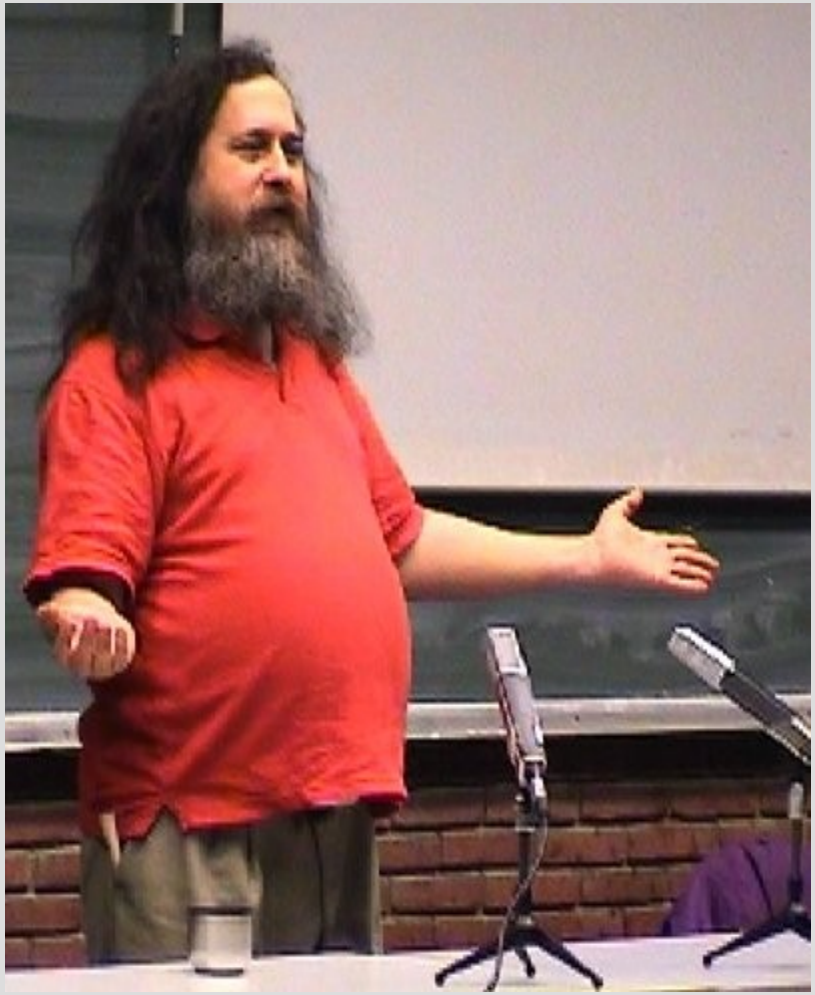
Tools for a New Generation

- Understanding how debuggers work boosts developers' efficiency
- Linux needs a **new generation** of tools
- D may lead the change

Current Tools Are ...



... stuck in the 80s, yet insist upon calling themselves a "GNU Generation"

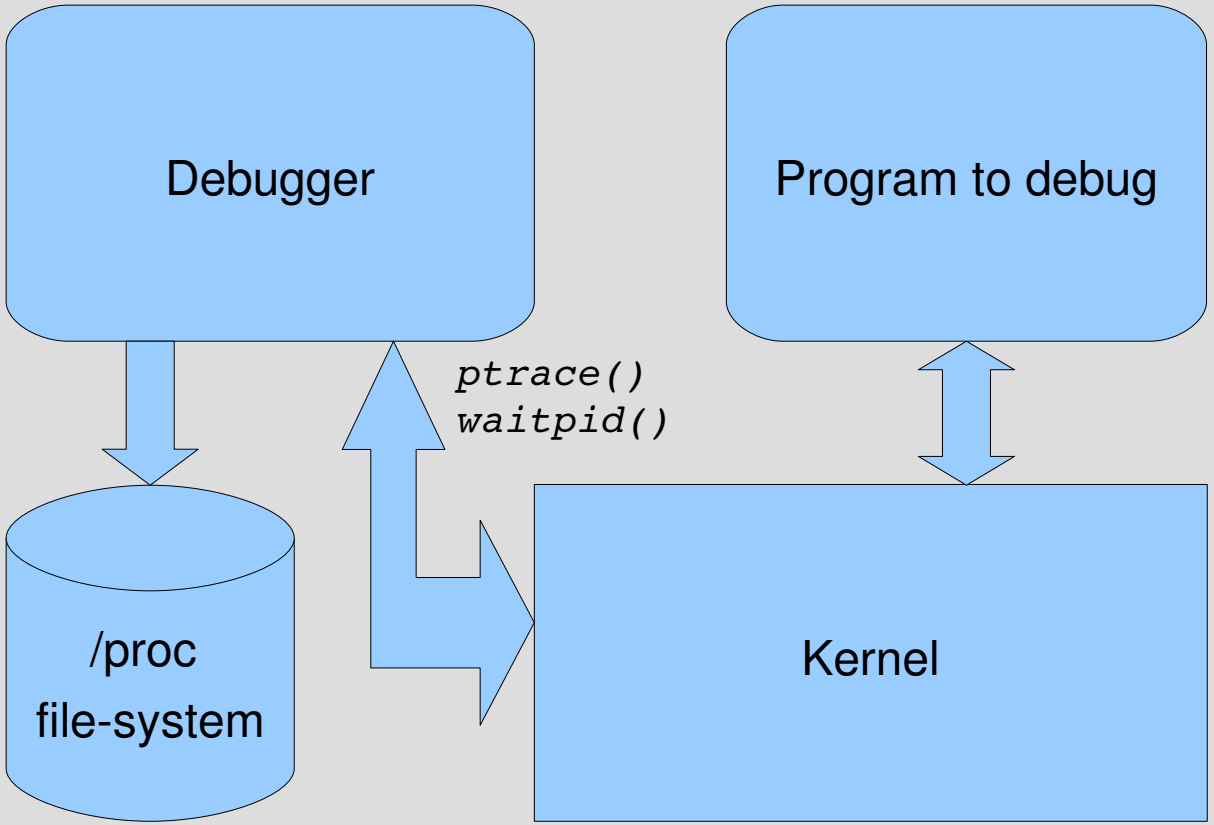
New designs should be modular, based on shared APIs.

GPL is **not** the only way!

Agenda

- Overview of debugger / program interaction
- Remote & cross-debugging: out of scope
- Fine-tunning the debug information
- Multi-threaded programs
- Limitations
- Advanced debugger usage
- Adding support for D Language
- Q and A

Debugger / Program Interaction



Program Interaction

- The debugger uses the ptrace() API to:
  - attach / detach to / from targets
  - read / write to / from debugged program memory
  - resume execution of program
- Debugger sends signals to program
  - most notably SIGSTOP
- The /proc file-system allows reading / writing of memory, and provides access to other information (command line args, etc)

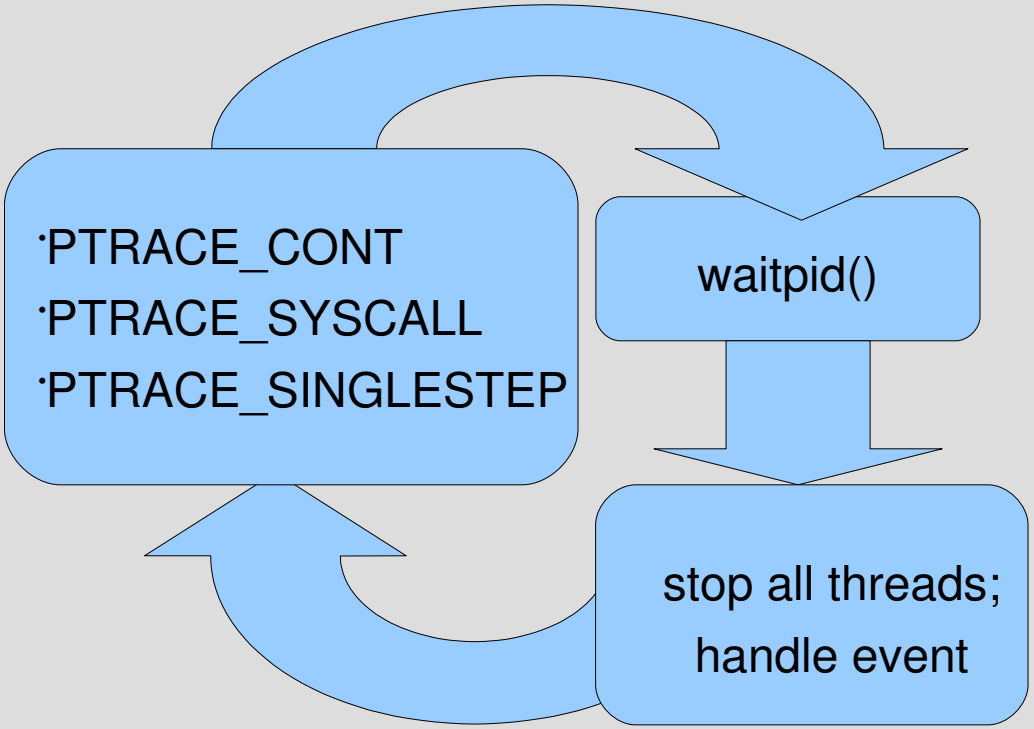
Worse Is Better

- "*It is more important for the implementation to be simpler than the interface*" Richard Gabriel, The Rise of Worse is Better
- (http://www.jwz.org/doc/worse-is-better.html)
- Example: when the debugger receives a SIGTRAP, it needs to figure what happened, on its own (could be a breakpoint, a singlestep, a system call, etc).

Debug Events

- When signals occur in the program the debugger is notified (before signal is delivered)
- The status of the debugged program is collected with the waitpid() call
- **The debugger is a big event loop**
- Example of events: SIGTRAP (breakpoint hit, or single-stepping), SIGSEGV, SIGSTOP

Main Event Loop



Resuming the Program

- PTRACE_CONT resumes program
- PTRACE_SINGLESTEP: executes one machine instruction
- PTRACE_SYSCALL: continue up until a system call occurs (this is what strace uses)

Handling Events

- Most common case: display the state of the program to user
  - call stack trace
  - variables
  - CPU registers
  - current source code location
- Execute a user command
- Allow user to insert breakpoints
- Allow user to evaluate expressions

Symbolic Information

- There are two sources of information:
  - the ELF symbol tables used by the linker
  - debug info stored in a special ELF section
- Each shared object and executable has one or two ELF symbol tables (static, dynamic)
- It is easy to do an address lookup to find the current function
- The maps file under /proc may be consulted to determine which symbol table to use

/proc/<pid>/maps

```
[cristiv@orcas:~]$ cat /proc/3454/maps
08048000-08061000 r-xp 00000000 08:01 568685                             /usr/bin/vmnet-dhcpd
08061000-08063000 rw-p 00019000 08:01 568685                             /usr/bin/vmnet-dhcpd
08063000-0808e000 rw-p 08063000 00:00 0                                  [heap]
f7db7000-f7dc0000 r-xp 00000000 08:01 90135                              /lib/libnss_files-2.4.so
f7dc0000-f7dc2000 rw-p 00008000 08:01 90135                              /lib/libnss_files-2.4.so
f7dc2000-f7dc3000 rw-p f7dc2000 00:00 0
f7dc3000-f7eea000 r-xp 00000000 08:01 90121                              /lib/libc-2.4.so
f7eea000-f7eeb000 r--p 00127000 08:01 90121                              /lib/libc-2.4.so
f7eeb000-f7eed000 rw-p 00128000 08:01 90121                              /lib/libc-2.4.so
f7eed000-f7ef0000 rw-p f7eed000 00:00 0
f7f0d000-f7f0f000 rw-p f7f0d000 00:00 0
f7f0f000-f7f27000 r-xp 00000000 08:01 90114                              /lib/ld-2.4.so
f7f27000-f7f28000 r--p 00017000 08:01 90114                              /lib/ld-2.4.so
f7f28000-f7f29000 rw-p 00018000 08:01 90114                              /lib/ld-2.4.so
ffce9000-ffcec000 rwxp ffce9000 00:00 0                                  [stack]
ffffe000-fffff000 r-xp ffffe000 00:00 0
[cristiv@orcas:~]$
`
```

Tracking Symbol Tables

- Symbol tables are read from ELF binary files (not from debugged program's memory, I.E. no debugging API is needed)
- The debugger needs to track the loading and unloading of dynamic library
- The dynamic loader provides a special location where the debugger needs to insert a breakpoint (see ***/usr/include/link.h***)

Debug Information

- Source file and line number
- Variables' names and data types
- Frame unwinding information
- The format is not specified by ELF
- The format is language-independent (kind of)
- Most common formats: DWARF, STAB
- The amount of debug info is proportional to the complexity of your code

Fine Tunning the Debug Info

- Prefer DWARF over STAB debug format
- STAB is linear by design, DWARF allows “accelerated” lookups
- In DWARF, there is one “handle” per shared object: modular programs are easier to debug
- Lookups work faster with smaller modules

Debugger Architectural Blocks

- Main Event Loop
- Thread Management (detects and attaches to new threads)
- Symbol Table Management
- Breakpoint Management
- Debug Info Readers
- Expression Evaluators (Interpreters)
- Scripting support, for automating tasks

Limitations and Challenges

- Stopping threads with SIGSTOP is kludgey
- In a multi-threaded debugger, only the thread that has initially attached to the target with ptrace(PTRACE_ATTACH) can subsequently use ptrace and /proc on the same target
- ptrace calls (other than PTRACE_ATTACH) fail when invoked with the ID of a running process

Multi-Threaded Programs

- The easiest way of handling debug events in multi-threaded programs is to stop all threads upon events
- Stopping all threads with SIGSTOP: kill the group or use tkill() where available
- The thread ID as returned by pthread_create is NOT the same as the task ID

Multi-Threaded, Part 2

- On Linux, the relationship between user-space threads and kernel threads is 1-to-1
- Not necessarily true on other UNIX derivatives
  - Example: In FreeBSD the mapping is dynamic
- ptrace() operates on task ID, not thread ID
- libthread_db.so provides thread information

Affinity Workaround

- In ZeroBUGS, the UI and the main debugger engine run on separate threads
- They each are a big event loop
- The UI needs to be responsive
- The UI thread **cannot call ptrace()**
- Calls need to be marshaled between threads

Debugging Memory

- Using custom breakpoints to track heap
- Break on malloc / calloc / realloc / free and record the addresses and sizes of block
- Very slow, because of **numerous context switches**
- Remember? Signals go through the kernel
- Valgrind works better for memory debugging

Advanced Usage

- Use the debugger to test a patch without rebuilding your code.
- Example: you suspect that an uninitialized variable is the cause of a bug
- Set a conditional breakpoint at the line where you want to initialize the variable, and make the condition something like **((x = -1) && false)**

Automatic Breakpoint Insertion

```
 1 import os
 2 import re
 3 import zero
 4
 5 def on_table_done(symTable):
 6     extension = re.compile('.d$')
 7
 8     process = symTable.process()
 9     module = symTable.module()
10
11     for unit in module.translation_units():
12         if unit.language() == zero.TranslationUnit.Language.D:
13             name = os.path.basename(unit.filename())
14
15             i = 0
16             while True:
17                 symName = "void " + extension.sub('.__unittest%d()' % i, name)
18                 matches = symTable.lookup(symName)
19                 if len(matches) == 0:
20                     break
21                 for sym in matches:
22                     process.set_breakpoint(sym.addr())
23                 i += 1
```

D Language Support

- Demangler, thanks to Thomas Kuhne
- Support for dynamic arrays
- (Some) support for associative arrays
  - repurposed DW_AT_containing_type to be the key type, and DW_AT_data_type gives the element type.
- Requires the ZERO_D_SUPPORT environment variable to be set

Q&A, Contact Info

- www.zerobugs.org
- www.zero-bugs.com
- Cristian Vlasceanu cristian@zerobugs.org