Ghost-counters used in the verification of `usbkbd` (draft text)

Willem Penninckx, Jan Tobias Mühlberg, Jan Smans, Bart Jacobs, and Frank Piessens

1 Introduction

This text was originally supposed to be part of the section in [1] about the parts of verification that were considered harder. Because of page limits, this text was not included in the [1].

See `http://people.cs.kuleuven.be/~willem.penninckx/usbkbd/`.

This text describes the ghost-counters used while performing verification of Linux's USB Boot Protocol keyboard driver (usbkbd). This is considered to be one the most complex part of the verification approach. However, it was not the most time-consuming.

This is only a draft text. Most authors should not be considered responsible for any errors, inaccuracies and opinions expressed in this text.

2 Killable Asynchrounous Resubmittable Requests with completion callback

Since both `usb_kbd_event` and `usb_kbd_led` submit the LED URB, they need to be synchronized. This is done by a boolean representing whether the URB is in progress and a spinlock protecting data including the boolean. `usb_kbd_led` will thus need a fraction of the lock handle. As a result, `usb_kbd_event` will need to give a fraction of his fraction of the lock handle to the `usb_submit_urb` as part of the callback data that the callback (here `usb_kbd_event`) will receive. Because the URB is submitted multiple times, the callback data (here fractions of the lock handle) will be given to the USB API multiple times. They will be given back when the URB is killed. If an URB is submitted n times, n times the callback data will be returned when the URB is killed.

This is a simplification. In fact, the cb-out is given back, not the cb-in. This is explained later on.

In order to enforce good API usage, the USB API's specifications ask to prove that the caller has submitted the URB n times if it wants the callback-data back n times. To facilitate this, `usb_submit_urb` returns a predicate that we call a ticket that indicates the URB is submitted. `usb_kill_urb` gives n times the callback predicate if it gets n tickets.

Note that we already introduced two counters: one counter that counts the number of times an URB is submitted (i.e. the amount of tickets), and one that counts the number of times a fraction of the lock handle is put in a callback data predicate. Since these two counters always have the same value, we only use one ghost counter. We call this counter `killcount`.

`usb_kbd_disconnect` must be able to free the LED URB. In order to do this, the struct containing the URB data must be obtained. This struct is contained in the lock invariant (i.e. "the data the lock protects") conditionally: if the boolean representing whether the URB is in progress is false, the lock contains the URB struct. Otherwise it does not since it is then owned by the URB API. Note that `usb_kbd_led` receives the URB struct such that it can resubmit the URB, but it can also store it in the spinlock if it does not resubmit.

Since `usb_kbd_disconnect` must obtain the URB struct, it must thus prove that the boolean is false. To be able to do this, we introduce a second counter `cb_out_count` which represent the number of times `usb_kbd_led` has returned without resubmitting the URB. A completion callback has "incoming data" (originally passed to `usb_submit_urb`), and "output data" (which will be given back by `usb_kill_urb`). If a completion handler does not resubmit, it must give back the output data in its postcondition. Otherwise, it must give back the incoming data and the predicate representing the URB struct such that the URB can be resubmitted by the USB API (note that resubmitting is thus deferred until the completion handler returns).

The lock invariant contains the claim that `cb_out_count` equals `killcount` if the boolean is false, and `cb_out_count` is one less than `killcount` otherwise.

So, `usb_kbd_disconnect` only needs to prove that the two counters are the same, such that it can prove the boolean is false and take the URB struct out of the lock invariant and free it. It is sufficient to prove (i.e. convince VeriFast) that `killcount=<cb_out_count`.

Since `usb_kbd_disconnect` kills the LED URB, it gets `killcount` times the callback "output data". We will use a special counter for `cb_out_count` that allows us to prove that `cb_out_count` is a least as big as `killcount`, provided that we have `killcount` times the callback "output data".

Let us now look at how the `cb_out_count` counter works. This counter is just represented by a predicate with the counter value as one of the arguments. By increasing the counter value, a ticket associated with the counter is returned. Decreases eats one ticket. Given n tickets, the counter axioms state that the counter must be at least n high. In order to allow this to work, creating a counter requires a unique predicate, i.e. a predicate that can only have one instance for its arguments. Otherwise you could exchange tickets between counters, which would result in an important unsoundness. Creating a counter thus requires a predicate and a proof that this predicate is unique; both need to be passed as argument. Also note that we needs this uniqueness property to allow sharing counters inside multiple predicates. Otherwise it would be unknown whether two predicates containing the same counter are referring to the same counter value.

By putting the tickets of `cb_out_count` in the callback "output data", we thus obtain `killcount` times this ticket, which allows us to prove that `cb_out_count>=killcount`.

While we were able to perform verification by working out the details of the above explained idea, we believe it would be nice if a more simple approach would be applicable.

It is also intresting to see that the part that was the hardest to get verified, was also the part that contained the bugs that we found in this driver. Originally, the driver did not perform any real synchronization between `usb_kbd_event` and `usb_kbd_led`, besides checking in a racy way whether the URB is in progress by reading a field of the URB struct (outside the completion handler), which is explicitly forbidden by the USB API documentation. The LED URB was originally also never killed. The patches we made for fixing these bugs are accepted by the maintainer of the driver.

References

[1] Willem Penninckx, Jan Tobias Mühlberg, Jan Smans, Bart Jacobs, and Frank Piessens. Sound formal verification of Linux's USB BP keyboard driver. In *NASA Formal Methods*, April 2012.