Kernel support for user debugging: ptrace, utrace, and what's next

Roland McGrath

redhat

Kernel support for user debugging

- What is ptrace?
- What is wrong with ptrace?
- What we do about it?
  - How do we support the next generation of tracing and debugging tools?
  - How do we get more hackers playing in this space?
  - New tracing API layer inside the kernel: utrace

redhat

What is ptrace?

- how one process traces & debugs others
  - used by all debugger applications (GDB, strace, etc.)
- from old BSD, repeated in Linux (interface 25+ years old)
  - ptrace() function interface, tweaked over the years
- ptrace facilities
  - stop on events
  - get/set user registers
  - read/write user memory (also /proc/pid/mem)
  - single-step/branch-step
  - h/w debug facilities

redhat

ptrace interface

- ptrace() function, <sys/ptrace.h>, <linux/ptrace.h>
- ~30 requests (0-2 args), 7 option bits
- always one thread at a time
- thread must be stopped (except PTRACE_ATTACH)
- once attached, debugger gets SIGCHLD, waitpid()
- event reports via pseudo-signal stop

redhat

What is wrong with ptrace? Userland perspective: interface

- changes behavior of traced processes
  - attach/detach interrupts system calls
  - overloads signals
- low throughput, high latency
- one tracer
- all-or-nothing security model
- no fun to program
  - clunky syscall interface
  - SIGCHLD/waitpid() difficult to use
    - too many races, corner cases
    - poor fit for application event loops
  - ad hoc arch requests

redhat

What is wrong with ptrace? Kernel perspective: implementation

- fragile kernel internals, poorly documented
  - task parent link, reparenting on exit/detach
  - waitpid() special cases
  - scattered magic checks
- arch code
  - poor separation of arch from generic
  - cut'n'paste maintenance

redhat

What do we do about it?

- clean up ptrace internals some
  - still maintaining it
- build new infrastructure inside the kernel
  - arch uniformity
  - layered approach, bottom up
  - not one-size-fits-all
    - well-specified tracing layer inside the kernel (utrace)
    - not just one new different user-level interface

redhat

arch internals cleanup

- ptrace arch cleanups (2.6.25, 2.6.26)
  - arch_ptrace
  - compat_arch_ptrace
- step (2.6.25)

```
#define arch_has_single_step()        (1)
#define arch_has_block_step()         (cpu_has_bt)
void user_enable_single_step(struct task_struct *task);
void user_enable_block_step(struct task_struct *task);
void user_disable_single_step(struct task_struct *task);
```

- asm/syscall.h (2.6.27)

redhat

user_regset (2.6.25)

- standardize formats: core ELF note type
- shared arch code for debug/core
- uniform interface for extension: NT_386_TLS, NT_PPC_*
- interface details: <linux/regset.h>
  - struct user_regset_view, task_user_regset_view()
    - e_machine, ..., n, regsets[]
  - struct user_regset
    - fields: n, size, core_note_type, ...
    - functions
      - get
      - set
      - active
      - writeback

redhat

<linux/tracehook.h> (2.6.27)

- well-specified calls from arch/core code
  - Kerneldoc comments, explain context (locking, etc.)
- core hooks
  - exec, clone, signals, exit, death, reap
- arch hooks
  - system call entry, exit
  - signal handler setup
- TIF_NOTIFY_RESUME
  - new arch support for noninvasive tracing

redhat

Architecture status

- 2.6.25: user_regset, step (x86, powerpc, ia64, sparc64)
- 2.6.27: powerpc, sparc64
- 2.6.28: x86, s390

redhat

utrace

- What is utrace?
  - in-kernel API (for kernel modules)
  - multiplexing layer (not just one new kind of tracing)
- What is utrace not?
  - ptrace replacement
  - new user-level interface
    - ptrace() is a user syscall; utrace is an in-kernel API
  - solution to “What's wrong with ptrace?”
- Then what is that good for?
  - platform for new solutions
  - can implement compatible ptrace() using it
  - means to build new interfaces + other new features

redhat

utrace goals

- Establish platform for new work
  - API for kernel modules
  - allows multiple separate uses: “tracing engines”
  - bottom layer, usable by non-gurus
    - block_device:fs :: utrace:tracing engine
    - net_device:net proto :: utrace:tracing engine
- Help you do it right
  - non-invasive (no interference with signals, wait, etc.)
  - low-overhead
  - arch-independent
  - maintain system invariants (SIGKILL)

redhat

utrace API concepts

- tracing engine = your code, calls into utrace API
- API calls are per-thread (aka task)
- asynchronous attach/detach
  - “attached engine” pointer is handle
- event callbacks (in traced thread)
- control
  - stop
  - resume, step, interrupt, report
  - detach
- report & quiesce: explicit synchronization via callbacks

redhat

utrace events

- SYSCALL_ENTRY, SYSCALL_EXIT
  - entry/exit distinguished, unlike ptrace
- SIGNAL
- SIGNAL_IGN, SIGNAL_STOP, SIGNAL_TERM, SIGNAL_CORE
  - signal disposition distinguished, unlike ptrace
- EXEC
- CLONE
- JCTL
  - not possible with ptrace
- EXIT, DEATH
- REAP
  - not possible with ptrace
- QUIESCE
  - pseudo-event, used with UTRACE_REPORT et al

redhat

utrace API

- struct utrace_engine_ops
  - callback function pointers for each event type
- struct utrace_attached_engine
  - void *data
  - utrace_engine_get() / utrace_engine_put()
- struct task_struct vs struct pid
  - choose your refcount/RCU poison
- enum utrace_resume_action
- utrace_attach_task() or utrace_attach_pid()
  - attach new engine, or look up attached engine
- utrace_set_events() or utrace_set_events_pid()
- utrace_control() or utrace_control_pid()
- utrace_barrier() or utrace_barrier_pid()
- utrace_prepare_examine(), utrace_finish_examine()

redhat

utrace callbacks

- run in traced thread
  - always at “safe point”: no locks, can use user_regset
  - preemptible
- arguments: engine, resume action, + event-specific
- return value
  - resume action (resume/stop/step/etc.) + event-specific
- well-behaved callbacks
  - don't run too long (using traced thread’s CPU time!)
  - don't block much (could break other engines, SIGKILL!)
  - use UTRACE_STOP to sleep: woken via utrace_control()
- synchronizing with callbacks
  - death races: utrace_set_events()/utrace_control() errors
  - utrace_barrier()

redhat

Callback example

```
static u32 syscall_exit(enum utrace_resume_action action,
                struct utrace_attached_engine *engine,
                struct task_struct *task,
                struct pt_regs *regs)
{
   printk("pid %d syscall-exit %ld\n",
          task->pid, syscall_get_error(task, regs));
   return UTRACE_RESUME;
}
...
static const struct utrace_engine_ops my_ops = {
    .report_syscall_exit = syscall_exit,
};
...
```

redhat

utrace API future work

- extension events
  - avoid overloading signals
  - use for hardware trace events
  - dynamically-registered
  - tie-in with tracepoints/markers?
- hw_breakpoint
- engine callback order
- global tracing (?)
  - redundant with tracepoints/markers, so maybe not
  - global syscall tracing
- arch improvements
  - optimize x86 syscall tracing
  - powerpc block-step

redhat

Beyond utrace: lots of hacking to do!

- User-level interfaces
  - fd-based, pollable
  - minimize kernel-user round-trips with debugger
- “groups & rules” engine
  - Underlies user-level interface + in-kernel uses (stap)
  - Trace many threads/processes uniformly (“groups”)
  - Event rules: filters & actions
    - Gather details (registers, etc.) & report to userland
    - Callback (e.g. to stap probe)
    - Manage groups (e.g. on clone, exec)
- Instruction-copying machinery, for:
  - Breakpoint assistance
  - Step emulation without hardware support
  - Step over atomic sequence, e.g. powerpc locks

Questions?

roland@redhat.com | **people.redhat.com/roland**
utrace-devel@redhat.com | **sourceware.org/systemtap/wiki/utrace**