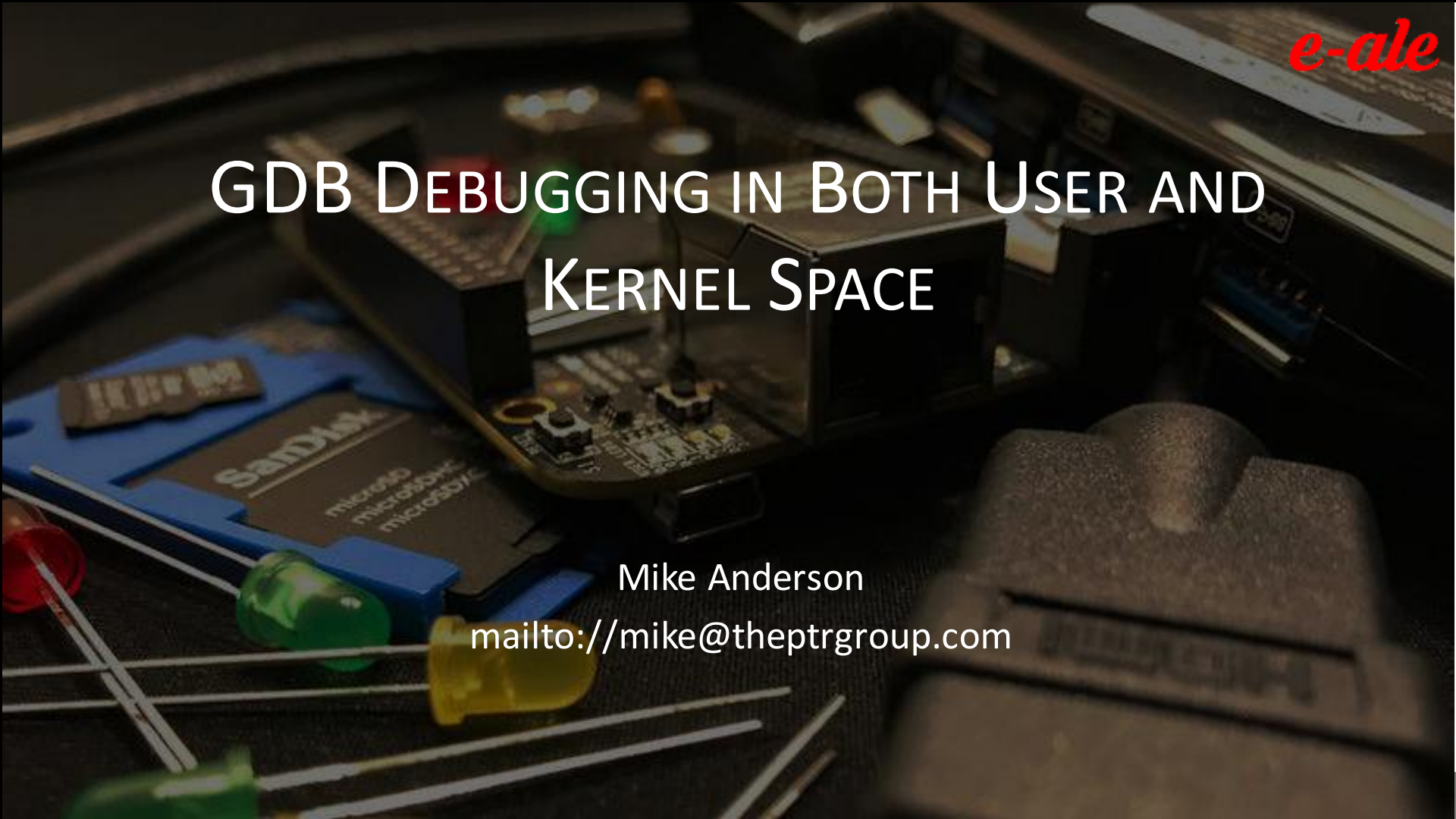


GDB Debugging in Both User and Kernel Space

Mike Anderson

mailto://mike@theptrgroup.com

What We Will Talk About…

- The GNU Project, GCC and gdb
- Native vs. cross debugging
- Compiling for debugging
- gdb CLI, TUI and gdbfront-ends
- Getting help, scripts and macros
- Launching, loading and running applications
- Attaching to a running application
- Breakpoints, watchpoints, catchpoints and more
- gdbserver and its options
- Setting up for kernel debugging
- Running the kernel with kgdb

The GNU Project and GCC

- The ostensible goal of the GNU Project was to create a Un*x clone without any AT&T sources
  - GNU's Not Unix
- First, there was the GNU C compiler (gcc)
  - Later there was a C++ version (g++)
  - Architected as a front-end language parser and a back-end code generator
  - Also added numerous *binutils* such as the linker, librarian, etc.
- gcc was originally targeted as an OS bring-up tool and was completely command-line driven
- As more language front-ends were added, it became GCC
  - GNU Compiler Collection
    - Front ends for C, C++, Objective-C, FORTRAN, Ada and Go with associated libraries like libstdc++, etc.
- Community supported and peer reviewed with support from major silicon manufacturers

GDB

- The GNU debugger (gdb) was built as a source debugger for GCC
  - gdb supports Ada, Assembly, C/C++, D, FORTRAN, Go, Objective-C, OpenCL, Modula-2, Pascal and Rust
    - No direct Python or Java support (gcj was dropped from GCC in 2017)
- Also designed to be run from a CLI
  - But, there are numerous GUI front-ends to gdb

Native vs. Cross Debugging

- This mirrors the development approach
  - Native means it's running on the platform your debugging the code on
  - Cross debugging is running gdb on a host computer and talking to the code via a connection to, typically, a foreign CPU architecture
- Native debugging may leave you with using the CLI if there isn't enough horsepower or hardware on the target to run a window manager or VNC
- Cross debugging allows you to use the horsepower of your development host to run the debugger while the code runs on the target via a *helper* application
  - Cross debugging adds connection latency and complexity
    - The ease of use of the GUI front end may be worth it

Compiling for Debugging

- gdb will typically require the source code to be specially compiled for debugging
  - You can still debug non-debug code if you really like assembly language
- Uses the DWARF debugging standard file format
  - The debugging information can be extracted from the executable using objcopy or strip
- To build the code with debugging enabled, we use the `-g` option to the compiler
  - But, there are several levels of debug information we can include

Source: dwarfstd.org

GDB Debug Levels

- `-g0` will explicitly produce no debug information
- `-g1` produces minimal information, enough for making back traces, but no information about local variables and no line numbers
- `-g2` default debug level when not specified. Typically this will produce symbols, line numbers etc needed for symbolic debugging
  - This is the default for the `-g` option to the compiler
- `-g3` includes extra information, such as all the macro definitions present in the program

Example Compile for GDB

- Example compilation to enable debugging

```
$ arm-linux-gnueabi-gcc -g3 -o hello helloWorld.c
```

- Example for examining the debug info in ELF header

```
$ arm-linux-gnueabi-objdump -h hello
…
 24 .comment      0000002a  00000000  00000000  00000a97  2**0
                  CONTENTS, READONLY
 25 .debug_aranges 00000020  00000000  00000000  00000ac1  2**0
                  CONTENTS, READONLY, DEBUGGING
 26 .debug_pubnames 00000031  00000000  00000000  00000ae1  2**0
                  CONTENTS, READONLY, DEBUGGING
 27 .debug_info   00000179  00000000  00000000  00000b12  2**0
                  CONTENTS, READONLY, DEBUGGING
 28 .debug_abbrev 000000d4  00000000  00000000  00000c8b  2**0
                  CONTENTS, READONLY, DEBUGGING
 29 .debug_line   000003ea  00000000  00000000  00000d5f  2**0
                  CONTENTS, READONLY, DEBUGGING
 30 .debug_frame  00000090  00000000  00000000  0000114c  2**2
                  CONTENTS, READONLY, DEBUGGING
 31 .debug_str    000000ea  00000000  00000000  000011dc  2**0
                  CONTENTS, READONLY, DEBUGGING
 32 .debug_loc    00000058  00000000  00000000  000012c6  2**0
                  CONTENTS, READONLY, DEBUGGING
 33 .debug_macinfo 00009e52  00000000  00000000  0000131e  2**0
                  CONTENTS, READONLY, DEBUGGING
```

Running GDB CLI

- When gdb is built from source, we will specify the host and target environments
  - Allows for Windows x Linux, x86 host x ARM target, etc.
- When running gdb from the CLI, we can just use the gdb command:

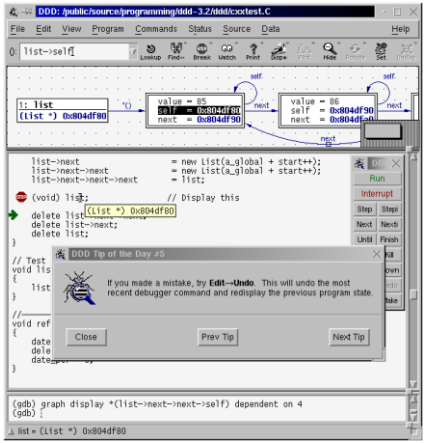
```
GNU gdb (Ubuntu 7.11.1-0ubuntu1~16.5) 7.11.1
Copyright (C) 2016 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later
  <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word".
(gdb)
```

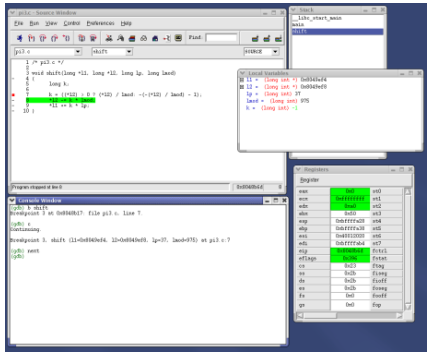
Running gdb in TUI Mode

- TUI mode is a text-based user interface that separates out the program text from the gdb command line
- More clear cut than using the typical CLI, but maybe not as good as the GUIs for gdb like ddd
- You can start gdb in TUI mode using the –TUI command line argument
- You can switch in and out of TUI mode using <CTRL> X A keyboard sequence

GDB GUIs

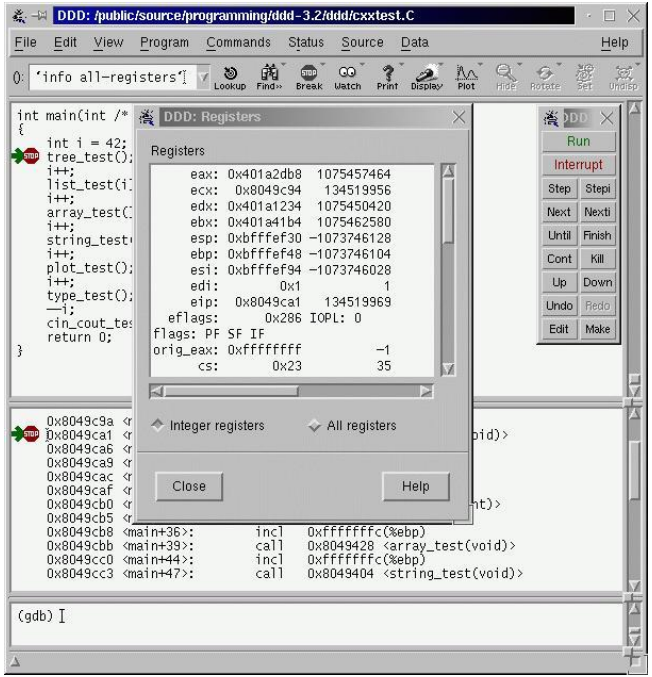
- There are standalone and IDE-based front ends to gdb
- These include:
  - ddd
    - Data Display Debugger
      - Also works with cross debugging
      - http://www.gnu.org/software/ddd/
  - insight
    - Redhat-developed
    - MDI GUI approach
    - http://sourceware.org/insight/
- IDE support includes Eclipse, Kdevelop, Slickedit®, CodeWarrior®, Arriba® and more





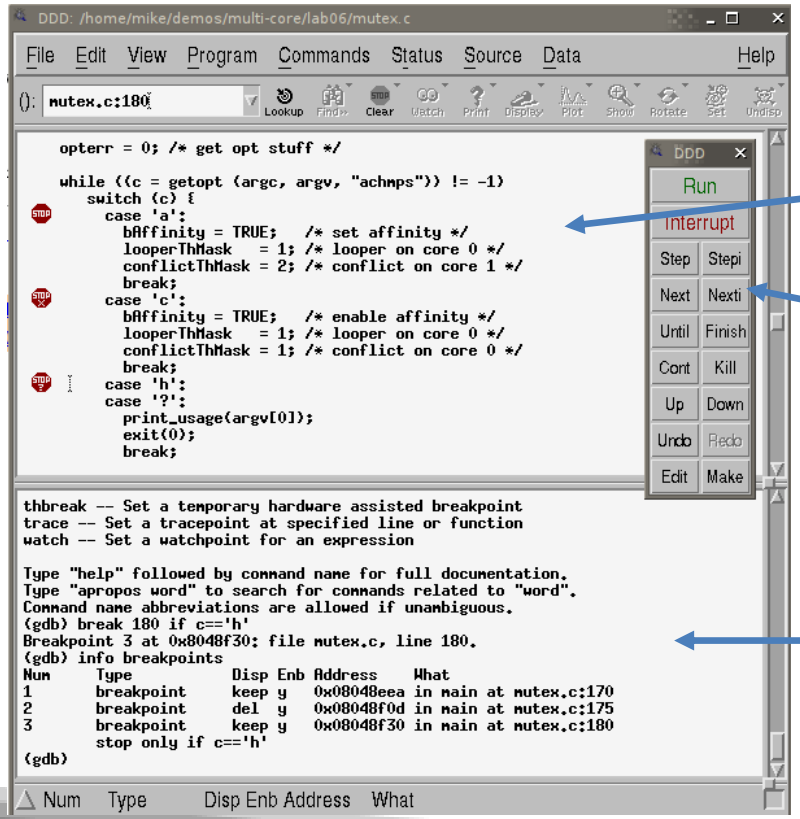
ddd Front End GUI

- ddd is the GNU-supported graphical interface for gdb
- ddd supports:
  - gdb, jdb, Python, Perl, TCL and PHP
- You can automatically load the application into gdb at invocation
- ddd can be started with the **`-debugger`** option to run a gdb backend other than the default gdb instance




```
$ ddd -debugger arm-linux-gnueabi-gdb myapp
```

The ddd GUI



Main code window

Command Shortcuts

GDB command line

Getting Help in gdb

- Whether using the CLI or the GUI, you should have access to the gdb command line
  - It looks like `(gdb)`
- You can use the help command at the `(gdb)` prompt for any query
- Use the apropos <word> command to find which gdb commands might apply to your query
- Throughout the gdb help output, you will see references to "the inferior"
  - Yes, that's your program – don't get a complex about it!
    - As far as gdb is concerned, your program is running under gdb's control

Example Help Output

```
(gdb) help
List of classes of commands:

aliases -- Aliases of other commands
breakpoints -- Making program stop at certain points
data -- Examining data
files -- Specifying and examining files
internals -- Maintenance commands
obscure -- Obscure features
running -- Running the program
stack -- Examining the stack
status -- Status inquiries
support -- Support facilities
tracepoints -- Tracing of program execution without stopping the program
user-defined -- User-defined commands

Type "help" followed by a class name for a list of commands in that class.
Type "help all" for the list of all commands.
Type "help" followed by command name for full documentation.
Type "apropos word" to search for commands related to "word".
Command name abbreviations are allowed if unambiguous.
```

Command Definition and Macros

- gdb has the ability to define your own commands/scripts
- Use the **`define <name>`** command to define a sequence of gdb commands
  - Enter each one line-by-line and finish with a single line "**`end`**"
    - Useful for creating debugging command scripts that you can save for later use or just to save repeated typing
- Use the **`document <name>`** to write documentation for your defined commands
  - Again, enter each one line-by-line and finish with a single line "**`end`**"
- There is also the ability to define C/C++ preprocessor macros using the **`macro define`** command
  - Visible to all of the inferior's source files

gdb Scripts

- If it exists, gdb will execute all of the commands found in `.gdbinit` in the current directory
- Useful for executing a sequence of gdb commands at gdb initialization
- The `-x` command line option to gdb also allows for running scripts at gdb load time

Examining Code

- Once the program is loaded in gdb, you can list any of the source files using the **`list`** command
  - Options to list a **`LINENUM`**, **`FILE:LINENUM`**, **`FUNCTION`**, **`FILE:FUNCTION`** or **`*ADDRESS`**
- You can specify the number of lines to list as a second parameter
  - Defaults to 10 but can be changed with **`set listsize`**

Manipulating Internal Settings

- gdb has dozens internal options like setting the radix, terminal type, command history size and more
  - Use **`show`** on the gdb command line to see them all
- You can change the options using the **`set`** command
  - E.g., **`set output-radix 16`** would set the display radix to hexidecimal

Load/Execute Your Code

- If you don't load the program from the command line, you can load additional files using the **`file <filename>`** command
- Once the code is loaded into gdb, you can execute it using the **`run`** command
- You can pass parameters in the same command or you can use the **`set args`** command
  - **`show args`** will allow you to see the arguments

Calling Functions Interactively

- Once the code is loaded, you can actually call program functions from the gdb command line
  - The syntax and parameter passing is based on the language the code is written in
- The function will be called and the return will be printed and saved in the value history

Setting Variables

- You can define new variables, set a register value or modify program variables using the **`set VAR = EXP`** (or whatever the language equivalent is for your language)
  - Expressions are any valid expression for the language
- You can set a variable that uses the same name as a gdb command using the **`set variable VAR = EXP`** syntax

Printing Expressions

- Use the **`print EXP`** syntax to print any value from the current stack frame, globals or an entire file
- Use $NUM to get the previous value of NUM
  - You can refer back farther using $$NUM
- Registers are accessed using the $<REGNAME> syntax
- {TYPE}ADREXP refers to datum of data type {TYPE} located at address ADREXP
- The @ symbol is a binary operator for treating consecutive data objects anywhere in memory as an array
  - E.g., FOO@NUM gives an array whose first element is foo, whose second is stored in the memory adjacent to FOO, etc.

Printing Expressions #2

- EXP may be preceded with a /FMT modifier where /FMT is a single character without a length modifier
- Format letters are o(octal), x(hex), d(decimal), u(unsigned decimal), t(binary), f(float), a(address), i(instruction), c(char), s(string) and z(hex, zero padded on the left)
- The specified number of objects of the specified size are printed according to the format
- `print` can also dereference addresses ala

```
(gdb) print *ptr
```

The Difference Between Print and Display

- There is another mechanism for displaying the value of a variable that that is the **`display`** command
- Like print, display can show the variable with a format
- However, display prints the variable's value after every step
  - Useful to see where a variable changes

Examine Memory at Address

- If we have a known address in memory, such as a pointer, we can display the memory at that address
- We can also use a format character but `x` also adds an additional character to indicate the size of the display
- Size letters are b(byte), h(halfword), w(word), g(giant, 8 bytes)
- E.g., `(gdb) x /db &test`

Setting Breakpoints

- gdb supports several types of breakpoint
  - Normal breakpoints `b <lineno>`
  - Temporary breakpoints `tbreak <lineno>`
  - Hardware breakpoints `hbreak <lineno>`
- Often the gdb GUI frontends normally have a menu option for these
- You can also set conditional breakpoints
  - Beware of the overhead of these

```
(gdb) break <lineno> if <condition is true>
```

Setting Breakpoints #2

- Use symbols to identify a breakpoint

```
(gdb) b main
Breakpoint 1 at 0x8048355: file watch.c, line 5
(gdb) run
Starting program:/home/pocket/debug/watch
Breakpoint 1, main() at watch.c:5
```

- You can issue commands to be run when the breakpoint is hit

```
(gdb) After b main
(gdb) commands
Type commands to be executed when breakpoint 1 is hit, one per line
End with a line saying just "End"
> silent
> printf " main started\n"
> cont
> end
```

Stepping Through Code

- gdb has a number of ways to step through the code after encountering a breakpoint
- Step – step one line and step into functions
- Next – step one line and step over functions
- Finish – you stepped into a function accidentally and you want to finish this routine
- Stepi – step one assembly language instruction and step into function calls
- Nexti – step one assembly language instruction but step over function calls

Attach to a Running Program

- gdb has the ability to attach to a running program

```
$ gdb -p <process id>
```

- This will stop the running program at its current execution point
- You can then load the executable's code and symbol table using the **`file`** command to load the source if it hasn't already been loaded

Monitoring a Variable

- Often, we'll find ourselves with a variable that's getting stepped on by something
  - We're just not sure where or when
- To help locate these occurrences, gdb has an option known as a **`watchpoint`**
- Uses a hardware breakpoint (if available)

Watchpoints

- Force a program break when a selected variable's value changes
- Show old and new value and location in code that caused the change in value

```
(gdb) b main
(gdb) run
(gdb) watch x
(gdb) cont
Hardware watchpoint 2: x
Old value = 13451357
New value = 10
Main() at watch.c:10
```

Breaking on Events

- gdb also has a means of breaking when certain events occur
  - These include exceptions, calls to fork, loading shared libraries, signals, syscalls, unloading shared libraries, calls to exec and much more
- We can also reinject a exception with the **`rethrow`** options

Catchpoints

- Used for C++ catch and throw events as well as Ada failed assertions

```
(gdb) catch throw
(gdb) catch catch
```

- Use **`tcatch`** for a one-time catch that is automatically deleted after the first time it's caught

Deleting Breakpoints, Watchpoints, etc.

- To list breakpoints, etc. use:

```
(gdb) info breakpoints
```

- To delete a breakpoint, watchpoint or catchpoint use:

```
(gdb) delete [breakpoint]
```

- Clear breakpoints associated with functions

```
(gdb) clear |function| line num | file:function |
  file:line |
```

- At any point, you can use the help feature to understand your options e.g.,:

```
(gdb) help breakpoints
```

- There are several good books, websites and manual pages on the features of gdb

Debugging with Optimization

- Ideally, if you're debugging code, you should disable any code optimization
  - But, sometimes that is impossible due to in-lining of functions like in the kernel
- You can debug optimized code
  - However, be prepared for some odd behavior on the part of the debugger
    - Variables might be removed
    - The instruction pointer can go backwards
    - The debug is still valid, but the behavior can be disconcerting

Source: blogspot.com

Accessing Local Variables

- Show local symbols

```
(gdb) info locals
```

- Show CPU registers

```
(gdb) info registers
```

- Display stack back trace

```
(gdb) bt (backtrace)
```

- Gives detailed information on the current stack frame

```
(gdb) info frame
```

Working with Signals via gdb

- Show signals

  ```
  (gdb) info signals
  ```

- Prints a table of how signals and how gdb will handle each one

  ```
  (gdb) info handle
  ```

- Change the way gdb will handle the signal
  - nostop – do not stop the program but still print that signal occurred
  - stop – stop program when signal occurs (implies print as well)
  - print – print a message when signal occurs
  - noprint – do not mention the occurrence of the signal
  - pass – allow your program to see the signal so it can be handled
  - nopass – do not pass the signal to your program

  ```
  (gdb) handle signal keywords
  ```

- Delivers a SEGV signal to the current program

  ```
  (gdb) signal SIGSEGV
  ```

Debugging Threads

- Show active thread ids

  `(gdb) info threads`
- Select a thread by id

  `(gdb) thread n`
- Restrict breakpoint to a particular thread

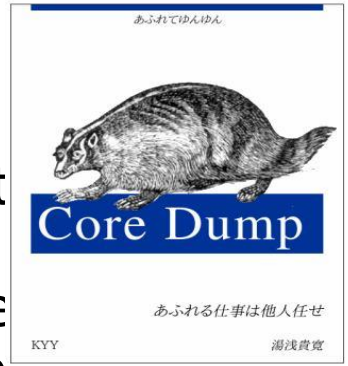
  `(gdb) break <break ident> thread <id>`
  - Not specifying a thread ID will cause the breakpoint to apply to all threads
- Restrict thread execution to current thread

  `(gdb) set scheduler-locking on | off`

Source: optusnet.com.au

Generating “Core Dumps”

- When an application terminates abnormally, a core file can be generated
  - core file - (n.) A file created when a program malfunct and terminates. The core file holds a snapshot of memory, taken at the time the fault occurred. This file can be used to determine the cause of the malfunction
- By default, this feature is disabled to preclude “core droppings” in the file system
  - Use:
    `$ ulimit -c <max core file size in disk sectors>`
    to re-enable core file generation



Source: heartrails.com

Checking Core Dumps Settings

- First, check to see the default settings in your system

```
$ ulimit -a
core file size          (blocks, -c) 0
data seg size           (kbytes, -d) unlimited
scheduling priority             (-e) 20
file size               (blocks, -f) unlimited
pending signals                 (-i) 16382
max locked memory       (kbytes, -l) 64
max memory size         (kbytes, -m) unlimited
open files                      (-n) 1024
pipe size            (512 bytes, -p) 8
POSIX message queues     (bytes, -q) 819200
real-time priority              (-r) 0
stack size              (kbytes, -s) 8192
cpu time               (seconds, -t) unlimited
max user processes              (-u) unlimited
virtual memory          (kbytes, -v) unlimited
file locks                      (-x) unlimited
```

Core Dump pid-centric Files

- Generating separate core files per process when multiple applications die simultaneously
- A proc entry already exists for this

```
/proc/sys/kernel/core_uses_pid
```

- Turn it on by

```
$ sudo echo "1" >
  /proc/sys/kernel/core_uses_pid
```

- Now, each process generating a core dump will have a corresponding file `core.<pid>` on rootfs

Using the Core File

- Once you have a core file, you can use gdb to try to determine what went wrong
- Load the core file using:

```
$ gdb <application name> -core <corefile>
```

- gdb will load the application and will show you the point of failure
- This will also work with most gdb front-ends
  - eclipse
  - ddd
  - Insight

Core Dump Debugging Example #1

- Application that produces a core dump
- app1.c

```
/* global variables */
int value = 20;
int divide = 4;
int sum = 0;
int main(int argc, char *argv[])
{
    volatile int result, x;
    for(x = 0; x < NUM_ITERATIONS; x++)
    {
      printf("value = %d divide = %d sum = %d\n", value, divide, sum);
      result = compute_it(value, divide);
      sum += result;
      divide++;
      value--;
    }
    return 0;
}
```

Core Dump Debugging Example #2

```
/*****************************************************************************
* compute_it - support routine
*
*/
int compute_it(int no1, int no2)
{
    volatile int result;
#ifdef FP_ERROR    /* Generate an arithmetic/floating point error */
    volatile int diff;
    diff = no1 - no2;
    result = no1 / diff;
#else if           /* Generate a segmentation violation error */
    int * ptr;
    *ptr = 0;
#endif
    return result;
}
```

Core Dump Debugging Example #3

- Build the application with debug info in the ELF

```
$ gcc -g3 -o app1 app1.c
```

- Run the application and generate core file

```
$ ./app1
value = 20 divide = 4 sum = 0
Segmentation fault (core dumped)
```

- Invoke gdb to analyze the core file

```
$ gdb app1 -core core
```

Core Dump Debugging Example #4

- gdb command line console output

```
GNU gdb (Ubuntu 7.11.1-0ubuntu1~16.5) 7.11.1
Copyright (C) 2016 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 "x86_64-linux-gnu".
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>...
Reading symbols from /home/student/debug/linux_debug_perf_labs/lab02/app1...done.
[New Thread 15231]

warning: Can't read pathname for load map: Input/output error.
Reading symbols from /lib/libc.so.6...(no debugging symbols found)...done.
Loaded symbols for /lib/libc.so.6
Reading symbols from /lib64/ld-linux-x86-64.so.2...(no debugging symbols found)...done.
Loaded symbols for /lib64/ld-linux-x86-64.so.2
Core was generated by `./app1'.
Program terminated with signal 11, Segmentation fault.
#0  0x00000000004005d6 in compute_it (no1=20, no2=4) at app1.c:70
70      *ptr = 0;
(gdb)
```

Core Dump Debugging Example #5

- Examine the core file for additional information

```
(gdb) info locals
  result = 32637
  ptr = 0x7f7d81d30250
(gdb) info frame
  Stack level 0, frame at 0x7fff55ea7a20:
  rip = 0x4005d6 in compute_it (app1.c:70); saved rip 0x400578
  called by frame at 0x7fff55ea7a60
  source language c.
   Arglist at 0x7fff55ea7a10, args: no1=20, no2=4
  Locals at 0x7fff55ea7a10, Previous frame's sp is 0x7fff55ea7a20
  Saved registers:
  rbp at 0x7fff55ea7a10, rip at 0x7fff55ea7a18
```

Core Dump Debugging Example #6

```
(gdb) info registers
rax            0x7f7d81d30250     140177025729104
rbx            0x14 20
rcx            0x400700           4196096
rdx            0x4  4
rsi            0x4  4
rdi            0x14 20
rbp            0x7fff55ea7a10     0x7fff55ea7a10
rsp            0x7fff55ea7a10     0x7fff55ea7a10
r8             0x1  1
r9             0x400700           4196096
r10            0x7fff55ea7030     140734634815536
r11            0x7f7d81a1c060     140177022500960
r12            0x400440           4195392
r13            0x7fff55ea7b30     140734634818352
r14            0x0  0
r15            0x0  0
rip            0x4005d6           0x4005d6 <compute_it+14>
eflags         0x10206            [ PF IF RF ]
cs             0x33 51
ss             0x2b 43
ds             0x0  0
es             0x0  0
fs             0x0  0
gs             0x0  0
```

Core Dump Debugging Example #7

```
(gdb) info stack
#0  0x00000000004005d6 in compute_it (no1=20, no2=4) at app1.c:70
#1  0x0000000000400578 in main (argc=1, argv=0x7fff55ea7b38) at app1.c:46
(gdb) info variables
All defined variables:

File app1.c:
int divide;
int sum;
int value;

Non-debugging symbols:
0x00000000004006d8  _IO_stdin_used

0x0000000000601018  __dso_handle
0x0000000000601028  completed.7382
0x0000000000601030  dtor_idx.7384
0x0000000000000008  __resp
0x0000000000000010  errno
0x0000000000000054  h_errno
```

What to Remember about Core Files

e-ale

- The key characteristic of a core file is that it is post mortem
  - The application is already dead and no amount or wishing on your part will bring it back
- Therefore, you are limited to examining the state of the software when the error occured

Cross Debugging

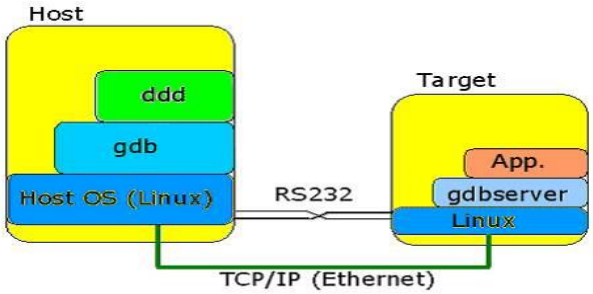
- While many of today's more modern development boards have sufficient RAM and CPU performance to support native debugging, you can often achieve a better experience using cross debugging
- We will run gdb on the host and a debug agent on the target to facilitate debugging
- We will use a connection between the target and host computer to enable cross debugging

Using gdb with Cross Debugging

- As indicated earlier, gdb has the **`–debugger`** feature to allow us to specify an alternate back end for the debugger
  - E.g., we can debug ARM code from an x86
- We will need to use the **`target remote`** command in gdb to connect to a waiting debug agent

Remote Debugging with gdb/gdbserver

- The DWARF debug format does not change because we're using an alternate processor type
- We will load the code to be debugged into the local gdb session and then connect with the remote gdbserver
  - Communications with the gdbserver helps keep the gdb session in sync
- The application to be debugged runs under the control of the gdbserver application
  - Uses `ptrace()` functions to control starting and stopping of the target application



Source: codeproject.com

Debugging with gdb/gdbserver #2

- Remote gdb debugging offers a flexible client/server design with many different target communications options
  - Serial
    - Has option to allow the target console to share the same serial port with the target console
  - Network
  - Direct hardware communications e.g., JTAG
- It is possible to use the non-debug enabled code on the target as long as the gdb has access to a debug-enabled version of the exact same code
  - Avoids memory footprint concerns on the target

gdb/gdbserver Cross Debug Example

e-ale

- For example:
  - On the target:

```
$ gdbserver 192.168.7.1:1929 myapp &
```

  - On the host:

```
(gdb) target remote 192.168.7.2:1929
```

- gdbserver can attach to a running program

```
$ gdbserver hostIP:2345 --attach PID
```

- Works within GUI-based front-ends as well
  - You must tell gdb which back-end to use

```
$ ddd –debugger arm-linux-gnueabi-gdb myapp
```


PTR

gdb/gdbserver Debugging Options

- `target remote` command options
  - serial port device (/dev/ttyS0)
  - tcp/ip address:port (e.g., 192.168.100.2:4000)
  - udp/ip address:port (e.g., udp:192.168.100.2:4000)
- Use `set debug remote 1` to see packets sent to and from the agent
- Once connected, a remote gdb/gdbserver session is just like a native gdb session

Running Your Program with gdbserver

- If you're using gdbserver, you can't "run" you can only "continue"
  - The program is already running under the control of the gdbserver helper
- You can examine your program's arguments using the **`show args`** command
  - You can change the arguments for the next run using **`set args`** command

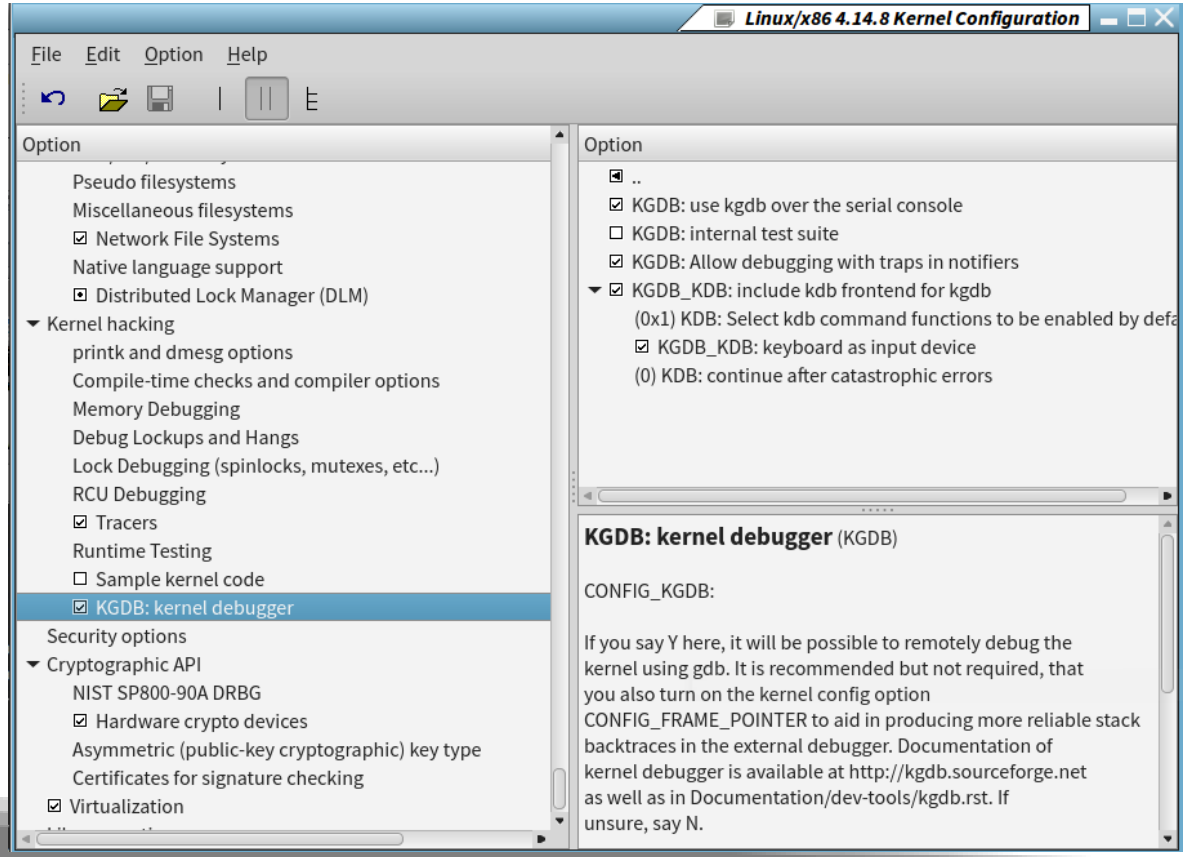
Debugging the Kernel with gdb

- For many years, Linus fought against including a source debugger into the kernel
  - “If you can’t debug with printk, you shouldn’t be in the kernel”
- Much of this reluctance was due to the implementation of kgdb at the time
  - It touched hundreds of files to patch the debugger in
- Late in the 2.6 kernel series, a new kgdb lite version was developed and we finally got a debugger!

Compiling the Kernel with Debug Info

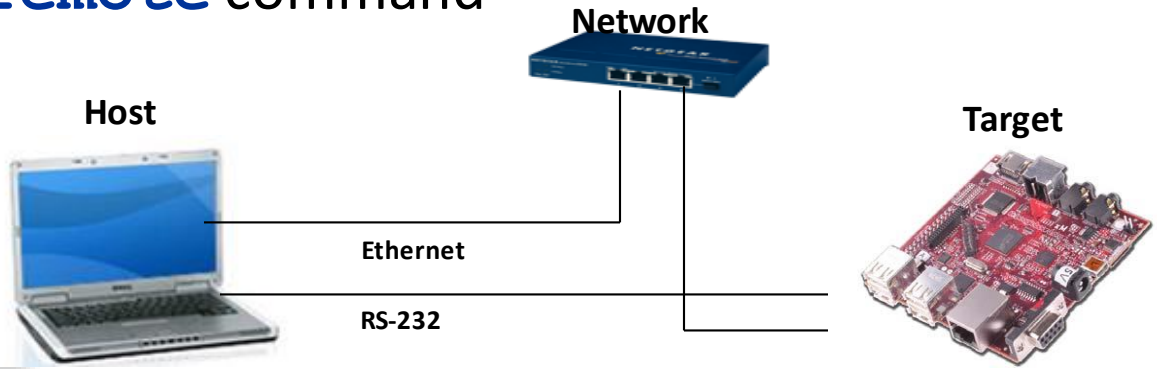
- This will increase the size of the debug kernel image by about 30%
- However, you don't need to load the debug version of the kernel
  - Load the non-debug version to the target, but use the debug version for the debugger/JTAG probe
- Save off the vmlinux and System.map files because these are used by the debugger or by you to find key addresses
  - The (b)zImage can be loaded on the target as normal

Enabling Debugging in the Kernel



KGDB Lash up

- KGDB supports debugging via the serial port
  - The gdb debugger is running on a second machine using the vmlinux you compiled with debugging symbols
  - You attach to the system being debugged using gdb's `target remote` command



Sources: dell.com, beagleboard.org, netgear.com

Once Everything is Connected

- Once everything is connected and the target remote command is entered on the debugger, you will reset the target with the new kernel
- The boot cycle will stop until the kernel detects gdb on the other side
  - Once connected, you can set breakpoints in the kernel and tell it to continue

Adding Device Driver Symbols

- Statically linked driver symbols are already built into the kernel's symbol table
  - Simply set break points on the driver methods themselves
- Dynamically loaded drivers require additional steps
  - Load the driver as normal
    - We need to find the addresses used by the driver

Debugging Loadable Modules

- In order to debug a loaded module, we need to tell the debugger where the module is in memory
  - The module's information is not in the kernel image because that shows only statically-linked drivers
- This information can typically be found in **`/proc/modules`** or **`/sys/module/<modulename>/sections/.text`**
- We then use the **`add-symbol-file`** gdb command to add the debug symbols for the driver at the address for the loaded module
  - **`(gdb)add-symbol-file ./mydriver.o 0x<addr>`**
- How we proceed depends on where we need to debug

Debugging Loadable Modules #2

- If we need to debug the **`__init`** code, we need to set a breakpoint in the **`load_module()`** function
- We'll need to breakpoint just before the control is transferred to the module's **`__init`**
  - Somewhere around line 3438 of module.c (4.14.8 kernel):

```
/* Start the module */
if (mod->init != NULL)
        ret = do_one_initcall(mod->init);
```

- Once the breakpoint is encountered, we can walk the module address list to find the assigned address for the module

Adding Additional Breakpoints

- Once you've added the module's symbols, you can set breakpoints at the various entry points of the driver

```
(gdb) b mydriver_read
```

- Other good breakpoint locations include:
  - `sys_sync`
  - `panic`
  - `oops_enter`
- When you hit the breakpoint, the debugger will drop to the source code and you can start single stepping code

Summary

- Wow, that was a lot!
- But, at this point, you should be pretty well aware of how to use gdb for both native and cross debugging
- Just remember, if you don’t put the bugs in there to begin with, you don’t have to work so hard to get them out ☺