THE LINUX FOUNDATION OPEN SOURCE SUMMIT NORTH AMERICA | Embedded Linux Conference North America

# Debugging and Profiling Linux Applications

**Mike Anderson**
Director of Technology
The PTR Group, LLC

THE LINUX FOUNDATION

# Speaker/Author Details

- **Website:**
  - [http://www.theptrgroup.com](http://www.theptrgroup.com)
- **Email:**
  - [mailto:mike@theptrgroup.com](mailto:mike@theptrgroup.com)
- **Linked-in:**
  - [https://www.linkedin.com/in/mikeandersonptr](https://www.linkedin.com/in/mikeandersonptr)
- **Twitter:**
  - **@hungjar**
- **PTR is now a subsidiary of** Huntington Ingalls Industries

# What We Will Talk About…

- The GNU Project, GCC and gdb
- 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 and more
- gdbserver and its options
- strace/ltrace
- gprof/gcov
- Valgrind
- LTTng and Ftrace

## Debugging Tool Classes

- We can think of debugging tools as falling into one of two classes:
  - Debuggers focused on determining what the code actually did or what it does
    - E.g., gdb, LLDB, rr, UndoDB, Live Recorder, etc.
  - Checkers that try to catch a particular bad thing such as a buffer overrun
    - Could be static or dynamic
      - E.g., Valgrind, Address Sanitizer, Coverity, etc.

# The GNU Project

- The ostensible goal of the GNU Project was to create a Un*x clone without any AT&T sources
  - GNU's Not Unix
- There are several projects developed that all targeted the original goal
  - First, there was the GNU C compiler (gcc) and later g++ finally manifesting itself as GCC – the Gnu Compiler Collection
    - Front ends for C, C++, Objective-C, FORTRAN, Ada and Go with associated libraries like libstdc++, etc.
  - Architected as a front-end language parser and a back-end code generator
  - Also added numerous *binutils* such as the linker, librarian, etc.
- 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

# GDB Command Line 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
- And, the mac-daddy of options: **`-ggdb3`**
  - This is like **`-g3`**, but generates debugging information specifically for gdb rather than normal COFF/XCOFF or DWARF 2 of **`-g`**

# Example Compile for GDB

- Example compilation to enable debugging

```
$ arm-linux-gnueabi-gcc -ggdb3 -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
```

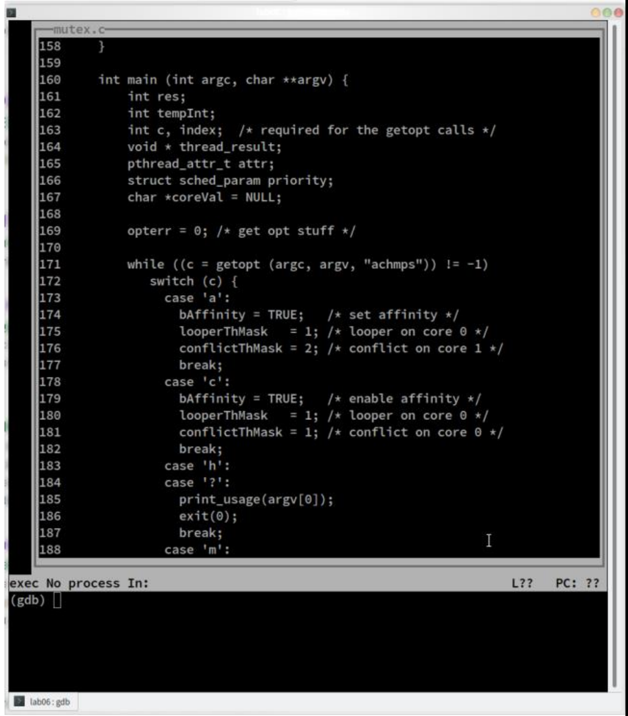
# Running GDB CLI

- 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 8.1-0ubuntu3) 8.1.0.20180409-git
Copyright (C) 2018 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

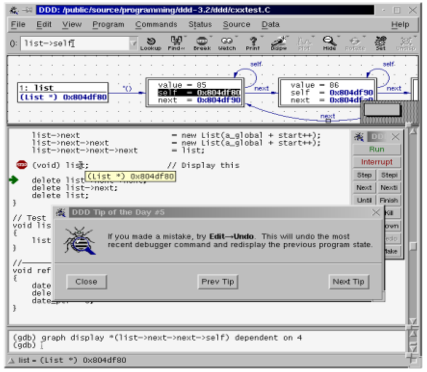
- TUI mode is a text-based user interface that separates out the program text from the gdb command line
  - Uses curses
- More clear cut than using the typical CLI, but maybe not as good as the GUIs for gdb like ddd
- You can start gdb in TUI mode using the --tui command line argument
- You can switch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



```
mutex.c
    }

    int main (int argc, char **argv) {
        int res;
        int tempInt;
        int c, index;  /* required for the getopt calls */
        void * thread_result;
        pthread_attr_t attr;
        struct sched_param priority;
        char *coreVal = NULL;

        opterr = 0; /* get opt stuff */

        while ((c = getopt (argc, argv, "achmps")) != -1)
           switch (c) {
             case 'a':
               bAffinity = TRUE;   /* set affinity */
               looperThMask   = 1; /* looper on core 0 */
               conflictThMask = 2; /* conflict on core 1 */
               break;
             case 'c':
               bAffinity = TRUE;   /* enable affinity */
               looperThMask   = 1; /* looper on core 0 */
               conflictThMask = 1; /* conflict on core 0 */
               break;
             case 'h':
             case '?':
               print_usage(argv[0]);
               exit(0);
               break;
             case 'm':
exec No process In:                                          L??   PC: ??
(gdb) 
```

# 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/
  - MS Visual Studio Code
    - Yes, it's OSS
    - GDB plug-in that enables graphical debug in IDE
    - https://code.visualstudio.com/download
- IDE support includes Eclipse, Kdevelop, Slickedit®, CodeWarrior®, Arriba® and more

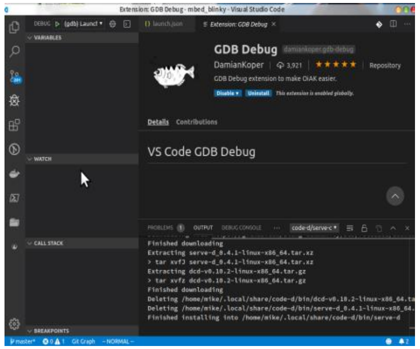


```
(): list->self

1: list
(List *) 0x804df80
value = 85
self  = 0x804df80
next  = 0x804df90
value = 86
self  = 0x804df90
next  = 0x804dfa0

    list->next                    = new List(a_global + start++);
    list->next->next              = new List(a_global + start++);
    list->next->next->next        = list;

    (void) list;                 // Display this
                (List *) 0x804df80
    delete list
    delete list->next;
    delete list;
}

// Test
void lis
{
    list
}

//
void ref
{
    date
    dele
    date
}
```

DDD Tip of the Day #5

If you made a mistake, try **Edit→Undo**. This will undo the most recent debugger command and redisplay the previous program state.

```
(gdb) graph display *(list->next->next->self) dependent on 4
(gdb)
list = (List *) 0x804df80
```



GDB Debug damiankoper.gdb-debug
DamianKoper | 3,921 | ★★★★★ | Repository
GDB Debug extension to make OiAK easier.
Disable ▾ Uninstall This extension is enabled globally.

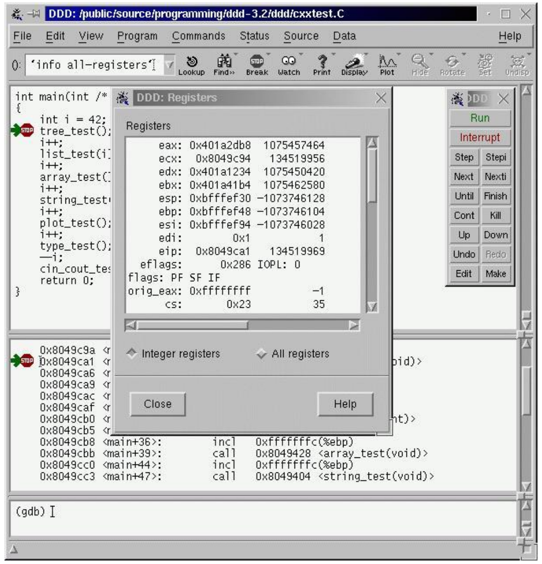
VS Code GDB Debug

```
Finished downloading
Extracting serve-d_0.4.1-linux-x86_64.tar.xz
> tar xvfJ serve-d_0.4.1-linux-x86_64.tar.xz
Extracting dcd-v0.10.2-linux-x86_64.tar.gz
> tar xvfz dcd-v0.10.2-linux-x86_64.tar.gz
Finished downloading
Deleting /home/mike/.local/share/code-d/bin/dcd-v0.10.2-linux-x86_64.ta
Deleting /home/mike/.local/share/code-d/bin/serve-d_0.4.1-linux-x86_64.
Finished installing into /home/mike/.local/share/code-d/bin/serve-d
```

# 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
```


DDD: /public/source/programming/ddd-3.2/ddd/cxxtest.C
File Edit View Program Commands Status Source Data Help
Lookup Find» Break Watch Print Display Plot Hide Rotate Set Undisp
DDD: Registers
Registers
Integer registers
All registers
Close
Help
Run
Interrupt
Step Stepi
Next Nexti
Until Finish
Cont Kill
Up Down
Undo Redo
Edit Make

```
(): 'info all-registers'
```

```
int main(int /*
{
    int i = 42;
    tree_test();
    i++;
    list_test(i
    i++;
    array_test(
    i++;
    string_test
    i++;
    plot_test();
    i++;
    type_test();
    --i;
    cin_cout_tes
    return 0;
}
```

```
      eax: 0x401a2db8  1075457464
      ecx:  0x8049c94   134519956
      edx: 0x401a1234  1075450420
      ebx: 0x401a41b4  1075462580
      esp: 0xbfffef30 -1073746128
      ebp: 0xbfffef48 -1073746104
      esi: 0xbfffef94 -1073746028
      edi:        0x1           1
      eip:  0x8049ca1   134519969
   eflags:      0x286 IOPL: 0
flags: PF SF IF
orig_eax: 0xffffffff          -1
       cs:       0x23          35
```

```
0x8049c9a <r
0x8049ca1 <r                                   oid)>
0x8049ca6 <r
0x8049ca9 <r
0x8049cac <r
0x8049caf <r
0x8049cb0 <r                                   nt)>
0x8049cb5 <r
0x8049cb8 <main+36>:      incl   0xfffffffc(%ebp)
0x8049cbb <main+39>:      call   0x8049428 <array_test(void)>
0x8049cc0 <main+44>:      incl   0xfffffffc(%ebp)
0x8049cc3 <main+47>:      call   0x8049404 <string_test(void)>
```

```
(gdb)
```

# 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:

```
set history save on
set print pretty on
set pagination off
set confirm off
```

- The `-x` command line option to gdb also allows for running scripts at gdb load time
- You can also define keyboard macros that launch python scripts or shell off to the OS using `shell <cmd>`

# Embedded Python Engine

- `gdb` now has a tightly integrated Python engine!
  - It doesn't just shell out
- Execute single Python expressions by prefacing the expression with "python"
  - Go into interactive mode by typing "python" and then remember to type "end" as the last line
- `(gdb) python gdb.execute()` to do gdb commands
- `(gdb) python gdb.parse_and_eval()` to get data from the inferior
- `(gdb) python help('gdb')` to see Python gdb package

# Working with Signals via gdb

- gdb is built on top of ptrace
  - When an inferior gets a signal, the inferior is suspended and the tracer gets notified
- 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
```

## 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

## 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

## 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 <value>`**
- You can change the options using the **`set`** command
  - E.g., **`set output-radix 16`** would set the display radix to hexidecimal
  - Use **`show`** to see the available options
  - **`help set <option>`** to get help on the different options

## 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.

## 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>`**
- You can also set conditional breakpoints (beware of the overhead)

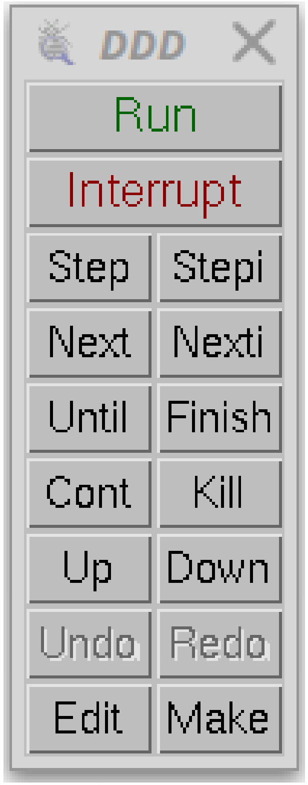
```
(gdb) break <lineno> if <condition is true>
```

- 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



## Watchpoints

- Force a program break when a selected variable's value changes
- Examples:

| | |
|---|---|
| `watch -l <address/symbol>` | command is scope aware |
| `rwatch <a/s>` | stops if the address is read |
| `watch <a/s> thread 3` | stops if thread 3 modifies |
| `watch <a/s> if <a/s> > 5` | stops when contents > 5 |

- 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
```

# 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

- Show backtraces for all threads:

```
(gdb) thread apply all bt
```

- Restrict thread execution to current thread

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

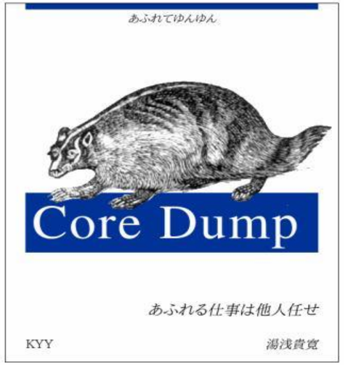
Source: optusnet.com.au

## Generating “Core Dumps”

- When an application terminates abnormally, a core file can be generated
  - core file - (n.) A file created when a program malfunctions 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

# Using the Core File

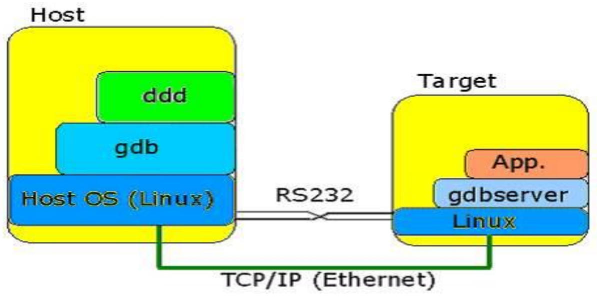
- Once you have a core file, you can use gdb to try to determine what went wrong
- Load the core file using:

```
$ 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

# 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

# gdb/gdbserver Cross Debug Example

- 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
    ```

# strace/ltrace

- Using tools like atsar, top, gkrellm, etc. will give you an idea as to what your system is doing globally. You can now focus on system call and library call tracing
  - `strace` and `ltrace`
- These require no special compilation flags
  - They can be attached to a running application at any time
  - They support time tagging for crude performance monitoring

# Using strace to Watch System Calls

- When debugging what appears to be a kernel-space error, it can be helpful to watch the system calls that are made from user-space
  - See what events lead to the error
- strace displays all system calls made by a program
  - Can display timestamp information per system call as well

STRACE
Give a newbie a tool that spits thousands of incomprehensible lines of output, and he'll think of himself as a great hacker
DIY.DESPAIR.COM

Source: diy.despair.com

# Example strace Output

```
/ # strace ls /dev/labdev
execve("/bin/ls", ["ls", "/dev/labdev"], [/* 8 vars */]) = 0
fcntl64(0, F_GETFD)                       = 0
fcntl64(1, F_GETFD)                       = 0
fcntl64(2, F_GETFD)                       = 0
geteuid()                                 = 0
getuid()                                  = 0
getegid()                                 = 0
getgid()                                  = 0
brk(0)                                    = 0x1028ad68
brk(0x1028bd68)                           = 0x1028bd68
brk(0x1028c000)                           = 0x1028c000
ioctl(1, TIOCGWINSZ or TIOCGWINSZ, {ws_row=0, ws_col=0, ws_xpixel=0, ws_ypixel=0
ioctl(1, TCGETS or TCGETS, {B9600 opost isig icanon echo ...}) = 0
ioctl(1, TCGETS or TCGETS, {B9600 opost isig icanon echo ...}) = 0
lstat("/dev/labdev", {st_mode=S_IFCHR|0644, st_rdev=makedev(254, 0), ...}) = 0
open("/etc/localtime", O_RDONLY)          = -1 ENOENT (No such file or directory)
lstat("/dev/labdev", {st_mode=S_IFCHR|0644, st_rdev=makedev(254, 0), ...}) = 0
fstat64(1, {st_mode=S_IFCHR|0600, st_rdev=makedev(4, 64), ...}) = 0
ioctl(1, TCGETS or TCGETS, {B9600 opost isig icanon echo ...}) = 0
mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x300
write(1, "\33[1;35m/dev/labdev\33[0m\n", 23/dev/labdev) = 23
munmap(0x30000000, 4096)                  = 0
Exit(0)                                   = ?
```

# strace – Where is time being spent?

- Use **-c** option while invoking the app
- Example – v4l2-based application

```
$ strace -c ./capture_stream -D /dev/video0 -w 640*480 -p 1|./viewer -w 640*480 -p 1
% time     seconds  usecs/call     calls    errors syscall
------ ----------- ----------- --------- --------- ----------------
 86.40    0.054382         365       149           write
  9.53    0.005999          41       148           select
  3.85    0.002425           8       310         2 ioctl
  0.21    0.000133          10        13           mmap
  0.00    0.000000           0         1           read
  0.00    0.000000           0         3           open
  0.00    0.000000           0         2           close
  0.00    0.000000           0         1           stat
  0.00    0.000000           0         3           fstat
  0.00    0.000000           0         4           mprotect
  0.00    0.000000           0         1           munmap
  0.00    0.000000           0         3           brk
  0.00    0.000000           0         3         3 access
  0.00    0.000000           0         1           execve
  0.00    0.000000           0         1           arch_prctl
------ ----------- ----------- --------- --------- ----------------
100.00    0.062939                   643         5 total
```

- Most of the time is consumed in the write call

# ltrace

- Just as strace allows us to trace system calls, ltrace allows us to trace library calls
  - You do not have to have the source nor compile the library for debugging to do this
- Can be used to trace glibc interaction
  - But be prepared to get a *lot* of output
- Command-line options exist to limit which libraries you are interested in tracing
  - Other options are similar to strace
- The output looks like strace

# Procfs

- Procfs is a representation of the kernel’s information about each process/thread and about the system itself
- Each process ID has its own directory with information about memory usage, processor affinity, child threads and more
- Many system tuning parameters for the protocol stack and interrupts are here as well

## What is Valgrind?

- Valgrind is an instrumentation framework for building dynamic analysis tools
  - Built to be user extensible
- It is widely used by Linux developers
- Valgrind tools can automatically detect potential memory management and threading problems
- Has tools for providing profiling data
- Mainly supports C and C++ programs
- Licensed under GPL v2
- Documentation is at http://valgrind.org/doc

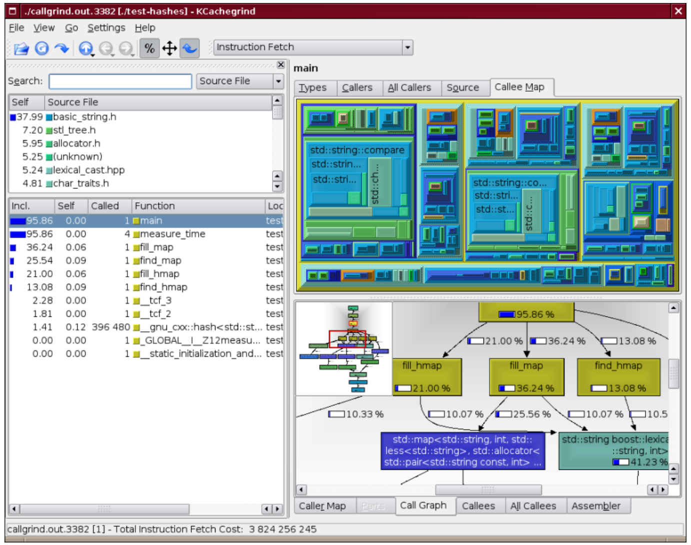
Source: valgrind.org

# Valgrind

- Collection of user-space analysis tools
- Embedded options are now available for PPC and ARM
- Traditionally built and run under x86 for analysis then transfer to target
  - Memcheck – memory leaks etc.
  - Cachegrind – cache and locality of reference
  - Callgrind – caller/callee relationships
  - Massif – heap profiler
  - Helgrind – POSIX threads helper
  - DRD – Multi threaded C/C++ detector

```
$ valgrind -vgdb=full --vgdb-error=0 ./prog
```

  - Runs gdb server inside of Valgrind that you can connect remotely to



| Self | Source File |
|---|---|
| 37.99 | basic_string.h |
| 7.20 | stl_tree.h |
| 5.95 | allocator.h |
| 5.25 | (unknown) |
| 5.24 | lexical_cast.hpp |
| 4.81 | char_traits.h |

| Incl. | Self | Called | Function | Loc |
|---|---|---|---|---|
| 95.86 | 0.00 | 1 | main | test |
| 95.86 | 0.00 | 4 | measure_time | test |
| 36.24 | 0.06 | 1 | fill_map | test |
| 25.54 | 0.09 | 1 | find_map | test |
| 21.00 | 0.06 | 1 | fill_hmap | test |
| 13.08 | 0.09 | 1 | find_hmap | test |
| 2.28 | 0.00 | 1 | __tcf_3 | test |
| 1.81 | 0.00 | 1 | __tcf_2 | test |
| 1.41 | 0.12 | 396 480 | __gnu_cxx::hash<std::st... | test |
| 0.00 | 0.00 | 1 | _GLOBAL__I__Z12measu... | test |
| 0.00 | 0.00 | 1 | __static_initialization_and... | test |

callgrind.out.3382 [1] - Total Instruction Fetch Cost: 3 824 256 245

Source: alexott.net

# Zooming in on User Space: gprof

- gprof is included as part of the GNU utilities
  - The GNU compiler instruments your code to collect execution information
- Compile the code with the -pg compiler flag
  - Then run the application to collect the information
    - Make sure that the application exits properly
- Compile the code with the -pg compiler flag and run it

```
$ gcc -o myprog myprog.c utils.c -g -pg
```

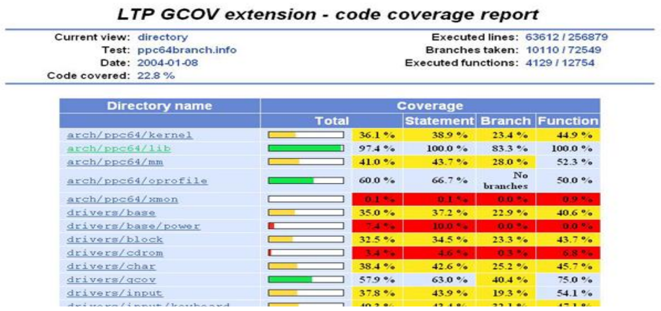
- Then run gprof:

```
$ gprof options [executable file [profile data file]]
```

  - The options control the type of output that gprof produces, possible symbols to ignore, etc.
- The resulting gmon.out file contains the profiling information
  - Viewable as a flat output, call graph or annotated source

# Profiling Via Code Coverage

- Another possibility is that your code has different use cases
  - Being able to tell what code actually ran can help zero in on specific test cases
- To help with this, the GNU tools have gcov
  - The GNU code coverage tool
    - Requires the -fprofile-arcs –ftest-coverage compiler options
  - Produces an output file with the source annotated with how many times each line of code was executed



**LTP GCOV extension - code coverage report**

| | | | |
|---|---|---|---|
| Current view: | directory | Executed lines: | 63612 / 256879 |
| Test: | ppc64branch.info | Branches taken: | 10110 / 72549 |
| Date: | 2004-01-08 | Executed functions: | 4129 / 12754 |
| Code covered: | 22.8 % | | |

| Directory name | Coverage | | | |
|---|---|---|---|---|
| | Total | Statement | Branch | Function |
| arch/ppc64/kernel | 36.1 % | 38.9 % | 23.4 % | 44.9 % |
| arch/ppc64/lib | 97.4 % | 100.0 % | 83.3 % | 100.0 % |
| arch/ppc64/mm | 41.0 % | 43.7 % | 28.0 % | 52.3 % |
| arch/ppc64/oprofile | 60.0 % | 66.7 % | No branches | 50.0 % |
| arch/ppc64/xmon | 0.1 % | 0.1 % | 0.0 % | 0.9 % |
| drivers/base | 35.0 % | 37.2 % | 22.9 % | 40.6 % |
| drivers/base/power | 7.4 % | 10.0 % | 0.0 % | 0.0 % |
| drivers/block | 32.5 % | 34.5 % | 23.3 % | 43.7 % |
| drivers/cdrom | 3.4 % | 4.6 % | 0.3 % | 6.8 % |
| drivers/char | 38.4 % | 42.6 % | 25.2 % | 45.7 % |
| drivers/gcov | 57.9 % | 63.0 % | 40.4 % | 75.0 % |
| drivers/input | 37.8 % | 43.9 % | 19.3 % | 54.1 % |
| drivers/input/keyboard | 40.3 % | 43.4 % | 32.1 % | 47.1 % |

Source: ibm.com

# Example gcov Execution

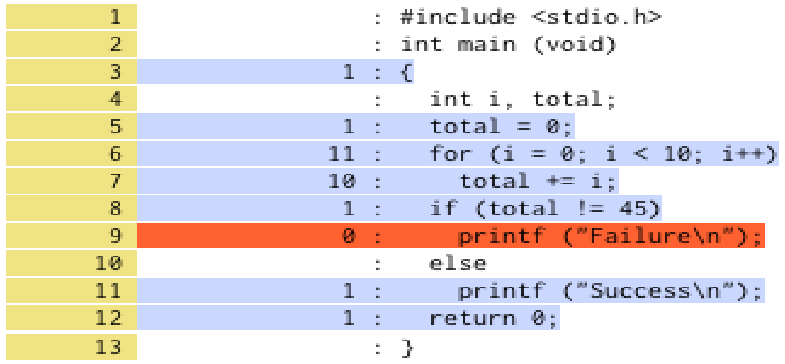
```
$ gcc -fprofile-arcs -ftest-coverage tmp.c -o tmp
              $ ./tmp
              $ gcov tmp.c
              90.00% of 10 source lines executed in file tmp.c
              Creating tmp.c.gcov.

              -:    6:
              1:    7:  total = 0;
              -:    8:
             11:    9:  for (i = 0; i < 10; i++)
             10:   10:    total += i;
              -:   11:
              1:   12:  if (total != 45)
          #####:   13:    printf ("Failure\n");
              -:   14:  else
              1:   15:    printf ("Success\n");
              1:   16:  return 0;
              1:   17:}
```

**The number on the left indicates the number of times that line was executed**

**The ##### indicates a line that was not executed**

# Branch Annotated Source

```
The annotated source code would then look like:

        -:    0:Source:tmp.c
        -:    0:Object:tmp.bb
        -:    1:#include <stdio.h>
        -:    2:
        -:    3:int main (void)
        1:    4:{
        1:    5:  int i, total;
        -:    6:
        1:    7:  total = 0;
        -:    8:
       11:    9:  for (i = 0; i < 10; i++)
branch  0: taken 90%
branch  1: taken 100%
branch  2: taken 100%
       10:   10:    total += i;
        -:   11:
        1:   12:  if (total != 45)
branch  0: taken 100%
    #####:   13:    printf ("Failure\n");
call    0: never executed
branch  1: never executed
        -:   14:  else
        1:   15:    printf ("Success\n");
call    0: returns 100%
        1:   16:  return 0;
        1:   17:}
```



```
 1          : #include <stdio.h>
 2          : int main (void)
 3        1 : {
 4          :   int i, total;
 5        1 :   total = 0;
 6       11 :   for (i = 0; i < 10; i++)
 7       10 :     total += i;
 8        1 :   if (total != 45)
 9        0 :     printf ("Failure\n");
10          :   else
11        1 :     printf ("Success\n");
12        1 :   return 0;
13          : }
```

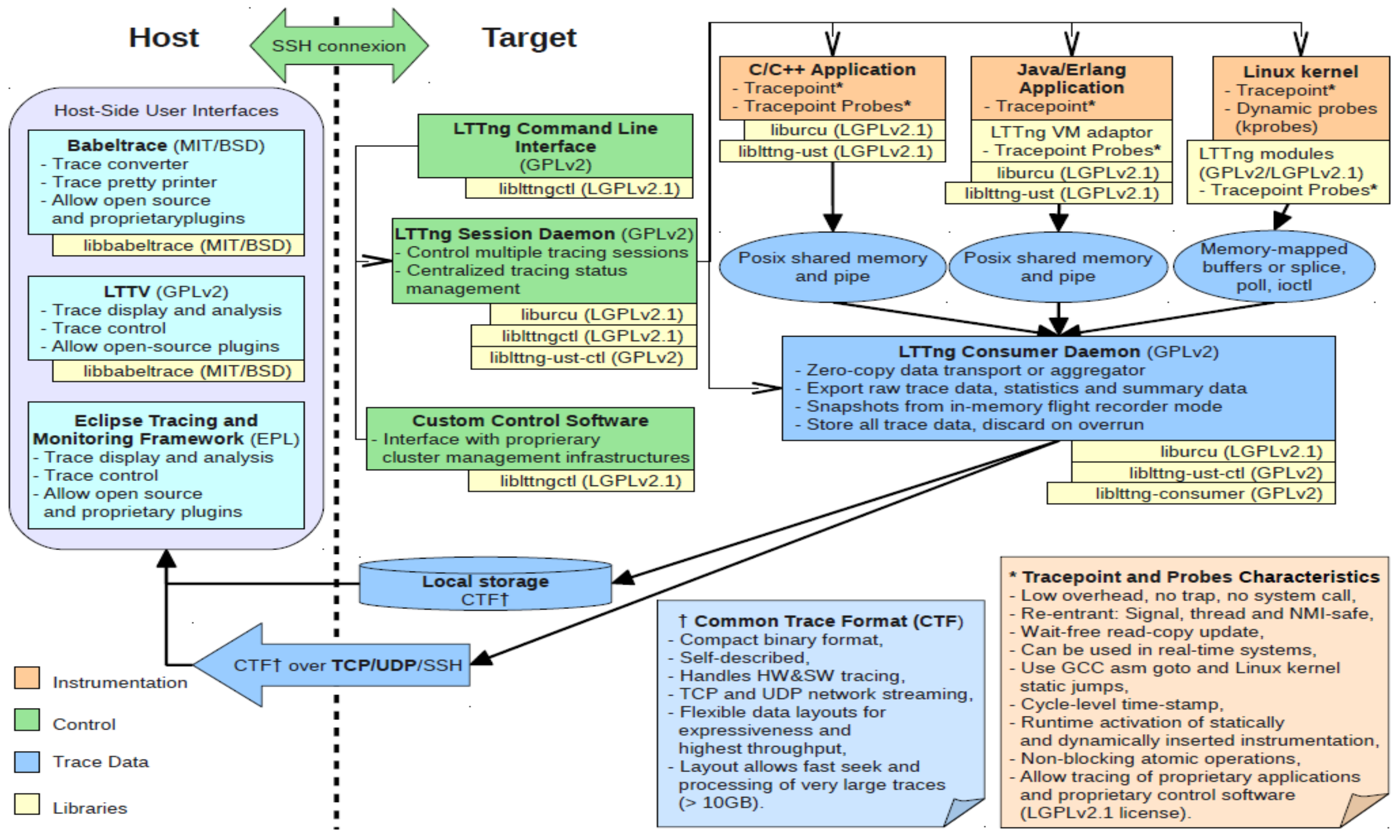
Source: postography.com

# LTTng 2.x

- "Linux Trace Toolkit – next generation"
  - Traces both user and kernel events
    - Can be used as a fast strace replacement
  - Gathers from multiple sources
    - Tracepoint, kprobes, perf monitors for kernel
    - Supports instrumentation in user and kernel space
  - Outputs to a common format
    - Supports the "Common Trace Format" (CTF)
    - Multiple viewers will be able to read output
      - Babeltrace, Eclipse LTTng plugin (currently available)
      - LTTV Viewer, (work in progress)

Source: lttng.org

# LTTng Tracer Architecture


Host
SSH connexion
Target
Host-Side User Interfaces
Babeltrace (MIT/BSD)
- Trace converter
- Trace pretty printer
- Allow open source and proprietaryplugins
libbabeltrace (MIT/BSD)
LTTV (GPLv2)
- Trace display and analysis
- Trace control
- Allow open-source plugins
libbabeltrace (MIT/BSD)
Eclipse Tracing and Monitoring Framework (EPL)
- Trace display and analysis
- Trace control
- Allow open source and proprietary plugins
LTTng Command Line Interface (GPLv2)
liblttngctl (LGPLv2.1)
LTTng Session Daemon (GPLv2)
- Control multiple tracing sessions
- Centralized tracing status management
liburcu (LGPLv2.1)
liblttngctl (LGPLv2.1)
liblttng-ust-ctl (GPLv2)
Custom Control Software
- Interface with proprieraty cluster management infrastructures
liblttngctl (LGPLv2.1)
C/C++ Application
- Tracepoint*
- Tracepoint Probes*
liburcu (LGPLv2.1)
liblttng-ust (LGPLv2.1)
Java/Erlang Application
- Tracepoint*
LTTng VM adaptor
- Tracepoint Probes*
liburcu (LGPLv2.1)
liblttng-ust (LGPLv2.1)
Linux kernel
- Tracepoint*
- Dynamic probes (kprobes)
LTTng modules (GPLv2/LGPLv2.1)
- Tracepoint Probes*
Posix shared memory and pipe
Posix shared memory and pipe
Memory-mapped buffers or splice, poll, ioctl
LTTng Consumer Daemon (GPLv2)
- Zero-copy data transport or aggregator
- Export raw trace data, statistics and summary data
- Snapshots from in-memory flight recorder mode
- Store all trace data, discard on overrun
liburcu (LGPLv2.1)
liblttng-ust-ctl (GPLv2)
liblttng-consumer (GPLv2)
Local storage CTF†
CTF† over TCP/UDP/SSH
Instrumentation
Control
Trace Data
Libraries

**† Common Trace Format (CTF)**
- Compact binary format,
- Self-described,
- Handles HW&SW tracing,
- TCP and UDP network streaming,
- Flexible data layouts for expressiveness and highest throughput,
- Layout allows fast seek and processing of very large traces (> 10GB).

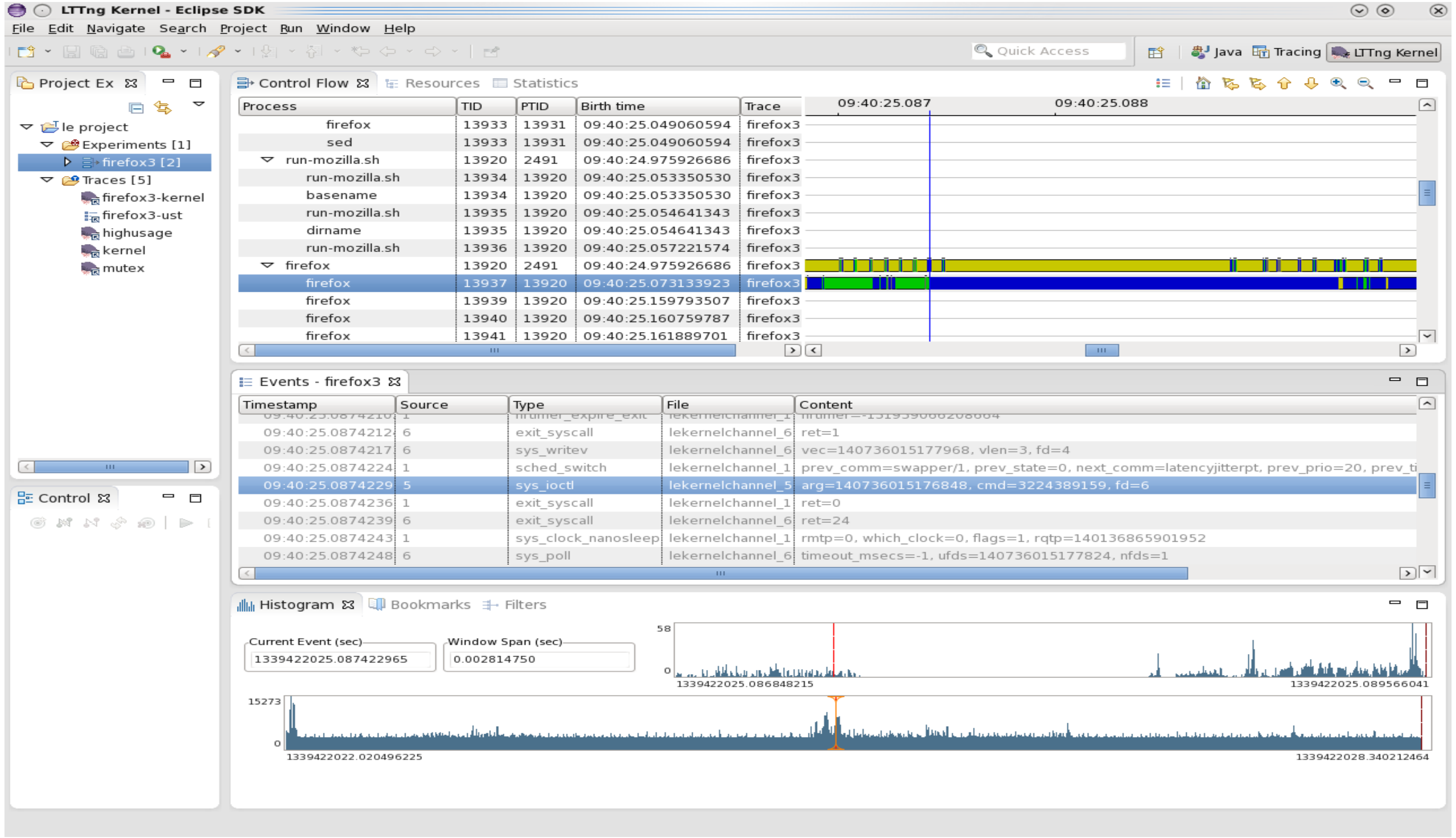
*** Tracepoint and Probes Characteristics**
- Low overhead, no trap, no system call,
- Re-entrant: Signal, thread and NMI-safe,
- Wait-free read-copy update,
- Can be used in real-time systems,
- Use GCC asm goto and Linux kernel static jumps,
- Cycle-level time-stamp,
- Runtime activation of statically and dynamically inserted instrumentation,
- Non-blocking atomic operations,
- Allow tracing of proprietary applications and proprietary control software (LGPLv2.1 license).

Source: lttng.org

## LTTng-UST

- LTTng-UST (User Space Tracer) designed to provide detailed information about user space activity
- Port of the LTTng kernel tracer to user space
- Tracing does not require system calls or traps
- LTTng-UST instrumentation points can be added in applications including signal handlers and libraries

# LTTng Eclipse Plug-in



| Process | TID | PTID | Birth time | Trace |
|---|---|---|---|---|
| firefox | 13933 | 13931 | 09:40:25.049060594 | firefox3 |
| sed | 13933 | 13931 | 09:40:25.049060594 | firefox3 |
| run-mozilla.sh | 13920 | 2491 | 09:40:24.975926686 | firefox3 |
| run-mozilla.sh | 13934 | 13920 | 09:40:25.053350530 | firefox3 |
| basename | 13934 | 13920 | 09:40:25.053350530 | firefox3 |
| run-mozilla.sh | 13935 | 13920 | 09:40:25.054641343 | firefox3 |
| dirname | 13935 | 13920 | 09:40:25.054641343 | firefox3 |
| run-mozilla.sh | 13936 | 13920 | 09:40:25.057221574 | firefox3 |
| firefox | 13920 | 2491 | 09:40:24.975926686 | firefox3 |
| firefox | 13937 | 13920 | 09:40:25.073133923 | firefox3 |
| firefox | 13939 | 13920 | 09:40:25.159793507 | firefox3 |
| firefox | 13940 | 13920 | 09:40:25.160759787 | firefox3 |
| firefox | 13941 | 13920 | 09:40:25.161889701 | firefox3 |

| Timestamp | Source | Type | File | Content |
|---|---|---|---|---|
| 09:40:25.0874212 | 6 | exit_syscall | lekernelchannel_6 | ret=1 |
| 09:40:25.0874217 | 6 | sys_writev | lekernelchannel_6 | vec=140736015177968, vlen=3, fd=4 |
| 09:40:25.0874224 | 1 | sched_switch | lekernelchannel_1 | prev_comm=swapper/1, prev_state=0, next_comm=latencyjitterpt, prev_prio=20, prev_ti |
| 09:40:25.0874229 | 5 | sys_ioctl | lekernelchannel_5 | arg=140736015176848, cmd=3224389159, fd=6 |
| 09:40:25.0874236 | 1 | exit_syscall | lekernelchannel_1 | ret=0 |
| 09:40:25.0874239 | 6 | exit_syscall | lekernelchannel_6 | ret=24 |
| 09:40:25.0874243 | 1 | sys_clock_nanosleep | lekernelchannel_1 | rmtp=0, which_clock=0, flags=1, rqtp=140136865901952 |
| 09:40:25.0874248 | 6 | sys_poll | lekernelchannel_6 | timeout_msecs=-1, ufds=140736015177824, nfds=1 |

Current Event (sec): 1339422025.087422965
Window Span (sec): 0.002814750

# OProfile

- OProfile is a kernel-based, statistical profiler
  - Samples instruction pointer to figure out what's running
    - Records current symbol and owner task
    - Can also track cache misses
  - Must be enabled in the kernel and rebuilt
- It's useful for seeing where in the kernel your applications are spending their time
  - There is a collection of userspace helpers that help control and display the OProfile data

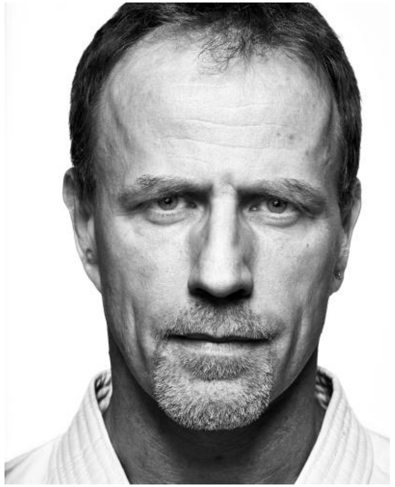
## OProfile Sample Output

- The OProfile data capture is made available via a /proc entry
  - The opreport command reads the data and creates a report

```
samples  %        app name                   symbol name
10469    34.0146  no-vmlinux                 (no symbols)
4358     14.1595  libxul.so                  (no symbols)
4306     13.9905  libmozjs.so                (no symbols)
3000      9.7472  libc-2.8.90.so             (no symbols)
1185      3.8502  libglib-2.0.so.0.1800.2    (no symbols)
895       2.9079  libflashplayer.so          (no symbols)
626       2.0339  libpixman-1.so.0.12.0      (no symbols)
588       1.9105  libnspr4.so.0d             (no symbols)
468       1.5206  bash                       (no symbols)
457       1.4848  libpthread-2.8.90.so       pthread_mutex_lock
454       1.4751  Xorg                       (no symbols)
401       1.3029  fglrx_dri.so               (no symbols)
```

## What is Ftrace?

- Ftrace (function tracer) was developed largely by Steven Rostedt and a few other key kernel developers
  - The primary focus was embedded Linux and the **`PREEMPT_RT`** real-time patches in the kernel
- Ftrace was introduced into the kernel in 2008 as a way of providing debugging info
  - Allowed for time stamps, calling hierarchies and more
    - Well beyond the typical **`printk()`**
- The documentation was actually in the kernel before the code!

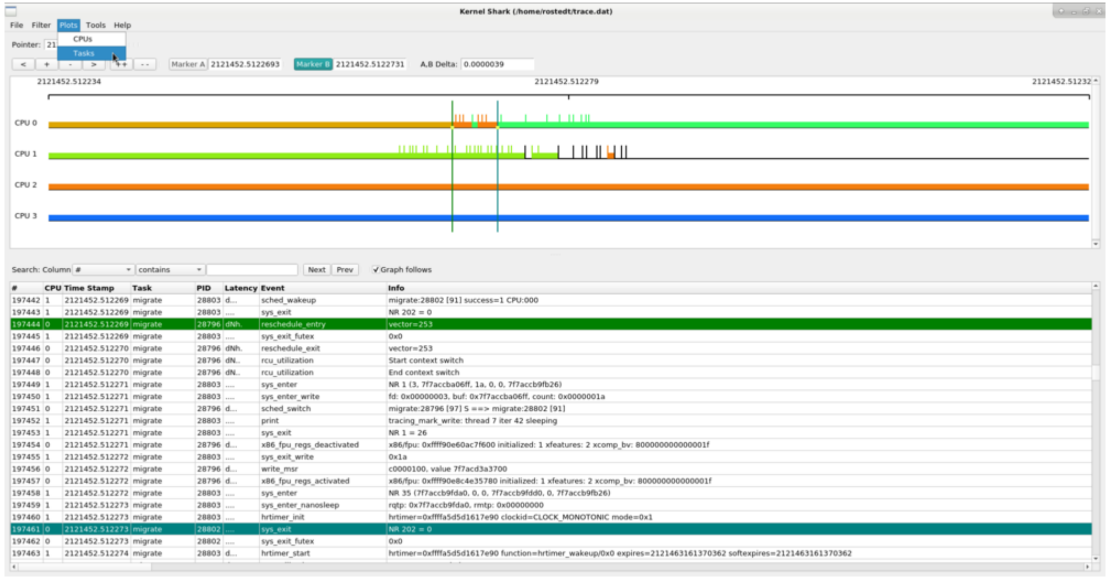

Source: facesofopensource.com

## What is Ftrace? #2

- `ftrace` provides tracing for internal kernel operations
  - Static tracepoints via event tracing for
    - interrupts, scheduling, file systems and more
    - Latencies for interrupts, preemption and combined
    - Process wakeup latencies with filtering for RT processes
  - Dynamic kernel function tracing
    - Tracing any function within kernel
    - Function filtering
    - Call graphs
    - Stack usage and frames
    - Many more tracepoints
- `ftrace` is a feature consolidated tool based or prior tracing facilities in the kernel
- Like OProfile, ftrace must be enabled in the kernel

# Graphing the Data via Kernelshark

- kernelshark is a GUI that reads and plots the data from ftrace
- Graphical output that is easy to include in reports
- Kernelshark v1.0 is now out!
  - Based on Qt instead of original GTK version



Kernel Shark (/home/rostedt/trace.dat)

Marker A 2121452.5122693  Marker B 2121452.5122731  A,B Delta: 0.0000039

2121452.512234  2121452.512279  2121452.51232

| # | CPU | Time Stamp | Task | PID | Latency | Event | Info |
|---|---|---|---|---|---|---|---|
| 197442 | 1 | 2121452.512269 | migrate | 28803 | d... | sched_wakeup | migrate:28802 [91] success=1 CPU:000 |
| 197443 | 1 | 2121452.512269 | migrate | 28803 | .... | sys_exit | NR 202 = 0 |
| 197444 | 0 | 2121452.512269 | migrate | 28796 | dNh. | reschedule_entry | vector=253 |
| 197445 | 1 | 2121452.512269 | migrate | 28803 | .... | sys_exit_futex | 0x0 |
| 197446 | 0 | 2121452.512270 | migrate | 28796 | dNh. | reschedule_exit | vector=253 |
| 197447 | 0 | 2121452.512270 | migrate | 28796 | dN.. | rcu_utilization | Start context switch |
| 197448 | 0 | 2121452.512270 | migrate | 28796 | dN.. | rcu_utilization | End context switch |
| 197449 | 1 | 2121452.512271 | migrate | 28803 | .... | sys_enter | NR 1 (3, 7f7accba06ff, 1a, 0, 0, 7f7accb9fb26) |
| 197450 | 1 | 2121452.512271 | migrate | 28803 | .... | sys_enter_write | fd: 0x00000003, buf: 0x7f7accba06ff, count: 0x0000001a |
| 197451 | 0 | 2121452.512271 | migrate | 28796 | d... | sched_switch | migrate:28796 [97] S ==> migrate:28802 [91] |
| 197452 | 1 | 2121452.512271 | migrate | 28803 | .... | print | tracing_mark_write: thread 7 iter 42 sleeping |
| 197453 | 1 | 2121452.512271 | migrate | 28803 | .... | sys_exit | NR 1 = 26 |
| 197454 | 0 | 2121452.512271 | migrate | 28796 | d... | x86_fpu_regs_deactivated | x86/fpu: 0xffff90e60ac7f600 initialized: 1 xfeatures: 2 xcomp_bv: 800000000000001f |
| 197455 | 1 | 2121452.512272 | migrate | 28803 | .... | sys_exit_write | 0x1a |
| 197456 | 0 | 2121452.512272 | migrate | 28796 | d... | write_msr | c0000100, value 7f7acd3a3700 |
| 197457 | 0 | 2121452.512272 | migrate | 28796 | d... | x86_fpu_regs_activated | x86/fpu: 0xffff90e8c4e35780 initialized: 1 xfeatures: 2 xcomp_bv: 800000000000001f |
| 197458 | 1 | 2121452.512272 | migrate | 28803 | .... | sys_enter | NR 35 (7f7accb9fda0, 0, 0, 7f7accb9fdd0, 0, 7f7accb9fb26) |
| 197459 | 1 | 2121452.512273 | migrate | 28803 | .... | sys_enter_nanosleep | rqtp: 0x7f7accb9fda0, rmtp: 0x00000000 |
| 197460 | 1 | 2121452.512273 | migrate | 28803 | .... | hrtimer_init | hrtimer=0xffffa5d5d1617e90 clockid=CLOCK_MONOTONIC mode=0x1 |
| 197461 | 0 | 2121452.512273 | migrate | 28802 | .... | sys_exit | NR 202 = 0 |
| 197462 | 0 | 2121452.512273 | migrate | 28802 | .... | sys_exit_futex | 0x0 |
| 197463 | 1 | 2121452.512274 | migrate | 28803 | d... | hrtimer_start | hrtimer=0xffffa5d5d1617e90 function=hrtimer_wakeup/0x0 expires=2121463161370362 softexpires=2121463161370362 |

Source: kernelshark.org

## Summary

- The problem with Linux is not that there aren't enough tools, but there are too many
  - Some overlap as well
- Debugging is at least mostly done with gdb
  - However, never underestimate the power of a lowly LED and an oscilloscope
- gprof/gcov and strace/ltrace provide additional insights
- Then, ftrace/LTTng can show additional interactions with kernel space for a system-wide view

Embedded Linux Conference

North America