GDB TUTORIAL

Vladimir Slavnić
Scientific Computing Laboratory
Institute of Physics Belgrade, Serbia
http://www.scl.rs/

Scientific Computing Laboratory

Sep 18, 2009

Debug or not?

- What is debugging?
- "The best debugging is to avoid bugs"
  - good program design
  - follow good programming practices
  - always consider maintainability and readability of code over getting results fast
  - maximize modularity and code re-use
- Debugging is a last resort

Scientific Computing Laboratory

printf() or debugger?

- Using printf() (adding trace to program)
- With debugger you can:
  - attach to running process
  - change the value of variables at run-time
  - make program stop on specific conditions
  - list source code
  - print variables type
  - inspect a process that has crashed
  - ...
- Answer is obvious!

Scientific Computing Laboratory

GDB

- Symbolic debugger – part of the Free Software Foundation's GNU OS (copyleft)
- Can debug Java, C, C++, Assembly, Fortran
- Runs on any Unix architecture
- Debugging standard
- There are others:
  - DBX
  - UPS
  - PGDBG

Scientific Computing Laboratory

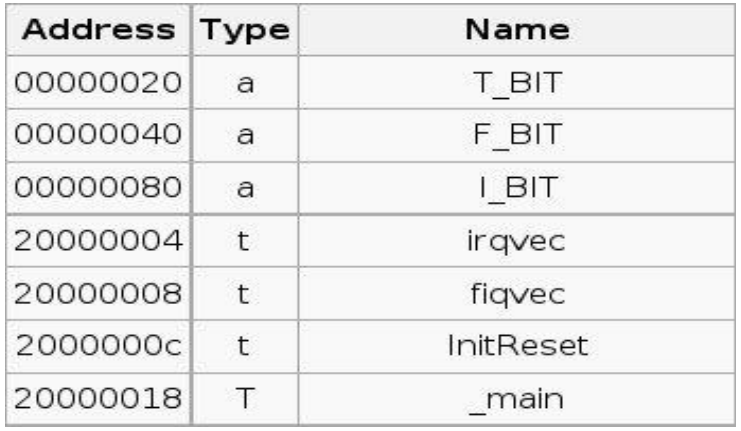
Basic usage: compiling

- Enable debugging with flags -g or –ggdb:

  gcc –g –o test test.c

- Source code and executable one to one mapping is made

- Symbol table



Address	Type	Name
00000020	a	T_BIT
00000040	a	F_BIT
00000080	a	I_BIT
20000004	t	irqvec
20000008	t	fiqvec
2000000c	t	InitReset
20000018	T	_main

- Optimization can change things!!!

Scientific Computing Laboratory

Basic usage: loading

- Load executable:

  ```
  gdb ./test
  ```

- Symbols are loaded and we can run program (VM)

- We see a command prompt:

  ```
  (gdb)_
  ```

Scientific Computing Laboratory

Basic usage: commands

- run - Start execution
- list [arg] - List source code around argument
- break [arg] - Add a “break point” at arg
- delete n - Delete break point number n
- print [arg] - Print the content of arg
- continue - Continue execution after a break
- next - Execute next line
- step - Step into next line (enters functions)
- backtrace - History of function calls
- help – Shows help
- kill - Kill program witout quitting gdb
- quit - Quit gdb

Scientific Computing Laboratory

Basic usage: run and list

- Type **run** and program will start (and finish, maybe)

  ```
  (gdb) run arg1 "arg2" ...
  ```

- **set args** – set arguments for next running

- **list** - list lines of source code (10 lines around argument are displayed):

  ```
  list
  list linenum
  list function
  list driver.c:20
  ```

- .gdbinit

Scientific Computing Laboratory

Breakpoints, Watchpoints and Catchpoints

- **breakpoint** - stops your program whenever a particular point in the program is reached
- **watchpoint** - stops your program whenever the value of a variable or expression changes
- **catchpoint** - stops your program whenever a particular event occurs

Scientific Computing Laboratory

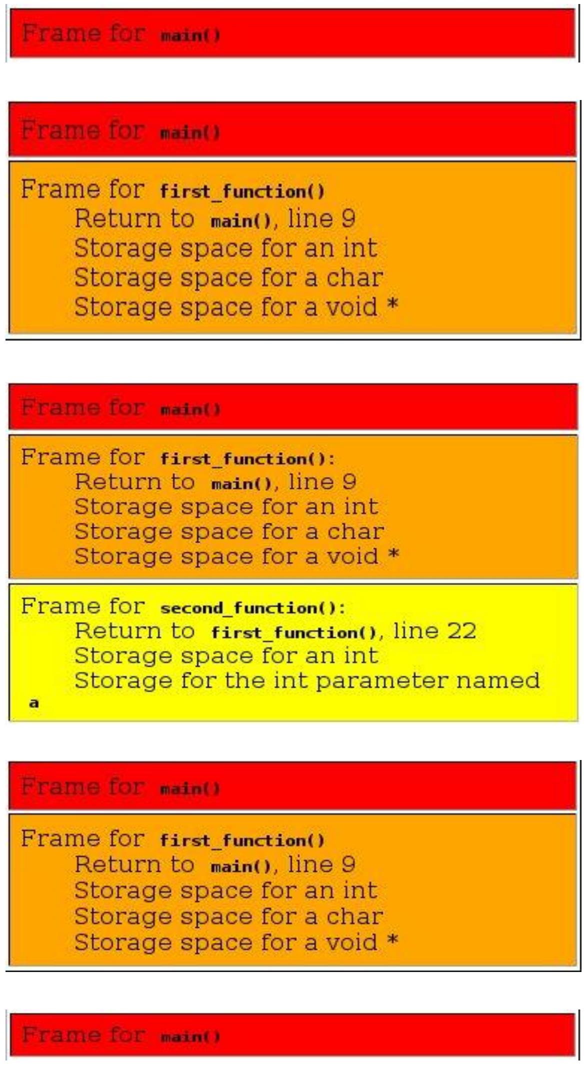
Navigating through program

- **next** - Execute a single line in the program. Skip over function calls
- **step** - Execute a single line in the program. Step into functions
- **continue** - continue program being debugged
- **advance** - continue the program up to the given location

Scientific Computing Laboratory

Call stack

```
1   #include <stdio.h>
2   void first_function(void);
3   void second_function(int);
4
5   int main(void)
6   {
7     printf("hello world\n");
8     first_function();
9     printf("goodbye goodbye\n");
10
11     return 0;
12  }
13
14  void first_function(void)
15  {
16     int imidate = 3;
17     char broiled = 'c';
18     void *where_prohibited = NULL;
19
20     second_function(imidate);
21     imidate = 10;
22  }
23  void second_function(int a)
24  {
25     int b = a;
25  }
```



Examining the stack

- **backtrace** - Print backtrace of all stack frames
- **frame** - Select and print a stack frame
- **up** - Select and print stack frame that called this one
- **down** - Select and print stack frame called by this one
- **info locals** - Local variables of current stack frame
- **info args** - Local arguments of current stack frame

Scientific Computing Laboratory

Setting Breakpoints

- Set a breakpoint at specific line on current source code file:

  (gdb) break 40

- Set a breakpoint at specific function:

  (gdb) break my_function

- Set a breakpoint at specific line on some source file :

  (gdb) break parsing.cc:45

- Add condition to a breakpoint:

  condition break_num expression

Scientific Computing Laboratory

Removing breakpoints

- **info breakpoints** - get a list of breakpoints
- **delete** – delete all break points
- **delete n** – delete breakpoint n
- **clear function** – delete breakpoint set on function
- **clear linenumber** – delete breakpoint at linenumber
- **disable n** – disable breakpoint n
- **enable n** <once, delete> – enable breakpoint n
- **ignore** - skip a breakpoint a certain number of times

Scientific Computing Laboratory

Watchpoints

- Set on variables (expressions) - variable must be in current scope
- **watch** – Set a watchpoint for an expression.
- **rwatch** - Set a read watchpoint for an expression.
- **awatch** - Set a read/write watchpoint for an expression.
- **Disable** – turn off watchpoint

Scientific Computing Laboratory

Catchpoints

- Set on events (C++ exceptions or the loading of a shared library and others)
- catch EVENT – event can be :

  throw - The throwing of a C++ exception.

  catch - The catching of a C++ exception.

  exec - A call to `exec'.

  fork - A call to `fork'.

  load - A loading of any library.

  load LIBNAME - A loading of specific library.

  unload - Unloading of library.

  thread_start – Starting any threads, just after creation . . .

Scientific Computing Laboratory

Inspecting variables 1/2

- **ptype** – print the data type of a variable

```
(gdb) ptype myvar
type = double
```

- **print** – view the value of a variable

```
(gdb) print i
$4 = -107
```

- Inspecting an array:

```
(gdb) p myIntArray
$46 = {0, 1, 2, 3, 4, 5}

(gdb) p myIntArray[3]@7
$54 = {3, 4, 5, 10, 1107293224,
       1079194419,-1947051841}
```

Scientific Computing Laboratory

Inspecting variables 2/2

- Inspecting a structure:

  ```
  (gdb) p myStruct
  $2 = {name = 0x40014978 “Mile mikic",
  EyeColour = 1}

  (gdb) print myStruct.name
  $6 = 0x40014978 "Mile Mikic“
  ```

- **set** - Changing variable value (must be in current context):

  ```
  (gdb) set  myvariable = 10.0
  ```

- All Fortran variables must be in lowercase!!!

Scientific Computing Laboratory

Debugging a Running Process

- **attach pid** (from gdb) - attach to the running process with pid

```
$ gdb
(gdb) attach 17399
Attaching to process 17399....
```

- **$ gdb program pid** (outside gdb) –

```
Attaching to program: code/running_process/some-process, process 17399
0x410c64fb in nanosleep () from /lib/tls/libc.so.6
(gdb)
```

- **detach** – detach from process
- Change variables

Scientific Computing Laboratory

Attach to a running process

```
#include <stdio.h>
#include <unistd.h>
static void PrintMessage(int i);
static void GoToSleep(void);
int main(void)
{
        int i = 100000;
        while ( 1 )
        {
                PrintMessage( i );
                GoToSleep();
                i -= 1;
        }
        return 0;
}
void PrintMessage(int i)
{
 printf("%d bottles of beer on the wall.\n", i);
}
static void GoToSleep(void)
{
        sleep(3);
}
```

Scientific Computing Laboratory

Segmentation Fault Example (1/2)

```
#include <iostream>

using namespace std;;

void do_stuff(void) {
    int *i;
    i = NULL;
    *i = 1;
}

int main(void) {
    cout << "Hello world" <<endl;
    do_stuff();
    return 0;
}
```

Scientific Computing Laboratory

Segmentation Fault Example (2/2)

- Common pointer pitfalls:
  - dereferencing a NULL pointer
  - dereferencing an uninitialized pointer
  - dereferencing a deleted pointer
  - deleting an uninitialized pointer
  - deleting a pointer twice
  - writing beyond the bounds of an array
- Right usage

```
    p = (char *) malloc(100);
    if ( p == NULL)
      { printf(``Error: Out of Memory \n'');
                exit(1);     }
    *p = `y';
```

Scientific Computing Laboratory

Debugging Programs with Multiple Threads

- **info threads** – display a summary off all threads in program

```
(gdb) info threads
  3 process 35 thread 27  0x34e5 in sigpause ()
  2 process 35 thread 23  0x34e5 in sigpause ()
* 1 process 35 thread 13  main (argc=1,
argv=0x7ffffff8)
   at threadtest.c:68
```

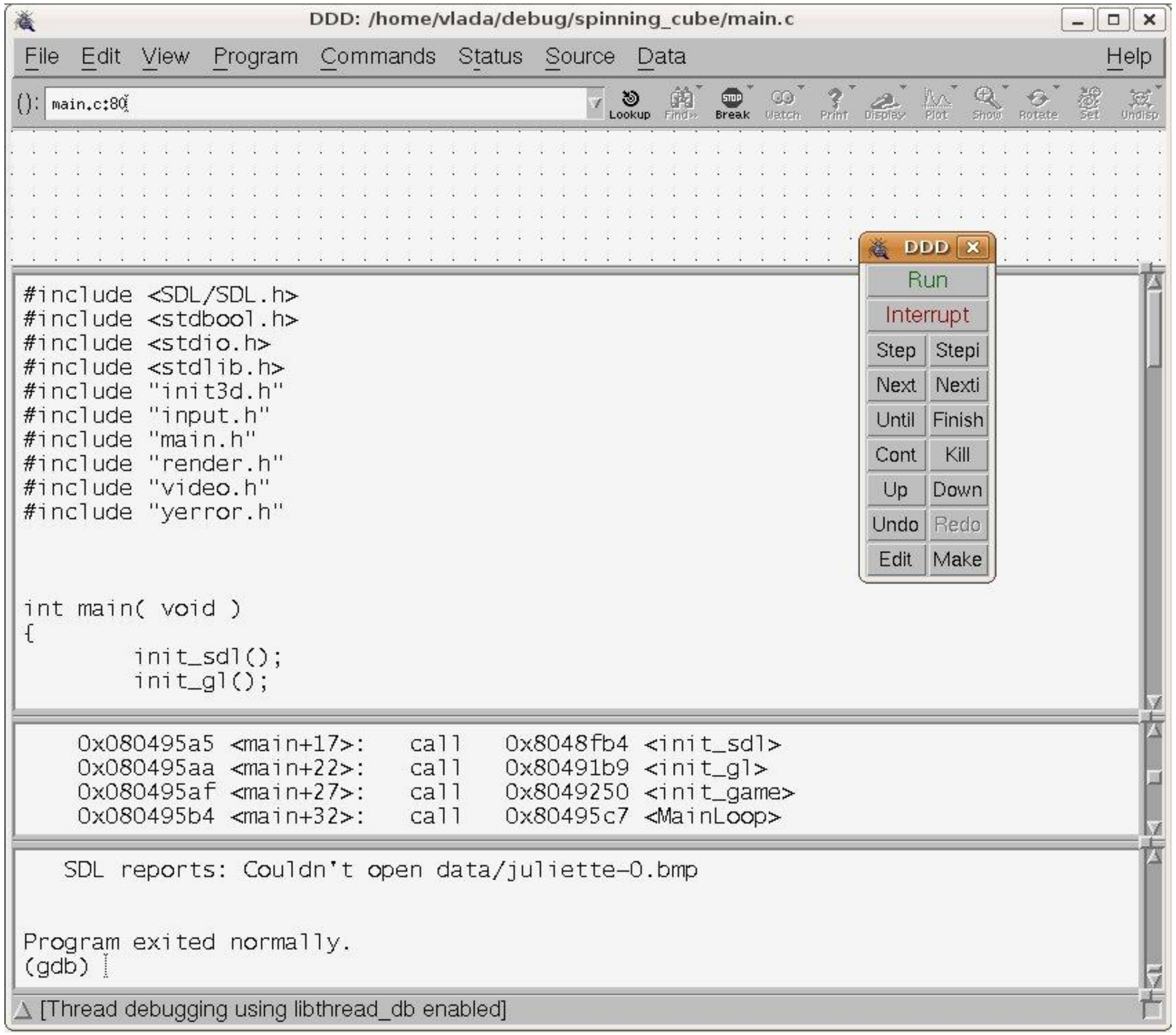
- **thread thread_num** – make thread number thread_num current

Scientific Computing Laboratory

Infinite loop example

```c
#include <stdio.h>
#include <ctype.h>

int main(int argc, char **argv)
{
  char c;

  c = fgetc(stdin);
  while(c != EOF){

      if(isalnum(c))
        printf("%c", c);
     else
        c = fgetc(stdin);
   }

   return 1;
}
```

DDD - GDB GRAPHICAL FRONTEND


DDD: /home/vlada/debug/spinning_cube/main.c
File Edit View Program Commands Status Source Data Help
(): main.c:80
Lookup Find» Break Watch Print Display Plot Show Rotate Set Undisp
DDD
Run
Interrupt
Step Stepi
Next Nexti
Until Finish
Cont Kill
Up Down
Undo Redo
Edit Make

```
#include <SDL/SDL.h>
#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>
#include "init3d.h"
#include "input.h"
#include "main.h"
#include "render.h"
#include "video.h"
#include "yerror.h"


int main( void )
{
        init_sdl();
        init_gl();
```

```
    0x080495a5 <main+17>:   call    0x8048fb4 <init_sdl>
    0x080495aa <main+22>:   call    0x80491b9 <init_gl>
    0x080495af <main+27>:   call    0x8049250 <init_game>
    0x080495b4 <main+32>:   call    0x80495c7 <MainLoop>
```

```
   SDL reports: Couldn't open data/juliette-0.bmp


Program exited normally.
(gdb)
```

[Thread debugging using libthread_db enabled]

Scientific Computing Laboratory

References:

- http://www.gnu.org/software/gdb/
- http://www.dirac.org/linux/gdb/
- http://www.delorie.com/gnu/docs/gdb/gdb_toc.html

Scientific Computing Laboratory