Basic gdb

This sample program computes the Fibonacci numbers using a subroutine named “fib”.

```
#include <stdio.h>

int fib (int n)
{
  if (n <= 1)
    return 1;
  else
    return fib(n-1) + fib(n-2);
}

int main()
{
  int i, result;
  puts("Enter a positive number: ");
  scanf("%d", &i);
  result = fib(i);
  printf("\nElement %d of the Fibbionacci Sequence is %d.\n",
          i, result);
  return 0;
}
```

We're going to use this example to understand how to analyze the program only using the binary. First, assume that the program has been compiled with debugging enabled (-g) to produce “fib”, the binary. I'm showing all my input using bold text.

```
$ gcc -o fib -g fib.c
$ ./fib
Enter a positive number:
10

Element 10 of the Fibbionacci Sequence is 89.
```

The “**-g**” flag adds additional information to the binary to keep track of the how names in the program (including variables names such as “i” and “result”) relate to the actual computer memory locations used when the program runs as well as how lines in the program correspond to instructions in the resulting binary. The **“-g”** flag doesn't affect the operation of the program in any way. If you don't add the “-g” flag, you'll still be able to debug things, but only at a course level (can't see variable names, *etc*).

To see how the code executes, we'll use “gdb” to debug the binary. In this example, we start up GDB and tell it we'll be debugging the fib program. Within GDB, we can cause the program to execute by saying “**run**”.

```
$ gdb fib

GNU gdb (Ubuntu/Linaro 7.2-1ubuntu11) 7.2

Copyright (C) 2010 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 "i686-linux-gnu".

For bug reporting instructions, please see:

<http://www.gnu.org/software/gdb/bugs/>...

Reading symbols from /home/foobar/grunwald/Classes/cs2400/gdb-
tutorials/fib...done.

(gdb) run

Starting program: /home/foobar/grunwald/Classes/cs2400/gdb-tutorials/fib

Enter a positive number:

10



Element 10 of the Fibbionacci Sequence is 89.



Program exited normally.

(gdb)
```

With no other commands,

At this point, execution has stopped (allowing us to examine memory) or set other break points. We'll use these commands:

1. display – display a memory location each time we come to a breakpoint
2. examine or x - "examine" memory
3. nexti or ni – execute the next instruction, but proceed through procedure calls
4. stepi or si - "step" the next instruction (if it's a call, go into called procedure)
5. continue – continue execution until the next breakpoint
6. finish – finish up the current procedure and then resume the command line

We usually use these in certain combinations or patterns. The display and examine commands will foten use the **/i** flag for "instructions" or **/x** for "heXidemcial". You'll see that below.

We're going to use this opportunity to set an additional breakpoint for "fib" and set up a "display" to show the current instruction being executed. We'll then continue execution until "fib" is called. You'll use the following commands:

```
(gdb) break main
Breakpoint 1 at 0x8048427: file fib.c, line 14.
(gdb) display/i $pc
(gdb) run
Starting program: /tmp/fib

Breakpoint 1, main () at fib.c:14
14   fib.c: No such file or directory.
     in fib.c
1: x/i $pc
0x8048427 <main+17>:    movl   $0x8048540,(%esp)
(gdb) break fib
Breakpoint 2 at 0x80483db: file fib.c, line 5.
(gdb) c
Continuing.
Enter a positive number:
10

Breakpoint 2, fib (n=10) at fib.c:5
5    in fib.c
1: x/i $pc
0x80483db <fib+7>:  cmpl   $0x1,0x8(%ebp)
(gdb)
```

Each time we stop execution, the "display" command is causing the memory location at the current "program counter" ($pc) to be displayed as an instruction. This shows you the next instruction about to be executed.

We're going to step a few instructions in "fib" and "step into" a procedure call. We're using "si" repeatedly here. We also use the "where" command to see what procedures we're in.

```
(gdb) si
0x080483df     5    in fib.c
1: x/i $pc
0x80483df <fib+11>: jg     0x80483ea <fib+22>
(gdb) si
```

```
8    in fib.c
1: x/i $pc
0x80483ea <fib+22>: mov    0x8(%ebp),%eax
....
(gdb) si
0x080483f3     8    in fib.c
1: x/i $pc
0x80483f3 <fib+31>: call   0x80483d4 <fib>
(gdb) si
fib (n=9) at fib.c:4
4    in fib.c
1: x/i $pc
0x80483d4 <fib>:    push   %ebp
(gdb) si
0x080483d5     4    in fib.c
1: x/i $pc
0x80483d5 <fib+1>:  mov    %esp,%ebp
(gdb) where
#0  0x080483d5 in fib (n=9) at fib.c:4
#1  0x080483f8 in fib (n=10) at fib.c:8
#2  0x08048451 in main () at fib.c:16
(gdb) si
0x080483d7     4    in fib.c
1: x/i $pc
0x80483d7 <fib+3>:  push   %ebx
.....
(gdb) si
fib (n=8) at fib.c:4
4    in fib.c
1: x/i $pc
0x80483d4 <fib>:    push   %ebp
(gdb) where
#0  fib (n=8) at fib.c:4
#1  0x080483f8 in fib (n=9) at fib.c:8
#2  0x080483f8 in fib (n=10) at fib.c:8
#3  0x08048451 in main () at fib.c:16
```

After this, we're going to disable the breakpoint – you do this by identifying the breakpoint "number" and disabling that using "disable". We'll then continue execution until the program is finished.

```
(gdb) info break
Num     Type           Disp Enb Address    What
1       breakpoint     keep y   0x08048427 in main at fib.c:14
     breakpoint already hit 1 time
2       breakpoint     keep y   0x080483db in fib at fib.c:5
     breakpoint already hit 9 times
```

```
(gdb) dis 2
(gdb) c
Continuing.

Element 10 of the Fibbionacci Sequence is 89.

Program exited normally.
(gdb)
```

In this case, the program exited immediately. We might want to stop *before* it exits so we can examine parts of memory.

To do this, just set a breakpoint for procedure "exit". For example:

```
(gdb) break exit
Breakpoint 3 at 0xb7da8fb6
(gdb) run
Starting program: /tmp/fib

Breakpoint 1, main () at fib.c:14
14   in fib.c
1: x/i $pc
0x8048427 <main+17>:    movl   $0x8048540,(%esp)
(gdb) c
Continuing.
Enter a positive number:
10

Element 10 of the Fibbionacci Sequence is 89.

Breakpoint 3, 0xb7dc8fb6 in exit () from
/lib/tls/i686/cmov/libc.so.6
1: x/i $pc
0xb7dc8fb6 <exit+6>:    call   0xb7db1290
<_Unwind_Find_FDE@plt+120>
(gdb)
```

You can examine "register" values using the "info regs" command, or you can examine individual registers using their names (like we did with $pc).

```
(gdb) info reg
eax            0x0  0
ecx            0xbf800810    -1082128368
edx            0xb7ee70f0    -1209110288
ebx            0xb7ee5ff4    -1209114636
esp            0xbf8007fc    0xbf8007fc
ebp            0xbf800808    0xbf800808
esi            0xb7f1fce0    -1208877856
```

```
edi            0x0  0
eip            0xb7dc8fb6     0xb7dc8fb6 <exit+6>
eflags         0x282     [ SF IF ]
cs             0x73 115
ss             0x7b 123
ds             0x7b 123
es             0x7b 123
fs             0x0  0
gs             0x33 51
(gdb) p $eax
$1 = 0
(gdb)
```

You can also dump out regions of memory using “examine”, in any format. For example, you can print the first 5 instructions of procedure “fib” using “examine” or the alternate “dis”:

```
(gdb) x/5i fib
0x80483d4 <fib>:    push   %ebp
0x80483d5 <fib+1>:  mov    %esp,%ebp
0x80483d7 <fib+3>:  push   %ebx
0x80483d8 <fib+4>:  sub    $0x8,%esp
0x80483db <fib+7>:  cmpl   $0x1,0x8(%ebp)
(gdb) disassemble fib
Dump of assembler code for function fib:
0x080483d4 <fib+0>: push   %ebp
0x080483d5 <fib+1>: mov    %esp,%ebp
0x080483d7 <fib+3>: push   %ebx
0x080483d8 <fib+4>: sub    $0x8,%esp
0x080483db <fib+7>: cmpl   $0x1,0x8(%ebp)
0x080483df <fib+11>:     jg     0x80483ea <fib+22>
....
(gdb)
```

The name “fib” is just like the name of any other variable in your program. You can also examine data relative to registers. For example, to examine the “stack” at this point, we could do:

```
(gdb) x/16x $sp
0xbf8007fc:    0xb7ee5ff4 0xb7f1fce0 0x00000000   0xbf800868
0xbf80080c:    0xb7db1458 0x00000000 0xbf800894   0xbf80089c
0xbf80081c:    0xb7f03b38 0x00000000 0x00000001   0x00000000
0xbf80082c:    0x08048243 0xb7ee5ff4 0xb7f1fce0   0x00000000
(gdb)
```