The usbmon: USB monitoring framework

Pete Zaitcev
Red Hat, Inc.
zaitcev@redhat.com

Abstract

For years, Linux developers used `printk()` to debug the USB stack, but this approach has serious limitations. In this paper we discuss "usbmon," a recently developed facility to snoop USB traffic in a more efficient way than can be done with `printk()`.

From far away, usbmon is a very straightforward piece of code. It consists of circular buffers which are filled with records by hooks into the USB stack and a thin glue to the user code which fetches these records. The devil, however, is in details. Also the user mode tools play a role.

1 Introduction

This paper largely deals with the kernel part of the USB monitoring infrastructure, which is properly called "usbmon" (all in lower case). We describe usbmon's origins, overall design, internals, and how it is used, both by C code in kernel and by human users. To conclude, we consider if experience with usbmon is applicable to subsystems other than USB.

2 Origins

Although the need to have a robust, simple, and unobtrusive method of snooping appears to be self-evident, Linux USB developers were getting by with layers of macros on top of `printk()` for years. Current debugging facilities are represented by a patchwork of build-time configuration settings, such as `CONFIG_USB_STORAGE_DEBUG`. To make the use of systems with tracing included more palatable, `usbserial` and several other modules offer "debug" parameter.

Limitations of the this approach became pronounced as more users running preconfigured kernels appeared. For a developer, it is often undesirable to make users to rebuild their kernels with `CONFIG_USB_STORAGE_DEBUG` enabled. These difficulties could be overcome by making tracing configurable at runtime, by a module parameter. Nonetheless, this style of tracing is still not ideal, for several reasons. Output to the system console and/or log file is lossy when a large amount of data is piped through though the syslog subsystem. Timing variations introduced by formatted printouts skew results, which makes it harder to pinpoint problems when peripherals require delays in the access pattern. And finally, `printk()` calls have to be added all the time to capture what is important. Often it seems as if the one key `printk()` is missing, but once added, it

stays in the code forever and serves to obscure printouts needed at that time.

A facility similar to tcpdump(8) would be a great help for USB developers. The usbmon aims to provide one.

David Harding proposed a patch to address this need back in 2002, but for various reasons that effort stalled without producing anything suitable to be accepted into the Linus' kernel. The usbmon picks up where the previous work left and is available in the mainline kernel starting with version 2.6.11.

3 Architecture

The USB monitoring or snooping facilities for Linux consist of the kernel part, or usbmon, and the user mode part, or user mode tools. To jump-start the development, usbmon took the lead while tools lagged.

At highest level of architecture, usbmon is uncomplicated. It consists of circular buffers, fed by hooks in the USB stack. Every call puts an event into a buffer. From there, user processes fetch these events for further processing or presentation to humans. Events for all devices on a particular bus are delivered to users together, separately from other buses. There is no filtering facility of any sort.

At the lower level, a couple of interesting decisions were made regarding the placement of hooks and the formatting of events when presented to user programs.

An I/O request in the USB stack is represented by so-called "URB". A peripheral-specific driver, such as usb-storage, initializes and submits URB with a call to the USB stack core. The core dispatches URB to a Host Controller Driver, or HCD. When I/O is done, HCD invokes a specified callback to notify the core and the requesting driver. The usbmon hooks reside in the core of USB stack, in the submission and callback paths. Thus, usbmon relies on HCD to function properly and is only marginally useful in debugging of HCDs. Such an arrangement is accepted for two reasons. First, it allows usbmon to be unobtrusive and significantly less buggy itself. Second, the vast majority of bugs occur outside of HCDs, in in upper level drivers or peripherals.

The user interface to the usbmon answers to diverse sets of requirements with priorities changing over time. Initially, a premium is placed on ease of implementation and the possibility to access the information with simple tools. But in perspective, performance starts to play a larger role. The first group of requirements favors an interface typified by `/proc` filesystem, the one of pseudo text files. The second group pulls toward a binary and versioned API.

Instead of forcing a choice between text and binary interfaces, usbmon adopts a neutral solution. Its data structures are set up to facilitate several distinct types of consumers of events (called "readers"). Various reader classes can provide text and binary interfaces. At this time, only text-based interface class is implemented.

Every instance of a reader has its own circular buffer. When hooks are called, they broadcast events to readers. Readers replicate events into all buffers which are active for a given bus. To be sure, this entails an extra overhead of data copying. However, the complication of having all aliasing properly tracked and resolved turned out to be insurmountable in the time frame desired, and the performance impact was found manageable.

```
struct mon_bus {
    struct list_head bus_link;
    spinlock_t lock;
    struct dentry *dent_s;      /* Debugging file */
    struct dentry *dent_t;      /* Text interface file */
    struct usb_bus *u_bus;
    /* Ref */
    int nreaders;               /* Under mon_lock AND mbus->lock */
    struct list_head r_list;    /* Chain of readers (usually one) */
    struct kref ref;            /* Under mon_lock */
    /* Stats */
    unsigned int cnt_text_lost;
};

struct mon_reader {    /* An instance of a process which opened a file */
    struct list_head r_link;
    struct mon_bus *m_bus;
    void *r_data;
    void (*rnf_submit)(void *data, struct urb *urb);
    void (*rnf_complete)(void *data, struct urb *urb);
};
```

Figure 1: The bus and readers.

4 Implementation

The usbmon is implemented as a Linux kernel module, which can be loaded and unloaded at will. This arrangement is not intrinsic to the design; it is intended to serve as a convenience to developers only. Hooks and additional data fields remain built into the USB stack core at all times as long as usbmon is configured on. In a proprietary OS, usbmon would have to be implemented in a different way. It is entirely possible to make the usbmon an add-on that stacks on top of HCDs by manipulating existing function pointers. Such an implementation would make usbmon effectively non-existing when not actively monitoring. However, this approach introduces a significant complexity of tracking of active URBs which had their function pointers replaced, and brings only a marginal advantage for an open-source OS. In present, when usbmon is not running, it adds 8 bytes of memory use per bus (on a 32-bit system) and an additional `if()` statement in submit and callback paths. This was deemed an acceptable price for the lack of tracking.

The key data structure that keeps usbmon together is `struct mon_bus` (See Figure 1). One of them is allocated for every USB bus present. It holds a list of readers attached to the bus, pointer to the corresponding bus structure, statistic counters, reference count, and a spinlock. The manner in which circular buffers are arranged is encapsulated entirely within a reader.

The locking model is straightforward. All hooks execute with the bus spinlock taken, so readers do not do any extra locking. The only time instances of `struct mon_bus` may in-

fluence each other is when buses are added or removed. Data words touched at that time, such as linked list entries, are covered with a global semaphore "mon_lock".

The reference count is needed because buses are added and removed while user processes access devices. Captured events may remain in buffers after a bus was removed. The count is implemented with a predefined kernel facility called "kref". The mon_lock is used to support kref as required by API.

5 Interfaces

The usbmon provides two principal interfaces: the one into the USB core and the other facing the user processes.

The USB core interface is conventional for an internal Linux kernel API. It consists of registration and deregistration routines provided by the core, operations table that is passed to the core upon registration, and inline functions for hooks called by the core. It all comes down to the code shown in Figure 2.

As was mentioned previously, only one type of interface to user processes exists at present: text interface. It is implemented with the help of a pseudo filesystem called "debugfs" and consists of a few pseudo files, same group per every USB bus in the system. Text records are produced for every event, to be read from pseudo files. Their format is discussed below.

6 Use (user mode)

A common way to access usbmon without any special tools is as following:

```
# mount -t debugfs none /sys/kernel/debug
# modprobe usbmon
# cat /sys/kernel/debug/usbmon/3t
dfa105cc 1578069602 C Ii:001:01 0 1 D
dfa105cc 1578069644 S Ii:001:01 -115 2 D
d6bda284 1578069665 S Ci:001:00 -115 4 <
d6bda284 1578069672 C Ci:001:00 0 4 = 01010100
........
```

The number 3 in the file name is the number of the USB bus as reported by /proc/bus/usb/devices.

Each record copied by cat starts with a tag that is used to correlate various events happening to the same URB. The tag is simply a kernel address of the URB. Next words are: a timestamp in milliseconds, event type, a joint word for the type of transfer, device number, and endpoint number, I/O status, data size, data marker and a varying number of data words. More precise documentation exists within the kernel source code, in file Documentation/usb/usbmon.txt.

This format is terse, but it can be read by humans in a pinch. It is also useful for postings to mailing lists, Bugzilla attachments, and other similar forms of data exchange.

Tools to ease dealing with usbmon are being developed. Only one such tool exists today: the USBMon (written with upper case letters), originally written by David Harding. It is a tool with a graphical interface.

7 Lessons

When compared to tcpdump(8) or Ethereal(1), usbmon today is rudimentary. Despite that, in the short time it has existed, usbmon helped the author to quickly pinpoint several bugs that otherwise would take many kernel rebuilds and gaining an understanding of unfamiliar system

```
struct usb_mon_operations {
    void (*urb_submit)(struct usb_bus *bus, struct urb *urb);
    void (*urb_submit_error)(struct usb_bus *bus, struct urb *urb, int err);
    void (*urb_complete)(struct usb_bus *bus, struct urb *urb);
    void (*bus_add)(struct usb_bus *bus);
    void (*bus_remove)(struct usb_bus *bus);
};

extern struct usb_mon_operations *mon_ops;

static inline void usbmon_urb_submit(struct usb_bus *bus, struct urb *urb)
{
    if (bus→monitored)
        (*mon_ops→urb_submit)(bus, urb);
}

static inline void usbmon_notify_bus_remove(struct usb_bus *bus)
{
    if (mon_ops)
        (*mon_ops→bus_remove)(bus);
}

int usb_mon_register(struct usb_mon_operations *ops);
void usb_mon_deregister(void);
```

Figure 2: The interface to the USB core.

log messages. Having any sort of usable unified tracing is helpful when developers have to work with an explicitly programmed message passing bus.

A large part of usbmon's utility comes from being always enabled, which requires its overhead to be undetectable when inactive and low enough not to change the system's behaviour when active. So it probably is unreasonable to implement an equivalent of usbmon for PCI: performance overhead may be too great; the level of messages is too low; there are no standard protocols to be parsed by upper level tools. But developers of subsystems such as SCSI or Infiniband are likely to benefit from introduction of "scsimon" or "infinimon" into their set of tools.

8 Future Work

The usbmon and user mode tools have a long way to go before they can be as developed as tcpdump(8) is today. Below we list issues which are prominent now.

- When USB buses are added and removed, tools have to be notified, which is not done at present. As a workaround, tools rescan file `/proc/bus/usb/devices` periodically. A solution may be something as

simple as select(2) working on a special file.

- Users often observe records which should carry data, but do not. For example:

```
c07835cc 1579486375 S Co:002:00 -115 0
d2ac6054 1579521858 S Ii:002:02 -115 4 D
```

  In the first case, setup packet of a control transfer is not captured, and in the second case, data part of an interrupt transfer is missing. The 'D' marker means that, according to the flags in the URB, the data was not mapped into the kernel space, and was only available for the DMA. The code to handle these cases has yet to be developed.

- The raw text data is difficult to interpret for people. So, it is desirable to decode the output to higher level protocols: SCSI commands, HID reports, hub control messages. This task belongs to the tools such as USBMon.

- Some tool developers express preferences for a binary and versioned API to compliment the existing text-based interface to usbmon. These requests need to be addressed.

References

Van Jacobson et al. *tcpdump(8), the manual.* In tcpdump version 3.8.2, 2004.

Greg Kroah-Hartman. *kobjects and krefs.* In Proceedings of the Linux Symposium (former OLS) 2004.