Title   : Kernel Probes (Kprobes)
Authors : Jim Keniston <jkenisto@us.ibm.com>
        : Prasanna S Panchamukhi <prasanna.panchamukhi@gmail.com>
        : Masami Hiramatsu <mhiramat@redhat.com>

CONTENTS

1. Concepts: Kprobes, Jprobes, Return Probes

Kprobes enables you to dynamically break into any kernel routine and collect debugging and performance information non-disruptively. You can trap at almost any kernel code address, specifying a handler routine to be invoked when the breakpoint is hit.

There are currently three types of probes: kprobes, jprobes, and kretprobes (also called return probes). A kprobe can be inserted on virtually any instruction in the kernel. A jprobe is inserted at the entry to a kernel function, and provides convenient access to the function's arguments. A return probe fires when a specified function returns.

In the typical case, Kprobes-based instrumentation is packaged as a kernel module. The module's init function installs ("registers") one or more probes, and the exit function unregisters them. A registration function such as register_kprobe() specifies where the probe is to be inserted and what handler is to be called when the probe is hit.

There are also register_/unregister_*probes() functions for batch registration/unregistration of a group of *probes. These functions can speed up unregistration process when you have to unregister a lot of probes at once.

The next four subsections explain how the different types of probes work and how jump optimization works. They explain certain things that you'll need to know in order to make the best use of Kprobes -- e.g., the difference between a pre_handler and a post_handler, and how to use the maxactive and nmissed fields of a kretprobe. But if you're in a hurry to start using Kprobes, you can skip ahead to section 2.

1.1 How Does a Kprobe Work?

When a kprobe is registered, Kprobes makes a copy of the probed
instruction and replaces the first byte(s) of the probed instruction
with a breakpoint instruction (e.g., int3 on i386 and x86_64).

When a CPU hits the breakpoint instruction, a trap occurs, the CPU's
registers are saved, and control passes to Kprobes via the
notifier_call_chain mechanism. Kprobes executes the "pre_handler"
associated with the kprobe, passing the handler the addresses of the
kprobe struct and the saved registers.

Next, Kprobes single-steps its copy of the probed instruction.
(It would be simpler to single-step the actual instruction in place,
but then Kprobes would have to temporarily remove the breakpoint
instruction. This would open a small time window when another CPU
could sail right past the probepoint.)

After the instruction is single-stepped, Kprobes executes the
"post_handler," if any, that is associated with the kprobe.
Execution then continues with the instruction following the probepoint.

1.2 How Does a Jprobe Work?

A jprobe is implemented using a kprobe that is placed on a function's
entry point. It employs a simple mirroring principle to allow
seamless access to the probed function's arguments. The jprobe
handler routine should have the same signature (arg list and return
type) as the function being probed, and must always end by calling
the Kprobes function jprobe_return().

Here's how it works. When the probe is hit, Kprobes makes a copy of
the saved registers and a generous portion of the stack (see below).
Kprobes then points the saved instruction pointer at the jprobe's
handler routine, and returns from the trap. As a result, control
passes to the handler, which is presented with the same register and
stack contents as the probed function. When it is done, the handler
calls jprobe_return(), which traps again to restore the original stack
contents and processor state and switch to the probed function.

By convention, the callee owns its arguments, so gcc may produce code
that unexpectedly modifies that portion of the stack. This is why
Kprobes saves a copy of the stack and restores it after the jprobe
handler has run. Up to MAX_STACK_SIZE bytes are copied -- e.g.,
64 bytes on i386.

Note that the probed function's args may be passed on the stack
or in registers. The jprobe will work in either case, so long as the
handler's prototype matches that of the probed function.

1.3 Return Probes

1.3.1 How Does a Return Probe Work?

When you call register_kretprobe(), Kprobes establishes a kprobe at
the entry to the function. When the probed function is called and this
probe is hit, Kprobes saves a copy of the return address, and replaces
the return address with the address of a "trampoline." The trampoline

is an arbitrary piece of code -- typically just a nop instruction.
At boot time, Kprobes registers a kprobe at the trampoline.

When the probed function executes its return instruction, control
passes to the trampoline and that probe is hit. Kprobes' trampoline
handler calls the user-specified return handler associated with the
kretprobe, then sets the saved instruction pointer to the saved return
address, and that's where execution resumes upon return from the trap.

While the probed function is executing, its return address is
stored in an object of type kretprobe_instance. Before calling
register_kretprobe(), the user sets the maxactive field of the
kretprobe struct to specify how many instances of the specified
function can be probed simultaneously. register_kretprobe()
pre-allocates the indicated number of kretprobe_instance objects.

For example, if the function is non-recursive and is called with a
spinlock held, maxactive = 1 should be enough. If the function is
non-recursive and can never relinquish the CPU (e.g., via a semaphore
or preemption), NR_CPUS should be enough. If maxactive <= 0, it is
set to a default value. If CONFIG_PREEMPT is enabled, the default
is max(10, 2*NR_CPUS). Otherwise, the default is NR_CPUS.

It's not a disaster if you set maxactive too low; you'll just miss
some probes. In the kretprobe struct, the nmissed field is set to
zero when the return probe is registered, and is incremented every
time the probed function is entered but there is no kretprobe_instance
object available for establishing the return probe.

1.3.2 Kretprobe entry-handler

Kretprobes also provides an optional user-specified handler which runs
on function entry. This handler is specified by setting the entry_handler
field of the kretprobe struct. Whenever the kprobe placed by kretprobe at the
function entry is hit, the user-defined entry_handler, if any, is invoked.
If the entry_handler returns 0 (success) then a corresponding return handler
is guaranteed to be called upon function return. If the entry_handler
returns a non-zero error then Kprobes leaves the return address as is, and
the kretprobe has no further effect for that particular function instance.

Multiple entry and return handler invocations are matched using the unique
kretprobe_instance object associated with them. Additionally, a user
may also specify per return-instance private data to be part of each
kretprobe_instance object. This is especially useful when sharing private
data between corresponding user entry and return handlers. The size of each
private data object can be specified at kretprobe registration time by
setting the data_size field of the kretprobe struct. This data can be
accessed through the data field of each kretprobe_instance object.

In case probed function is entered but there is no kretprobe_instance
object available, then in addition to incrementing the nmissed count,
the user entry_handler invocation is also skipped.

1.4 How Does Jump Optimization Work?

If your kernel is built with CONFIG_OPTPROBES=y (currently this flag

is automatically set 'y' on x86/x86-64, non-preemptive kernel) and
the "debug.kprobes_optimization" kernel parameter is set to 1 (see
sysctl(8)), Kprobes tries to reduce probe-hit overhead by using a jump
instruction instead of a breakpoint instruction at each probepoint.

1.4.1 Init a Kprobe

When a probe is registered, before attempting this optimization,
Kprobes inserts an ordinary, breakpoint-based kprobe at the specified
address. So, even if it's not possible to optimize this particular
probepoint, there'll be a probe there.

1.4.2 Safety Check

Before optimizing a probe, Kprobes performs the following safety checks:

- Kprobes verifies that the region that will be replaced by the jump
instruction (the "optimized region") lies entirely within one function.
(A jump instruction is multiple bytes, and so may overlay multiple
instructions.)

- Kprobes analyzes the entire function and verifies that there is no
jump into the optimized region. Specifically:
  - the function contains no indirect jump;
  - the function contains no instruction that causes an exception (since
  the fixup code triggered by the exception could jump back into the
  optimized region -- Kprobes checks the exception tables to verify this);
  and
  - there is no near jump to the optimized region (other than to the first
  byte).

- For each instruction in the optimized region, Kprobes verifies that
the instruction can be executed out of line.

1.4.3 Preparing Detour Buffer

Next, Kprobes prepares a "detour" buffer, which contains the following
instruction sequence:
- code to push the CPU's registers (emulating a breakpoint trap)
- a call to the trampoline code which calls user's probe handlers.
- code to restore registers
- the instructions from the optimized region
- a jump back to the original execution path.

1.4.4 Pre-optimization

After preparing the detour buffer, Kprobes verifies that none of the
following situations exist:
- The probe has either a break_handler (i.e., it's a jprobe) or a
post_handler.
- Other instructions in the optimized region are probed.
- The probe is disabled.
In any of the above cases, Kprobes won't start optimizing the probe.
Since these are temporary situations, Kprobes tries to start
optimizing it again if the situation is changed.

If the kprobe can be optimized, Kprobes enqueues the kprobe to an
optimizing list, and kicks the kprobe-optimizer workqueue to optimize
it. If the to-be-optimized probepoint is hit before being optimized,
Kprobes returns control to the original instruction path by setting
the CPU's instruction pointer to the copied code in the detour buffer
-- thus at least avoiding the single-step.

1.4.5 Optimization

The Kprobe-optimizer doesn't insert the jump instruction immediately;
rather, it calls synchronize_sched() for safety first, because it's
possible for a CPU to be interrupted in the middle of executing the
optimized region(*). As you know, synchronize_sched() can ensure
that all interruptions that were active when synchronize_sched()
was called are done, but only if CONFIG_PREEMPT=n. So, this version
of kprobe optimization supports only kernels with CONFIG_PREEMPT=n.(**)

After that, the Kprobe-optimizer calls stop_machine() to replace
the optimized region with a jump instruction to the detour buffer,
using text_poke_smp().

1.4.6 Unoptimization

When an optimized kprobe is unregistered, disabled, or blocked by
another kprobe, it will be unoptimized. If this happens before
the optimization is complete, the kprobe is just dequeued from the
optimized list. If the optimization has been done, the jump is
replaced with the original code (except for an int3 breakpoint in
the first byte) by using text_poke_smp().

(*)Please imagine that the 2nd instruction is interrupted and then
the optimizer replaces the 2nd instruction with the jump *address*
while the interrupt handler is running. When the interrupt
returns to original address, there is no valid instruction,
and it causes an unexpected result.

(**)This optimization-safety checking may be replaced with the
stop-machine method that ksplice uses for supporting a CONFIG_PREEMPT=y
kernel.

NOTE for geeks:
The jump optimization changes the kprobe's pre_handler behavior.
Without optimization, the pre_handler can change the kernel's execution
path by changing regs->ip and returning 1. However, when the probe
is optimized, that modification is ignored. Thus, if you want to
tweak the kernel's execution path, you need to suppress optimization,
using one of the following techniques:
- Specify an empty function for the kprobe's post_handler or break_handler.
 or
- Execute 'sysctl -w debug.kprobes_optimization=n'

2. Architectures Supported

Kprobes, jprobes, and return probes are implemented on the following
architectures:

- i386 (Supports jump optimization)
- x86_64 (AMD-64, EM64T) (Supports jump optimization)
- ppc64
- ia64 (Does not support probes on instruction slot1.)
- sparc64 (Return probes not yet implemented.)
- arm
- ppc

3. Configuring Kprobes

When configuring the kernel using make menuconfig/xconfig/oldconfig,
ensure that CONFIG_KPROBES is set to "y". Under "Instrumentation
Support", look for "Kprobes".

So that you can load and unload Kprobes-based instrumentation modules,
make sure "Loadable module support" (CONFIG_MODULES) and "Module
unloading" (CONFIG_MODULE_UNLOAD) are set to "y".

Also make sure that CONFIG_KALLSYMS and perhaps even CONFIG_KALLSYMS_ALL
are set to "y", since kallsyms_lookup_name() is used by the in-kernel
kprobe address resolution code.

If you need to insert a probe in the middle of a function, you may find
it useful to "Compile the kernel with debug info" (CONFIG_DEBUG_INFO),
so you can use "objdump -d -l vmlinux" to see the source-to-object
code mapping.

4. API Reference

The Kprobes API includes a "register" function and an "unregister"
function for each type of probe. The API also includes "register_*probes"
and "unregister_*probes" functions for (un)registering arrays of probes.
Here are terse, mini-man-page specifications for these functions and
the associated probe handlers that you'll write. See the files in the
samples/kprobes/ sub-directory for examples.

4.1 register_kprobe

```
#include <linux/kprobes.h>
int register_kprobe(struct kprobe *kp);
```

Sets a breakpoint at the address kp->addr. When the breakpoint is
hit, Kprobes calls kp->pre_handler. After the probed instruction
is single-stepped, Kprobe calls kp->post_handler. If a fault
occurs during execution of kp->pre_handler or kp->post_handler,
or during single-stepping of the probed instruction, Kprobes calls
kp->fault_handler. Any or all handlers can be NULL. If kp->flags
is set KPROBE_FLAG_DISABLED, that kp will be registered but disabled,
so, its handlers aren't hit until calling enable_kprobe(kp).

NOTE:
1. With the introduction of the "symbol_name" field to struct kprobe,
the probepoint address resolution will now be taken care of by the kernel.
The following will now work:

```
        kp.symbol_name = "symbol_name";
```

(64-bit powerpc intricacies such as function descriptors are handled
transparently)

2. Use the "offset" field of struct kprobe if the offset into the symbol
to install a probepoint is known. This field is used to calculate the
probepoint.

3. Specify either the kprobe "symbol_name" OR the "addr". If both are
specified, kprobe registration will fail with -EINVAL.

4. With CISC architectures (such as i386 and x86_64), the kprobes code
does not validate if the kprobe.addr is at an instruction boundary.
Use "offset" with caution.

register_kprobe() returns 0 on success, or a negative errno otherwise.

User's pre-handler (kp->pre_handler):
```
#include <linux/kprobes.h>
#include <linux/ptrace.h>
int pre_handler(struct kprobe *p, struct pt_regs *regs);
```

Called with p pointing to the kprobe associated with the breakpoint,
and regs pointing to the struct containing the registers saved when
the breakpoint was hit. Return 0 here unless you're a Kprobes geek.

User's post-handler (kp->post_handler):
```
#include <linux/kprobes.h>
#include <linux/ptrace.h>
void post_handler(struct kprobe *p, struct pt_regs *regs,
        unsigned long flags);
```

p and regs are as described for the pre_handler. flags always seems
to be zero.

User's fault-handler (kp->fault_handler):
```
#include <linux/kprobes.h>
#include <linux/ptrace.h>
int fault_handler(struct kprobe *p, struct pt_regs *regs, int trapnr);
```

p and regs are as described for the pre_handler. trapnr is the
architecture-specific trap number associated with the fault (e.g.,
on i386, 13 for a general protection fault or 14 for a page fault).
Returns 1 if it successfully handled the exception.

4.2 register_jprobe

```
#include <linux/kprobes.h>
int register_jprobe(struct jprobe *jp)
```

Sets a breakpoint at the address jp->kp.addr, which must be the address
of the first instruction of a function. When the breakpoint is hit,
Kprobes runs the handler whose address is jp->entry.

The handler should have the same arg list and return type as the probed
function; and just before it returns, it must call jprobe_return().

(The handler never actually returns, since jprobe_return() returns
control to Kprobes.) If the probed function is declared asmlinkage
or anything else that affects how args are passed, the handler's
declaration must match.

register_jprobe() returns 0 on success, or a negative errno otherwise.

4.3 register_kretprobe

```
#include <linux/kprobes.h>
int register_kretprobe(struct kretprobe *rp);
```

Establishes a return probe for the function whose address is
rp->kp.addr. When that function returns, Kprobes calls rp->handler.
You must set rp->maxactive appropriately before you call
register_kretprobe(); see "How Does a Return Probe Work?" for details.

register_kretprobe() returns 0 on success, or a negative errno
otherwise.

User's return-probe handler (rp->handler):

```
#include <linux/kprobes.h>
#include <linux/ptrace.h>
int kretprobe_handler(struct kretprobe_instance *ri, struct pt_regs *regs);
```

regs is as described for kprobe.pre_handler. ri points to the
kretprobe_instance object, of which the following fields may be
of interest:
- ret_addr: the return address
- rp: points to the corresponding kretprobe object
- task: points to the corresponding task struct
- data: points to per return-instance private data; see "Kretprobe
        entry-handler" for details.

The regs_return_value(regs) macro provides a simple abstraction to
extract the return value from the appropriate register as defined by
the architecture's ABI.

The handler's return value is currently ignored.

4.4 unregister_*probe

```
#include <linux/kprobes.h>
void unregister_kprobe(struct kprobe *kp);
void unregister_jprobe(struct jprobe *jp);
void unregister_kretprobe(struct kretprobe *rp);
```

Removes the specified probe. The unregister function can be called
at any time after the probe has been registered.

NOTE:
If the functions find an incorrect probe (ex. an unregistered probe),
they clear the addr field of the probe.

4.5 register_*probes

```
#include <linux/kprobes.h>
int register_kprobes(struct kprobe **kps, int num);
int register_kretprobes(struct kretprobe **rps, int num);
int register_jprobes(struct jprobe **jps, int num);
```

Registers each of the num probes in the specified array. If any
error occurs during registration, all probes in the array, up to
the bad probe, are safely unregistered before the register_*probes
function returns.
- kps/rps/jps: an array of pointers to *probe data structures
- num: the number of the array entries.

NOTE:
You have to allocate(or define) an array of pointers and set all
of the array entries before using these functions.

4.6 unregister_*probes

```
#include <linux/kprobes.h>
void unregister_kprobes(struct kprobe **kps, int num);
void unregister_kretprobes(struct kretprobe **rps, int num);
void unregister_jprobes(struct jprobe **jps, int num);
```

Removes each of the num probes in the specified array at once.

NOTE:
If the functions find some incorrect probes (ex. unregistered
probes) in the specified array, they clear the addr field of those
incorrect probes. However, other probes in the array are
unregistered correctly.

4.7 disable_*probe

```
#include <linux/kprobes.h>
int disable_kprobe(struct kprobe *kp);
int disable_kretprobe(struct kretprobe *rp);
int disable_jprobe(struct jprobe *jp);
```

Temporarily disables the specified *probe. You can enable it again by using
enable_*probe(). You must specify the probe which has been registered.

4.8 enable_*probe

```
#include <linux/kprobes.h>
int enable_kprobe(struct kprobe *kp);
int enable_kretprobe(struct kretprobe *rp);
int enable_jprobe(struct jprobe *jp);
```

Enables *probe which has been disabled by disable_*probe(). You must specify
the probe which has been registered.

5. Kprobes Features and Limitations

Kprobes allows multiple probes at the same address. Currently,
however, there cannot be multiple jprobes on the same function at
the same time. Also, a probepoint for which there is a jprobe or

a post_handler cannot be optimized. So if you install a jprobe,
or a kprobe with a post_handler, at an optimized probepoint, the
probepoint will be unoptimized automatically.

In general, you can install a probe anywhere in the kernel.
In particular, you can probe interrupt handlers. Known exceptions
are discussed in this section.

The register_*probe functions will return -EINVAL if you attempt
to install a probe in the code that implements Kprobes (mostly
kernel/kprobes.c and arch/*/kernel/kprobes.c, but also functions such
as do_page_fault and notifier_call_chain).

If you install a probe in an inline-able function, Kprobes makes
no attempt to chase down all inline instances of the function and
install probes there. gcc may inline a function without being asked,
so keep this in mind if you're not seeing the probe hits you expect.

A probe handler can modify the environment of the probed function
-- e.g., by modifying kernel data structures, or by modifying the
contents of the pt_regs struct (which are restored to the registers
upon return from the breakpoint). So Kprobes can be used, for example,
to install a bug fix or to inject faults for testing. Kprobes, of
course, has no way to distinguish the deliberately injected faults
from the accidental ones. Don't drink and probe.

Kprobes makes no attempt to prevent probe handlers from stepping on
each other -- e.g., probing printk() and then calling printk() from a
probe handler. If a probe handler hits a probe, that second probe's
handlers won't be run in that instance, and the kprobe.nmissed member
of the second probe will be incremented.

As of Linux v2.6.15-rc1, multiple handlers (or multiple instances of
the same handler) may run concurrently on different CPUs.

Kprobes does not use mutexes or allocate memory except during
registration and unregistration.

Probe handlers are run with preemption disabled. Depending on the
architecture, handlers may also run with interrupts disabled. In any
case, your handler should not yield the CPU (e.g., by attempting to
acquire a semaphore).

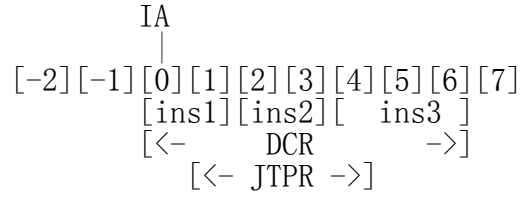
Since a return probe is implemented by replacing the return
address with the trampoline's address, stack backtraces and calls
to __builtin_return_address() will typically yield the trampoline's
address instead of the real return address for kretprobed functions.
(As far as we can tell, __builtin_return_address() is used only
for instrumentation and error reporting.)

If the number of times a function is called does not match the number
of times it returns, registering a return probe on that function may
produce undesirable results. In such a case, a line:
kretprobe BUG!: Processing kretprobe d000000000041aa8 @ c00000000004f48c
gets printed. With this information, one will be able to correlate the
exact instance of the kretprobe that caused the problem. We have the

do_exit() case covered. do_execve() and do_fork() are not an issue.
We're unaware of other specific cases where this could be a problem.

If, upon entry to or exit from a function, the CPU is running on
a stack other than that of the current task, registering a return
probe on that function may produce undesirable results. For this
reason, Kprobes doesn't support return probes (or kprobes or jprobes)
on the x86_64 version of __switch_to(); the registration functions
return -EINVAL.

On x86/x86-64, since the Jump Optimization of Kprobes modifies
instructions widely, there are some limitations to optimization. To
explain it, we introduce some terminology. Imagine a 3-instruction
sequence consisting of a two 2-byte instructions and one 3-byte
instruction.



ins1: 1st Instruction
ins2: 2nd Instruction
ins3: 3rd Instruction
IA: Insertion Address
JTPR: Jump Target Prohibition Region
DCR: Detoured Code Region

The instructions in DCR are copied to the out-of-line buffer
of the kprobe, because the bytes in DCR are replaced by
a 5-byte jump instruction. So there are several limitations.

a) The instructions in DCR must be relocatable.
b) The instructions in DCR must not include a call instruction.
c) JTPR must not be targeted by any jump or call instruction.
d) DCR must not straddle the border betweeen functions.

Anyway, these limitations are checked by the in-kernel instruction
decoder, so you don't need to worry about that.

6. Probe Overhead

On a typical CPU in use in 2005, a kprobe hit takes 0.5 to 1.0
microseconds to process. Specifically, a benchmark that hits the same
probepoint repeatedly, firing a simple handler each time, reports 1-2
million hits per second, depending on the architecture. A jprobe or
return-probe hit typically takes 50-75% longer than a kprobe hit.
When you have a return probe set on a function, adding a kprobe at
the entry to that function adds essentially no overhead.

Here are sample overhead figures (in usec) for different architectures.
k = kprobe; j = jprobe; r = return probe; kr = kprobe + return probe
on same function; jr = jprobe + return probe on same function

i386: Intel Pentium M, 1495 MHz, 2957.31 bogomips
k = 0.57 usec; j = 1.00; r = 0.92; kr = 0.99; jr = 1.40

x86_64: AMD Opteron 246, 1994 MHz, 3971.48 bogomips
k = 0.49 usec; j = 0.76; r = 0.80; kr = 0.82; jr = 1.07

ppc64: POWER5 (gr), 1656 MHz (SMT disabled, 1 virtual CPU per physical CPU)
k = 0.77 usec; j = 1.31; r = 1.26; kr = 1.45; jr = 1.99

6.1 Optimized Probe Overhead

Typically, an optimized kprobe hit takes 0.07 to 0.1 microseconds to
process. Here are sample overhead figures (in usec) for x86 architectures.
k = unoptimized kprobe, b = boosted (single-step skipped), o = optimized kprobe,
r = unoptimized kretprobe, rb = boosted kretprobe, ro = optimized kretprobe.

i386: Intel(R) Xeon(R) E5410, 2.33GHz, 4656.90 bogomips
k = 0.80 usec; b = 0.33; o = 0.05; r = 1.10; rb = 0.61; ro = 0.33

x86-64: Intel(R) Xeon(R) E5410, 2.33GHz, 4656.90 bogomips
k = 0.99 usec; b = 0.43; o = 0.06; r = 1.24; rb = 0.68; ro = 0.30

7. TODO

a. SystemTap (http://sourceware.org/systemtap): Provides a simplified
programming interface for probe-based instrumentation. Try it out.
b. Kernel return probes for sparc64.
c. Support for other architectures.
d. User-space probes.
e. Watchpoint probes (which fire on data references).

8. Kprobes Example

See samples/kprobes/kprobe_example.c

9. Jprobes Example

See samples/kprobes/jprobe_example.c

10. Kretprobes Example

See samples/kprobes/kretprobe_example.c

For additional information on Kprobes, refer to the following URLs:
http://www-106.ibm.com/developerworks/library/l-kprobes.html?ca=dgr-lnxw42Kprobe
http://www.redhat.com/magazine/005mar05/features/kprobes/
http://www-users.cs.umn.edu/~boutcher/kprobes/
http://www.linuxsymposium.org/2006/linuxsymposium_procv2.pdf (pages 101-115)

Appendix A: The kprobes debugfs interface

With recent kernels (> 2.6.20) the list of registered kprobes is visible
under the /sys/kernel/debug/kprobes/ directory (assuming debugfs is mounted at
//sys/kernel/debug).

/sys/kernel/debug/kprobes/list: Lists all registered probes on the system

```
c015d71a  k  vfs_read+0x0
c011a316  j  do_fork+0x0
c03dedc5  r  tcp_v4_rcv+0x0
```

The first column provides the kernel address where the probe is inserted.
The second column identifies the type of probe (k - kprobe, r - kretprobe
and j - jprobe), while the third column specifies the symbol+offset of
the probe. If the probed function belongs to a module, the module name
is also specified. Following columns show probe status. If the probe is on
a virtual address that is no longer valid (module init sections, module
virtual addresses that correspond to modules that've been unloaded),
such probes are marked with [GONE]. If the probe is temporarily disabled,
such probes are marked with [DISABLED]. If the probe is optimized, it is
marked with [OPTIMIZED].

/sys/kernel/debug/kprobes/enabled: Turn kprobes ON/OFF forcibly.

Provides a knob to globally and forcibly turn registered kprobes ON or OFF.
By default, all kprobes are enabled. By echoing "0" to this file, all
registered probes will be disarmed, till such time a "1" is echoed to this
file. Note that this knob just disarms and arms all kprobes and doesn't
change each probe's disabling state. This means that disabled kprobes (marked
[DISABLED]) will be not enabled if you turn ON all kprobes by this knob.

Appendix B: The kprobes sysctl interface

/proc/sys/debug/kprobes-optimization: Turn kprobes optimization ON/OFF.

When CONFIG_OPTPROBES=y, this sysctl interface appears and it provides
a knob to globally and forcibly turn jump optimization (see section
1.4) ON or OFF. By default, jump optimization is allowed (ON).
If you echo "0" to this file or set "debug.kprobes_optimization" to
0 via sysctl, all optimized probes will be unoptimized, and any new
probes registered after that will not be optimized. Note that this
knob *changes* the optimized state. This means that optimized probes
(marked [OPTIMIZED]) will be unoptimized ([OPTIMIZED] tag will be
removed). If the knob is turned on, they will be optimized again.