb* rodando, carregue o exemplo digitando: **file ex**.

Bom, com um exemplo rodando vamos exemplificar um pouco esses comandos:

- **List** é usado para vermos uma porção do código do programa. Normalmente podemos chamar "list" (ou "l") seguido de um número que corresponderia a linha do código fonte. Também podemos passar como argumento o nome de uma função ou o endereço de memória.

```
(gdb) list
1       #include <stdio.h>
2
3       int main(void)
4       {
5               int a=1;
6
7
8               while (a<5) {
9                       printf("%2d! ",a);
10                       a++;
```

Bem, como já aprendemos a como executar o programa, vamos exemplificar o uso de "next" depois de breakpoint. O que deixará mais claro.

```
(gdb) b 7
Breakpoint 1 at 0x8048373: file ex.c, line 7.
```

Quando digitamos "breakpoint" (ou "b") seguido de um número, o gdb entende que este número é o numero da linha onde devemos "pausar" o programa. Assim, breakpoint é nada mais que uma parada

da execução num determinado ponto.

Geralmente o breakpoint é usado quando sabemos que o erro ocorre na linha tal, aí usamos um breakpoint na linha tal-1, por exemplo.

Agora, se digitarmos "run" o programa irá parar na próxima linha com algo a fazer, no caso, a linha 8. Onde podemos dar um "print" na variável "a" e ver quanto ela está valendo.

```
(gdb) run
Starting program: /home/rlucca/C/seila/ex

Breakpoint 1, main () at ex.c:8
8               while (a<5) {
(gdb) print a
$1 = 1
```

Digitando "next" (ou "n") iremos pedir pro gdb ir para a próxima instrução (linha) do programa.

```
(gdb) n
9                       printf("%2d! ",a);
(gdb) n
10                      a++;
(gdb) n
11                      sleep(1);
(gdb) print a
$2 = 2
```

Apreendendo a utilizar o GNU Debugger (parte 2)

Autor: Ricardo Rodrigues Lucca <rlucca at gmail.com>
Data: 24/09/2004

Introdução

Está é a segunda parte do artigo Apreendendo a utilizar o GNU Debugger, onde foi falado como fazer para depurar um programa utilizando o GDB (Gnu DeBugger). Vamos a um rápido resumo do que foi tratado no artigo anterior:

- Iniciar a depuração de três formas diferentes;
- Executar o programa dentro do gdb;
- Listar parte do código;
- Conceito e como utilizar Breakpoint;
- Fazer execução linha-a-linha(step-by-step) do código;
- Encerrando o gdb e obtendo ajuda!
- Compilar o programa corretamente! Parâmetro "-g" do gcc.
- Gerar arquivos core apartir de um programa em execução.

Assim, se houver qualquer dúvida referente ao mencionado acima, favor consultar a primeira parte sobre o artigo. Se mesmo assim, tiver dúvidas podem mandar email. :)

Agora, vamos iniciar este aqui :)

Fonte

Estarei utilizando um código-fonte diferente do que utilizei no primeiro artigo. Esse programa contém uma função que será muito útil para exemplos futuros nesse artigo.

Assim, eis o fonte:

```
#include <stdio.h>

void bla( void )
{
	int x;
```

```
	scanf(" %d", &x);
	x += 50;
	printf("X = %d\n", x);
}

int main( void )
{
	bla();
}
```

Acho muito importante separar o código-fonte usado do artigo em si para não gerar confusão. Assim, se sente-se seguro em utilizar outro durante a leitura, porque não? Sinta-se a vontade!

Compile-o e prepare o *gdb* :)

$ gcc teste.c -g
$ gdb a.out
GNU gdb 6.1.1
Copyright 2004 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB. Type "show warranty" for details.
This GDB was configured as "i486-slackware-linux"...Using host libthread_db library "/lib/libthread_db.so.1".

(gdb)

Continue e finish

Vamos iniciar definindo dois breakpoints. Um na linha 7 e outro na 9:

(gdb) **b 7**
Breakpoint 1 at 0x80483ca: file teste.c, line 7.
(gdb) **b 9**
Breakpoint 2 at 0x80483e4: file teste.c, line 9.

No artigo anterior terminamos falando do comando "next" e esquecemos de falar do "continue", que faz com que o programa retome seu curso normal de execução. Assim, se encontrar algum breakpoint ele irá parar normalmente.

(gdb) **run**
Starting program: /home/rlucca/C/t/a.out

Breakpoint 1, bla () at teste.c:7

```
7               scanf(" %d", &x);
(gdb) c
Continuing.
1

Breakpoint 2, bla () at teste.c:9
9               printf("X = %d\n", x);
(gdb) c
Continuing.
X = 51

Program exited with code 07.
```

Outro comando muito utilizado é o "finish". Ele retoma o curso normal de execução até o fim da função. Vejamos o que acontece!

```
(gdb) run
Starting program: /home/rlucca/C/t/a.out

Breakpoint 1, bla () at teste.c:7
7               scanf(" %d", &x);
(gdb) c
Continuing.
2

Breakpoint 2, bla () at teste.c:9
9               printf("X = %d\n", x);
(gdb) finish
Run till exit from #0  bla () at teste.c:9
X = 52
main () at teste.c:15
15      }
(gdb)
```

Isso ocorre porque na nossa linha 15 está localizado a chave que encerra a nossa função. Útil não? :D

Mais breakpoints

Agora nós vamos conhecer um pouquinho mais sobre os *breakpoints*. Primeiro de tudo, vamos aprender como vimos os breakpoints definidos atualmente. A exibição desse tipo de informação é feita através do comando "*info*" que tem como objetivo mostrar informações genéricas. Se quiser conhecê-lo a fundo utilize a ajuda online do gdb. Agora, como visualizamos os breakpoints atuais? Simples!

(gdb) **info breakpoints**

```
Num Type           Disp Enb Address    What
1   breakpoint     keep y   0x080483ca in bla at teste.c:7
        breakpoint already hit 1 time
2   breakpoint     keep y   0x080483e4 in bla at teste.c:9
        breakpoint already hit 1 time
```

Do mesmo modo, podemos abreviar o comando sem problemas!

```
(gdb) i b
Num Type           Disp Enb Address    What
1   breakpoint     keep y   0x080483ca in bla at teste.c:7
        breakpoint already hit 1 time
2   breakpoint     keep y   0x080483e4 in bla at teste.c:9
        breakpoint already hit 1 time
```

Agora, como faríamos para apagar um desses breakpoints? Para isso utilizamos o comando "delete" seguido do numero do breakpoint (que obtemos via "info b"). Assim, vamos remover o breakpoint da linha 9.

(gdb) **del 2**

Novamente estou mostrando como o *gdb* é poderoso, permitindo que se não acha outro comando com nome parecido possa haver a abreviação do nome. Assim, ao invés de escrever "delete" podemos escrever "del" ou até mesmo "d".

Vejamos um exemplo para remover todos os breakpoints de uma só vez.

```
(gdb) d
Delete all breakpoints? (y or n) y
(gdb) info breakpoints
No breakpoints or watchpoints.
```

No artigo anterior você aprendeu a utilizar o comando "break" com o número da linha informado com o comando "list". Mas fora essa opção também podemos utilizá-lo com o nome de uma função. Assim:

```
(gdb) b bla
Breakpoint 3 at 0x80483ca: file teste.c, line 7.
```

Essa linha "Breakpoint 3 at 0x80483ca: file teste.c, line 7." nos informa que o breakpoint foi bem sucedido. Mas não só isso. Nela é dita que foi criado o "breakpoint" de número 3, na linha 7 do arquivo "teste.c". Muito útil, não acham???

(un)display

O comando "Display", muito útil, realiza a mostra de dados automática quando um ponto de parada é atingido. Isso evita termos de ficar utilizando o famoso e trabalhoso "print"(ou "p").

Da mesma forma que os "breakpoints", podemos ver os displays já definidos com a ajuda do comando "info". Vejamos um exemplo:

(gdb) **info display**
There are no auto-display expressions now.

Ops! Não temos nenhum display definido, pois então vamos definí-lo. O "display" tem funcionamento muito parecido aos "breakpoints". Ele pode receber tanto o nome de uma função, como de variáveis. Como também endereços de variáveis, mas essa não utilizo e nem ao menos cheguei a testar.

(gdb) **display bla**
1: bla = {void (void)} 0x80483c4 <bla>
(gdb) **display x**
No symbol "x" in current context.
(gdb) **display bla.x**
2: bla.x = Attempt to extract a component of a value that is not a structure.
Disabling display 2 to avoid infinite recursion.

No exemplo acima definimos um único display para nos mostrar o que "bla" guarda. Não podemos definir um display para o "x" porque ele só existe localmente. Aqui demonstramos a tentativa de um "espertinho" que resulta em falha, pois "bla" não é uma estrutura. Logo, o "display" 2 é criado e automaticamente desabilitado.

Vamos rodar isso!

(gdb) **run**
The program being debugged has been started already.
Start it from the beginning? (y or n) **y**
Starting program: /home/rlucca/C/t/a.out

Breakpoint 3, bla () at teste.c:7
7 scanf(" %d", &x);
1: bla = {void (void)} 0x80483c4 <bla>

(gdb)

Agora, vamos definir o "display x" e fazer alguns "nexts" e depois utilizar "finish".

(gdb) **display x**
4: x = -1073743948
(gdb) **n**
3
8 x += 50;
4: x = 3
1: bla = {void (void)} 0x80483c4 <bla>
(gdb) **n**

```
9               printf("X = %d\n", x);
4: x = 53
1: bla = {void (void)} 0x80483c4 <bla>
(gdb) finish
Run till exit from #0  bla () at teste.c:9
X = 53
main () at teste.c:15
15      }
1: bla = {void (void)} 0x80483c4 <bla>
(gdb)
```

Como deveríamos saber quando fazemos "finish", o próximo ponto de parada já é considerado fora da função (mesmo estando na ultima linha dela "}"). E por isso o nosso "display x" não aparece. Vamos ver nossos "displays" atuais.

(gdb) **i display**
Auto-display expressions now in effect:
Num Enb Expression
4: y x (cannot be evaluated in the current context)
3: n bla->x
2: n bla.x
1: y bla

Aqui podemos ver algumas coisas interessantes, o "1" é um display para a função bla e esta ativo("y"). Já o "4", também esta ativo. Mas, fora do contexto(ele é uma variável local e estamos fora da função).

Como expliquei anteriormente, o "display" funciona muito parecido com o "breakpoint". Assim, para remover um "display" criado anteriormente passamos o número dele para uma função que é chamada "undisplay". Assim:

(gdb) **undisplay 4**
(gdb) **undisplay 1**
(gdb) **i display**
Auto-display expressions now in effect:
Num Enb Expression
3: n bla->x
2: n bla.x

E para remover todos os "displays" de uma única vez basta utilizar o comando "undisplay" sem parâmetros e confirmar!

(gdb) **undisplay**
Delete all auto-display expressions? (y or n) **y**
(gdb) **i display**
There are no auto-display expressions now.

Backtrace, indo e vindo de funções

Bom, estamos chegando já no final do nosso conteúdo. Mas vamos dar uma olhada se temos algum "breakpoint" esquecido já criado.

(gdb) **i b**
Num Type Disp Enb Address What
3 breakpoint keep y 0x080483ca in bla at teste.c:7
breakpoint already hit 1 time

Opa! Temos um na linha 7, perfeito! Vamos deixá-lo aí para nossa explicação... reinicie a execução do programa com o comando "run" e confirme.

(gdb) **run**
The program being debugged has been started already.
Start it from the beginning? (y or n) **y**
Starting program: /home/rlucca/C/t/a.out

Breakpoint 3, bla () at teste.c:7
7 scanf(" %d", &x);

Ótimo, antes de iniciar a expôr o "backtrace", vamos falar do seu conceito. Conforme seu código vai chamando funções em funções, estas são postas em algum lugar que informa que a função tal chamou a X na linha Z. Do mesmo modo, se função X chamar a função Y na linha S esses dados também terão que ser guardados de um jeito que a maquina possa resgata-los sempre na ordem correta. Para isso, é utilizado uma pilha para guardar essas informações (origem, linha e destino digamos) e essa pilha é chamada de "backtrace"! Vejamos o exemplo:

(gdb) **backtrace**
#0 bla () at teste.c:7
#1 0x0804840e in main () at teste.c:14

Como havia explicado, o "backtrace" funciona como uma pilha. O conceito de funcionamento de uma pilha é "O ultimo que entra é o primeiro a sai". Por isto que a nossa "main()" é a última da lista, pois foi a primeira a ser chamada e, assim, deve ser sempre.

Do mesmo modo, que podemos abreviar comandos o "backtrace" pode ser abreviado para "bt". O "backtrace" também é conhecido como o comando "where" (abreviação: "whe"), pois informa a função que estamos atualmente como "#0".

Agora que já sabemos onde estamos no nosso programa podemos utilizar as funções "up" e "down" para ir e vir pelo "backtrace".

Vejamos um exemplo:

(gdb) **p x**

```
(gdb) p x
$1 = -1073743948
(gdb) up
#1  0x0804840e in main () at teste.c:14
14              bla();
(gdb) p x
No symbol "x" in current context.
(gdb) down
#0  bla () at teste.c:7
7               scanf(" %d", &x);
(gdb) p x
$2 = -1073743948
(gdb)
```

Antes do "up" estamos na função "bla" e, assim, conseguimos ter acesso a variável local "x" para imprimí-la. Mas quando realizamos o "up" passamos para a função "main" onde não existe "x". Por fim, voltamos para a função "bla" utilizando o comando "down" onde conseguimos imprimir o valor de "x".

Isso pode ser muito útil, pois em programas uma função chama outra com parâmetros específicos. Não pensando no pior, podemos voltar dá função com possível problema e ver se os parâmetros passados para ela estão corretos. Se estiverem começamos a depurar a função com problema realmente. :)

Conclusão

Bom, vou concluir o artigo falando de um comando que exibe uma janela em modo texto tornando permanente a exibição do código-fonte e transformando a depuração mais visual. O comando é o "*winheight*". Digite isso e será aberto uma janela com o código fonte do programa e uma flecha indicando a linha em execução. Acho isso muito, útil, mas não primordial...

Bom pessoal, assim, termina nosso artigo. Eu sei que é muita coisa para ser absorvida de uma única forma, mas espero que vocês leiam, releiam e pratiquem muito a utilização do nosso depurador favorito! :D

E lembrem-se de utilizar a ajuda "online" do GDB, pois ajuda em muito o aprendizado!

[]'s
