

NOVALUG

Using the 'gdb' Debugger
Doug Toppin
31-Mar-2006

agenda

- what can it do
- debug vs print
- symbols
- commands
- memory/variables
- cores
- stack
- process control
- user macros
- more commands
- GUIs
- languages
- links

what can it do?

- why something aborted (core)
- what a running process is doing now
- current data state
- induce artificial data states (set values)
- change execution path

debugger vs prints (pros)

- many developers use prints to determine what a pgm is doing
- adding debug statements frequently adds bugs and is intrusive (changes what the code is doing)
- a debugger is less intrusive and does not require changing the code
- a debugger can frequently be used to find out what is going on

debugger vs prints (pros)

- if you're not familiar with the code you can usually more easily and quickly debug it than you can add prints
- prints change the code and add bugs

debugger vs prints (cons)

- debugger stops the process, prints do not
- prints can be put into critical points in the app to make it easier to quickly determine what it is doing w/o stopping it
- prints allow you to monitor an operational app

symbol table

- gcc '**-g**' option
- includes symbol information in the executable
- allows gdb to set breakpoints, examine values by name, and match executable to source/object file(s)
- symbol table can be generated separately and used with a stripped binary

symbol table

- not uncommon to run stripped binaries in production and retain binaries with symbols separately for debug support
- you can gdb a stripped binary and load the symbols from another binary
- is it possible to store symbol tables in a separate file?

symbol table

- related useful utilities:
 - ***strip*** (symbols from a binary)
 - ***objdump*** (display info from obj files)
 - ***readelf*** (display info about ELF files)
 - ***nm*** (display symbol table info)
 - ***addr2line*** (conv addr into file/line info)

symbol table

- binaries without symbols will look like this with gdb

```
$ gdb a  
GNU gdb 6.1
```

```
This GDB was configured as "i686-pc-linux-gnu"...(no debugging  
symbols found)...Using host libthread_db library  
"/lib/tls/libthread_db.so.1".
```

```
(gdb) break main  
Function "main" not defined.  
Make breakpoint pending on future shared library load? (y or [n])
```

basic commands

- ***run*** [args]
- ***next*** (source line)
- ***step*** (into a function)
- ***print*** (some variable)
- ***cont*** (continue running)
- ***bt*** (stack backtrace)
- ***finish*** (current stack frame)
- ***quit*** (exit the gdb session)

basic commands

- ***info*** (print all sorts of stuff)
 - args, breakpoints, files, frame, line, locals, mem, proc, registers, set, sharelibirary, signals, source, sources, stack, symbol, target, terminal, threads, types, variables, watchpoints, win, ...

list and text windows

- ***directory, path*** - specify where src, libraries are located
- ***list*** - list src lines
- ***win*** - split window, show src and control
 - **focus cmd** – set focus to cmd window
 - **focus src** – set focus to src window
 - **set tui** – config various TUI (Text User Intfc) settings

commands – win, list

```
dtoppin@localhost:~/examples/gdb/1
a.c
15  int profile1()
16  {
17      printf("profile1\n");
18      profile2();
19  }
20  static int profile2()
21  {
22      int bob;
> 23  int fred=3;
24
25      printf("profile2\n");
26      profile3();
27  }
28  int profile3()
29  {
core process 6005 In: profile2                               Line: 23   PC: 0x80483e3
$1 = 9019380
(gdb) bt
#0  profile2 () at a.c:23
#1  0x080483db in profile1 () at a.c:18
#2  0x080483be in main () at a.c:12
(gdb) list 15,26
(gdb) print bob
$2 = 9019380
(gdb) □
```

examine memory

- ***print*** variable
- ***x/20s*** variable
- ***x/s20 0x***
- static variables and addresses

examine memory

- print variable (with cast)
 - ***print*** *(thing *) generic_var
- ***set print pretty on***

```
(gdb) print fred
$1 = {a = 0, b = 0x0}
(gdb) set print pretty on
(gdb) print fred
$2 = {
  a = 0,
  b = 0x0
}
```

examine memory

```
dtoppin@localhost:~/examples/gdb/1
Defaults for format and size letters are those previously used.
Default count is 1. Default address is following last thing printed
with this command or "print".
(gdb) help format
Undefined command: "format". Try "help".
(gdb) x/s 0x80485ac
0x80485ac <_IO_stdin_used+4>:  "this is a string"
(gdb) x/x 0x80485ac
0x80485ac <_IO_stdin_used+4>:  0x73696874
(gdb) x/c 0x80485ac
0x80485ac <_IO_stdin_used+4>:  116 't'
(gdb) x/b 0x80485ac
0x80485ac <_IO_stdin_used+4>:  116 't'
(gdb) □
```

variable information

- ***ptype variable*** will show variable type information

```
(gdb) ptype bob  
type = int  
(gdb)
```

```
(gdb) ptype fred  
type = struct thing {  
    int a;  
    char *b;  
}
```

changing memory state

- you can change variables/memory
- ***set variable = 5***
- ***set {int}0x4773c0 200***

save/restore memory

- can be useful for preserving and restoring a data state (such as a hard to repro message) or examining the memory using another tool
- save a portion of memory with:
 - **dump memory mem.bin 0x80495b8 0x80495c8**
- restore memory with:
 - **restore mem.bin binary 0x80495b8**

cores and what to do with them

- why they exist
- what they contain
- how to look for useful stuff in them
- core vs core.pid
- cores are likely to be as large as the process data space (potentially immense)

stack backtrace

- ***bt***
- shows function calls still in effect to current location
- move up/down the stack with the ***frame*** command
- '***f*x**' (to access the data state at that point in execution)

frame commands

```
(gdb) bt    (stack backtrace)
#0  profile2 () at a.c:28
#1  0x080483db in profile1 () at a.c:20
#2  0x080483be in main () at a.c:14
(gdb) info locals    (variables in this frame)
bob = 9019380
fred = 3
(gdb) info f (info about current frame)
Stack level 0, frame at 0xbfe2e920:
 eip = 0x80483fa in profile2 (a.c:28); saved eip 0x80483db
 called by frame at 0xbfe2e930
 source language c.
 Arglist at 0xbfe2e918, args:
 Locals at 0xbfe2e918, Previous frame's sp is 0xbfe2e920
 Saved registers:
  ebp at 0xbfe2e918, eip at 0xbfe2e91c
(gdb)
```

attach/detach

- “grab” a running process for a debug session with '***attach*** process_id'
- process is stopped
- “release” an attached process with '***detach***', process is free'd and continues
- “***gdb -pid=PID***” to start and attach to a running process

user macros

- you can create your own common used or complicated commands (linked list traversal for example)
- ***show user*** – show all user defined macros

user macros

```
# .gdbinit
# some project specific gdb macros
document proj_request
Print a request block
end
define proj_request
    print *request_p
end
```

user macros

```
document proj_array
```

```
Print a project defined array
```

```
Usage: proj_array
```

```
end
```

```
define proj_array
```

```
    set $i = 0
```

```
    while ( $i < arrayMax )
```

```
        printf "array index %u, value %u\n", $i, array[$i]
```

```
        set $i = $i + 1
```

```
    end
```

```
end
```

user macros

```
document proj_list
```

```
Print a project defined array,
```

```
Usage proj_list NUMBER
```

```
end
```

```
define proj_list
```

```
    set $i = 0
```

```
    while ( $i < $arg0 )
```

```
        printf "array index %u, value %u\n", $i, array[$i]
```

```
        set $i = $i + 1
```

```
    end
```

```
end
```

control scripts

- useful for canned debug sessions such as preserving an involved amount of setup (breakpoints for example), can be specific to individual apps/projects
- ***.gdbinit*** read from \$HOME and then current dir also (allowing you to add to/override existing global definitions)
- `gdb [-command=.gdbrc] a.out`

.gdbinit

```
break profile1
```

```
commands 1
```

```
    bt
```

```
    cont
```

```
end
```

```
break profile2 if num == 50
```

```
commands 2
```

```
    set num = 10
```

```
    bt
```

```
    cont
```

```
end
```

gdbserver

- gdb can communicate with gdbserver on a remote machine to debug
- ***gdbserver host:2345*** a.out
- ***gdb***
 - ***target remote :2345***
 - ***symbol*** a.out

debugging threads

GDB provides these facilities for debugging multi-thread programs:

- automatic notification of new threads
- ``thread threadno'`, a command to switch among threads
- ``info threads'`, a command to inquire about existing threads
- ``thread apply [threadno] [all] args'`, a command to apply a command to a list of threads
- thread-specific breakpoints

generate-core-file

- ***generate-core-file*** file_name
- create core from current gdb state
- does not end current debugging session
- can be helpful to avoid losing a good/known debug state if you have to stop or accidentally run past something useful

tracepoints

- evaluate an expression at a location and continue running
- intent is to not disturb program behavior
- ***trace location***
- ***passcount*** allows you to pass a tracepoint n times before action is taken
- ***collect item*** will capture data

watchpoints

- **watch variable** – if variable changes stop execution
- **rwatch variable** – if variable is read stop execution
- **awatch variable** – if variable is read or written stop execution

watchpoints

```
gdb a  
watch array[2] - stop when array[2] is changed  
rwatch array[3] - stop when array[3] is read  
run
```

```
Hardware watchpoint 1: array[2]  
Old value = 0  
New value = 2  
main () at a.c:10  
10          for (i=0; i<arrayMax; i++)  
(gdb) print i  
$1 = 2  
(gdb) cont  
Continuing.  
Hardware read watchpoint 2: array[3]  
  
Value = 3  
0x080483d5 in main () at a.c:13  
13          if (array[3] == 0)
```

patching a binary

- gdb can write to a file
- ***set write on***
- “fix” a binary temporarily until a new one can be delivered/installed
- (I haven't been able to get this to work anymore on Linux but was useful at one time, used it successfully on Xenix way back when)

symbols

- ***maint print symbols*** symfile.txt

Write a dump of debugging symbol data into the file filename. These commands are used to debug the GDB symbol-reading code. Only symbols with debugging data are included. If you use ``maint print symbols'`, GDB includes all the symbols for which it has already collected full details: that is, filename reflects symbols for only those files whose symbols GDB has read.

symbols

cat symfile.txt

Line table:

line 3 at 0x8048444

line 4 at 0x804844a

Blockvector:

block #000, object at 0x9599608, 1 syms/buckets in 0x8048444..0x8048484,
compile

d with gcc2

int b(int); block object 0x9599544, 0x8048444..0x8048484

block #001, object at 0x95995d0 under 0x9599608, 1 syms/buckets in
0x8048444..

0x8048484, compiled with gcc2

typedef int int;

typedef char char;

block #002, object at 0x9599544 under 0x95995d0, 2 syms/buckets in
0x8048444

..0x8048484, function b, compiled with gcc2

int number; computed at runtime

int newnumber; computed at runtime

how does gdb work?

- how does gdb control a process?
 - ***ptrace***

DESCRIPTION

The ptrace system call provides a means by which a parent process may observe and control the execution of another process, and examine and change its core image and registers. It is primarily used to implement breakpoint debugging and system call tracing.

how does gdb work?

```
long ptrace(enum __ptrace_request request, pid_t pid, void *addr,  
            void *data);
```

__ptrace_request can be:

```
PTRACE_TRACEME  
PTRACE_PEEKTEXT, PTRACE_PEEKDATA  
PTRACE_PEEKUSR  
PTRACE_POKETEXT, PTRACE_POKEDATA  
PTRACE_POKEUSR  
PTRACE_GETREGS, PTRACE_GETFPREGS  
PTRACE_SETREGS, PTRACE_SETFPREGS  
PTRACE_CONT  
PTRACE_SYSCALL, PTRACE_SINGLESTEP  
PTRACE_KILL  
PTRACE_ATTACH  
PTRACE_DETACH
```

GUIs for gdb

- ***ddd***
- ***kdbg***

gui-*ddd*

- Data Display Debugger
- graphical front-end to gdb

DDD: /home/dtoppin/examples/gdb/1/a.c

File Edit View Program Commands Status Source Data Help

0: a.c:26

Lookup Find Clear Match Print Display Plot Show Rotate Set Undisp

```

int profile1();
static int profile2();
int profile3();

main()
{
    int number = 5;

    printf("calling b\n");
    b(number);
    printf("returned from b\n");
    profile1();
}

int profile1()
{
    printf("profile1\n");
    profile2();
}

static int profile2()
{
    int bob;
    int fred=3;

    printf("profile2\n");
    profile3();
}

int profile3()
{
    printf("profile3\n");
}

int utility(int a)
{
    printf("in utility\n");
    printf("a is %d\n",a);
}

```

in c, num = 5, newnum = 10
returned from c
returned from b
profile1
profile2

Breakpoint 2, profile2 () at a.c:26
(gdb) print fred
\$1 = 3
(gdb) |

\$1 = 3

DD

Run

Interrupt

Step Stepi

Next Nexti

Until Finish

Cont Kill

Up Down

Undo Redo

Edit Make

gui-*kdbg*

- kde front-end to gdb

a (/home/dtoppin/examples/gdb/1/a.c) - KDbg

File View Execution Breakpoint Window Help

```
+ int profile1();
+ static int profile2();
+ int profile3();
+
+ main()
+ {
+     int number = 5;
+
+     printf("calling b\n");
+     b(number);
+     printf("returned from b\n");
+     profile1();
+ }
+
+ int profile1()
+ {
+     printf("profile1\n");
+     profile2();
+ }
+ static int profile2()
+ {
+     int bob;
+     int fred=3;
+
+     printf("profile2\n");
+     profile3();
+ }
+ int profile3()
+ {
+     printf("profile3\n");
+ }
+
+ int utility(int a)
+ {
+     printf("in utility\n");
+     printf("a is %d\n",a);
+ }
+
```

Locals

```
bob 9019380
fred 3
```

KDbg: Program output

```
calling b
calling c
in c, num = 5, newnum = 10
returned from c
returned from b
profile1
profile2

```

active Line 26

different language applications

- c++
- fortran
- basic
- pascal

gdb and f77

```
dtoppin@localhost:~/examples/f77
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-redhat-linux-gnu"...Using host libthread_d
b library "/lib/tls/libthread_db.so.1".

(gdb) list
1      C=====
2      C  Simple Program to Illustrate
3      C  Fortran Programming Tools
4      C=====
5          PROGRAM F77DEMO
6          DIMENSION X(100), Y(100)
7          PI=2.*ACOS(0.)
8          N=100
9          DO 10 I=1,N
10         X(I)=I*(2*PI/N)
(gdb) break 9
Breakpoint 1 at 0x804879e: file a.f, line 9.
(gdb) run
Starting program: /home/dtoppin/examples/f77/a
Reading symbols from shared object read from target memory...done.
Loaded system supplied DSO at 0xb7f8b000

Breakpoint 1, MAIN__ () at a.f:9
9          DO 10 I=1,N
Current language: auto; currently fortran
(gdb) █
```

for more info

- <http://www.gnu.org/software/gdb/documentation/>
- http://www.linuxselfhelp.com/gnu/gdb/html_chapter/gdb_toc.html
- <http://www.delorie.com/gnu/docs/gdb/>