

## Re: [PATCH][RFC] kprobes: Add user entry–handler in kretprobes

---

*Source:* <http://linux.derkeiler.com/Mailing-Lists/Kernel/2007-11/msg05204.html>

---

- *From:* Jim Keniston <jkenisto@xxxxxxxxxx>
  - *Date:* Thu, 15 Nov 2007 13:16:10 -0800
- 

On Thu, 2007-11-15 at 18:46 +0530, Abhishek Sagar wrote:

On Nov 15, 2007 4:21 AM, Jim Keniston <jkenisto@xxxxxxxxxx> wrote:

...

First of all, some general comments. We seem to be trying to solve two problems here:

1. Prevent the asymmetry in entry– vs. return–handler calls that can develop when we temporarily run out of kretprobe\_instances. E.g., if we have m kretprobe "misses", we may report n calls but only (n–m) returns.

That has already been taken care of. ...

[Much discussion snipped]

... in your patch. Yes, I think we're in violent agreement on that point.

2. Simplify the task of correlating data (e.g., timestamps) between function entry and function return.

Right. Srinivasa and you have been hinting at the use of per–instance private data for the same. I think ri should be enough.

Problem #1 wouldn't exist if we could solve problem #1a:

- 1a. Ensure that we never run out of kretprobe\_instances (for some appropriate value of "never").

I've thought of various approaches to #1a -- e.g., allocate kretprobe\_instances from GFP\_ATOMIC memory when we're out of preallocated instances -- but I've never found time to pursue them.

Re: [PATCH][RFC] kprobes: Add user entry–handler in kretprobes

This might be a good time to brainstorm solutions to that problem.

Lacking a solution to #1a, I think Abhishek's approach provides a reasonable solution to problem #1.

If you're not convinced that problem #1 isn't appropriately handled,

I don't know where you got that idea. Reread my last sentence above.

My concern is that your patch fixes one symptom (number of function returns reported < number of function entries reported), but not the root cause (we can fail to report some function returns). If fixing one symptom is the best we can do, then I have no real objection to that.

Of course, we need to keep in mind that you're adding an externally visible feature that would need to be maintained, so it demands more scrutiny than a simple bug fix.

we should look for something like that I guess.

I don't think it takes us very far toward solving #2, though. I agree with Srinivasa that it would be more helpful if we could store the data collected at function–entry time right in the kretprobe\_instance. Kevin Stafford prototyped this "data pouch" idea a couple of years ago — <http://sourceware.org/ml/systemtap/2005-q3/msg00593.html> — but an analogous feature was implemented at a higher level in SystemTap. Either approach — in kprobes or SystemTap — would benefit from a fix to #1 (or #1a).

There are three problems in the "data pouch" approach, which is a generalized case of Srinivasa's timestamp case:

1. It bloats the size of each return instance to a run–time defined value. I'm certain that quite a few memory starved ARM kprobes users would certainly be wishing they could install more probes on their system without taking away too much from the precious memory pools which can impact their system performance. This is not a deal breaker though, just an annoyance.

entry\_info is, by default, a zero–length array, which adds nothing to the size of a uretprobe\_instance — at least on the 3 architectures I've tested on (i386, x86\_64, powerpc).

## Re: [PATCH][RFC] kprobes: Add user entry-handler in kretprobes

2. Forces user entry/return handlers to use `ri->entry_info` (see Kevin's patch) even if their design only wanted such private data to be used in certain instances.

No, it doesn't. Providing a feature isn't the same as forcing people to use the feature.

Therefore ideally, any per-instance data allocation should be left to user entry handlers, IMO. Even if we allow a pouch for private data in a return instance, the user handlers would still need be aware of "return instances" to actually use them globally.

3. It's redundant. 'ri' can uniquely identify any entry-return handler pair. This itself solves your problem #2. It only moves the onus of private data allocation to user handlers.

Having `ri` available at function entry time is definitely a win, but maintaining separate data structures and using `ri` to map to the right one is no trivial task. (It's a lot easier if you pre-allocate the maximum number of data structures you'll need -- presumably matching `rp->maxactive` -- but then you have at least the same amount of "bloat" as with the data pouch approach.)

I suggest you show us a probe module that captures data at function entry and reports it upon return, exploiting your proposed patch. Profiling would be a reasonable example, but something where multiple values are captured might be more relevant. (Your example below doesn't count because it doesn't work.) Then I'll code up the same example using your enhancement + the data pouch enhancement, and we can compare.

Of course, the data pouch enhancement would build nicely atop your enhancement, so it could be proposed and considered separately.

### Review of Abhishek's patch:

I see no reason to save a copy of `*regs` and pass that to the entry handler. Passing the real `regs` pointer is good enough for other kprobe handlers.

No, a copy is required because if the `entry_handler()` returns error (a voluntary miss), then there has to be a way to roll-back all the changes that `arch_prepare_kretprobe()` and `entry_handler()` made to

Re: [PATCH][RFC] kprobes: Add user entry-handler in kretprobes

\*regs. Such an instance is considered "aborted".

No. Handlers shouldn't be writing to the pt\_regs struct unless they want to change the operation of the probed code. If an entry handler scribbles on \*regs and then decides to "abort", it's up to that handler to restore the contents of \*regs. It's that way with all kprobe and kretprobe handlers, and the proposed entry handler is no different, as far as I can see.

And if a handler on i386 uses &regs->esp as the value of the stack pointer (which is correct -- long story), it'll get the wrong value if its regs arg points at the copy.

That's a catch! I've made the fix (see inlined patch below). It now passes regs instead of &copy to both the entry\_handler() and arch\_prepare\_kretprobe().

More comments below.

[snip]

1. Multiple function entries from various tasks (the one you've just pointed out).
2. Multiple kretprobe registration on the same function.
3. Nested calls of kretprobe'd function.

In cases 1 and 3, the following information can be used to match corresponding entry and return handlers inside a user handler (if needed):

(ri->task, ri->ret\_addr)  
where ri is struct kretprobe\_instance \*

This tuple should uniquely identify a return address (right?).

But if it's a recursive function, there could be multiple instances in the same task with the same return address. The stack pointer would be different, FWIW.

Wouldn't the return addresses be different for recursive/nested calls?

Re: [PATCH][RFC] kprobes: Add user entry-handler in kretprobes

Not for recursive calls. Think about it. Or write a little program that calls a recursive function like factorial(), and have factorial() report the value of \_\_builtin\_return\_address(0) each time it's called.

I think the only case where the return addresses would be same is when multiple return probes are installed on the same function.

The fact that ri is passed to both handlers should allow any user handler to identify each of these cases and take appropriate synchronization action pertaining to its private data, if needed.

I don't think Abhishek has made his case here. See below.

(Hence I feel sol a) would be nice).

With an entry-handler, any module aiming to profile running time of a function (say) can simply do the following without being "return instance" conscious. Note however that I'm not trying to address just this scenario but trying to provide a general way to use entry-handlers in kretprobes:

```
static unsigned long flag = 0; /* use bit 0 as a global flag */
unsigned long long entry, exit;
```

```
int my_entry_handler(struct kretprobe_instance *ri, struct
pt_regs *regs)
{
if (!test_and_set_bit(0, &flag))
/* this instance claims the first entry to kretprobe'd function
*/
entry = sched_clock();
/* do other stuff */
return 0; /* right on! */
}
return 1; /* error: no return instance to be allocated for this
function entry */
}
```

Re: [PATCH][RFC] kprobes: Add user entry-handler in kretprobes

```
/* will only be called iff flag == 1 */
int my_return_handler(struct kretprobe_instance *ri, struct
pt_regs *regs)
{
BUG_ON(!flag);
exit = sched_clock();
set_bit(0, &flag);
}
```

I think something like this should do the trick for you.

Since flag is static, it seems to me that if there were instances of the probed function active concurrently in multiple tasks, only the first-called instance would be profiled.

Oh that's right, or we could use a per-cpu flag which would restrict us to only one profiling instance per processor.

If the probed function can sleep, then it could return on a different CPU; a per-cpu flag wouldn't work in such cases.

Jim Keniston

---  
Thanks & Regards  
Abhishek Sagar

---  
Signed-off-by: Abhishek Sagar <sagar.abhishek@xxxxxxxxxx>

```
diff -upNr linux-2.6.24-rc2/include/linux/kprobes.h
linux-2.6.24-rc2_kp/include/linux/kprobes.h
--- linux-2.6.24-rc2/include/linux/kprobes.h 2007-11-07 03:27:46.000000000 +0530
+++ linux-2.6.24-rc2_kp/include/linux/kprobes.h 2007-11-15
15:49:39.000000000 +0530
@@ -152,6 +152,7 @@ static inline int arch_trampoline_kprobe
struct kretprobe {
struct kprobe kp;
kretprobe_handler_t handler;
+ kretprobe_handler_t entry_handler;
int maxactive;
int nmissed;
struct hlist_head free_instances;
diff -upNr linux-2.6.24-rc2/kernel/kprobes.c
```

Re: [PATCH][RFC] kprobes: Add user entry-handler in kretprobes

Re: [PATCH][RFC] kprobes: Add user entry-handler in kretprobes

```
linux-2.6.24-rc2_kp/kernel/kprobes.c
--- linux-2.6.24-rc2/kernel/kprobes.c 2007-11-07 03:27:46.000000000 +0530
+++ linux-2.6.24-rc2_kp/kernel/kprobes.c 2007-11-15 16:00:57.000000000 +0530
@@ -694,12 +694,24 @@ static int __kprobes pre_handler_kretpro
spin_lock_irqsave(&kretprobe_lock, flags);
if (!hlist_empty(&rp->free_instances)) {
struct kretprobe_instance *ri;
+ struct pt_regs copy;
```

NAK. Saving and restoring regs is expensive and inconsistent with existing kprobes usage.

```
ri = hlist_entry(rp->free_instances.first,
struct kretprobe_instance, uflist);
ri->rp = rp;
ri->task = current;
- arch_prepare_kretprobe(ri, regs);
+
+ if (rp->entry_handler) {
+ copy = *regs;
+ arch_prepare_kretprobe(ri, regs);
+ if (rp->entry_handler(ri, regs)) {
+ *regs = copy;
+ spin_unlock_irqrestore(&kretprobe_lock, flags);
+ return 0; /* skip current kretprobe instance */
+ }
+ } else {
+ arch_prepare_kretprobe(ri, regs);
+ }

/* XXX(hch): why is there no hlist_move_head? */
hlist_del(&ri->uflist);
```

Jim

-

To unsubscribe from this list: send the line "unsubscribe linux-kernel" in the body of a message to majordomo@xxxxxxxxxxxxxxxxxxx

More majordomo info at <http://vger.kernel.org/majordomo-info.html>

Please read the FAQ at <http://www.tux.org/lkml/>