General notification mechanism

The general notification mechanism is built on top of the standard pipe driver whereby it effectively splices notification messages from the kernel into pipes opened by userspace. This can be used in conjunction with:

```
* Key/keyring notifications
```

The notifications buffers can be enabled by:

"General setup"/"General notification queue" (CONFIG_WATCH_QUEUE)

This document has the following sections:

Overview

This facility appears as a pipe that is opened in a special mode. The pipe's internal ring buffer is used to hold messages that are generated by the kernel. These messages are then read out by read(). Splice and similar are disabled on such pipes due to them wanting to, under some circumstances, revert their additions to the ring - which might end up interleaved with notification messages.

The owner of the pipe has to tell the kernel which sources it would like to watch through that pipe. Only sources that have been connected to a pipe will insert messages into it. Note that a source may be bound to multiple pipes and insert messages into all of them simultaneously.

Filters may also be emplaced on a pipe so that certain source types and subevents can be ignored if they're not of interest.

A message will be discarded if there isn't a slot available in the ring or if no preallocated message buffer is available. In both of these cases, read() will insert a WATCH_META_LOSS_NOTIFICATION message into the output buffer after the last message currently in the buffer has been read.

Note that when producing a notification, the kernel does not wait for the consumers to collect it, but rather just continues on. This means that notifications can be generated whilst spinlocks are held and also protects the kernel from being held up indefinitely by a userspace malfunction.

Message Structure

Notification messages begin with a short header:

```
struct watch_notification {
        __u32   type:24;
        __u32   subtype:8;
        __u32   info;
};
```

"type" indicates the source of the notification record and "subtype" indicates the type of record from that source (see the Watch Sources section below). The type may also be "WATCH_TYPE_META". This is a special record type generated internally by the watch queue itself. There are two subtypes:

- WATCH_META_REMOVAL_NOTIFICATION
- WATCH_META_LOSS_NOTIFICATION

The first indicates that an object on which a watch was installed was removed or destroyed and the second indicates that some messages have been lost.

"info" indicates a bunch of things, including:

- The length of the message in bytes, including the header (mask with WATCH_INFO_LENGTH and shift by WATCH_INFO_LENGTH__SHIFT). This indicates the size of the record, which may be between 8 and 127 bytes.
- The watch ID (mask with WATCH_INFO_ID and shift by WATCH_INFO_ID__SHIFT). This indicates that caller's ID of the watch, which may be between 0 and 255. Multiple watches may share a queue, and this provides

a means to distinguish them.
- A type-specific field (WATCH_INFO_TYPE_INFO). This is set by the notification producer to indicate some meaning specific to the type and subtype.

Everything in info apart from the length can be used for filtering.

The header can be followed by supplementary information. The format of this is at the discretion is defined by the type and subtype.

Watch List (Notification Source) API

A "watch list" is a list of watchers that are subscribed to a source of notifications. A list may be attached to an object (say a key or a superblock) or may be global (say for device events). From a userspace perspective, a non-global watch list is typically referred to by reference to the object it belongs to (such as using KEYCTL_NOTIFY and giving it a key serial number to watch that specific key).

To manage a watch list, the following functions are provided:

- ```
  void init_watch_list(struct watch_list *wlist,
                       void (*release_watch)(struct watch *wlist));
  ```

  Initialise a watch list. If `release_watch` is not NULL, then this indicates a function that should be called when the watch_list object is destroyed to discard any references the watch list holds on the watched object.

- `void remove_watch_list(struct watch_list *wlist);`

  This removes all of the watches subscribed to a watch_list and frees them and then destroys the watch_list object itself.

Watch Queue (Notification Output) API

A "watch queue" is the buffer allocated by an application that notification records will be written into. The workings of this are hidden entirely inside of the pipe device driver, but it is necessary to gain a reference to it to set a watch. These can be managed with:

- `struct watch_queue *get_watch_queue(int fd);`

  Since watch queues are indicated to the kernel by the fd of the pipe that implements the buffer, userspace must hand that fd through a system call. This can be used to look up an opaque pointer to the watch queue from the system call.

- `void put_watch_queue(struct watch_queue *wqueue);`

  This discards the reference obtained from `get_watch_queue()`.

Watch Subscription API

A "watch" is a subscription on a watch list, indicating the watch queue, and thus the buffer, into which notification records should be written. The watch queue object may also carry filtering rules for that object, as set by userspace. Some parts of the watch struct can be set by the driver:

```
struct watch {
        union {
                u32             info_id;        /* ID to be OR'd in to info field */
                ...
        };
        void                    *private;       /* Private data for the watched object */
        u64                     id;             /* Internal identifier */
        ...
};
```

The `info_id` value should be an 8-bit number obtained from userspace and shifted by WATCH_INFO_ID__SHIFT. This is OR'd into the WATCH_INFO_ID field of struct watch_notification::info when and if the notification is written into the associated watch queue buffer.

The `private` field is the driver's data associated with the watch_list and is cleaned up by the `watch_list::release_watch()` method.

The `id` field is the source's ID. Notifications that are posted with a different ID are ignored.

The following functions are provided to manage watches:

- `void init_watch(struct watch *watch, struct watch_queue *wqueue);`

  Initialise a watch object, setting its pointer to the watch queue, using appropriate barriering to avoid lockdep complaints.

- `int add_watch_to_object(struct watch *watch, struct watch_list *wlist);`
```

Subscribe a watch to a watch list (notification source). The driver-settable fields in the watch struct must have been set before this is called.

- ```
  int remove_watch_from_object(struct watch_list *wlist,
                               struct watch_queue *wqueue,
                               u64 id, false);
  ```

  Remove a watch from a watch list, where the watch must match the specified watch queue (`wqueue`) and object identifier (`id`). A notification (`WATCH_META_REMOVAL_NOTIFICATION`) is sent to the watch queue to indicate that the watch got removed.

- `int remove_watch_from_object(struct watch_list *wlist, NULL, 0, true);`

  Remove all the watches from a watch list. It is expected that this will be called preparatory to destruction and that the watch list will be inaccessible to new watches by this point. A notification (`WATCH_META_REMOVAL_NOTIFICATION`) is sent to the watch queue of each subscribed watch to indicate that the watch got removed.

Notification Posting API

To post a notification to watch list so that the subscribed watches can see it, the following function should be used:

```
void post_watch_notification(struct watch_list *wlist,
                             struct watch_notification *n,
                             const struct cred *cred,
                             u64 id);
```

The notification should be preformatted and a pointer to the header (`n`) should be passed in. The notification may be larger than this and the size in units of buffer slots is noted in `n->info & WATCH_INFO_LENGTH`.

The `cred` struct indicates the credentials of the source (subject) and is passed to the LSMs, such as SELinux, to allow or suppress the recording of the note in each individual queue according to the credentials of that queue (object).

The `id` is the ID of the source object (such as the serial number on a key). Only watches that have the same ID set in them will see this notification.

Watch Sources

Any particular buffer can be fed from multiple sources. Sources include:

- WATCH_TYPE_KEY_NOTIFY

  Notifications of this type indicate changes to keys and keyrings, including the changes of keyring contents or the attributes of keys.

  See Documentation/security/keys/core.rst for more information.

Event Filtering

Once a watch queue has been created, a set of filters can be applied to limit the events that are received using:

```
struct watch_notification_filter filter = {
        ...
};
ioctl(fd, IOC_WATCH_QUEUE_SET_FILTER, &filter)
```

The filter description is a variable of type:

```
struct watch_notification_filter {
        __u32   nr_filters;
        __u32   __reserved;
        struct watch_notification_type_filter filters[];
};
```

Where "nr_filters" is the number of filters in filters[] and "__reserved" should be 0. The "filters" array has elements of the following type:

```
struct watch_notification_type_filter {
        __u32   type;
        __u32   info_filter;
        __u32   info_mask;
        __u32   subtype_filter[8];
};
```

Where:
```

- `type` is the event type to filter for and should be something like "WATCH_TYPE_KEY_NOTIFY"
- `info_filter` and `info_mask` act as a filter on the info field of the notification record. The notification is only written into the buffer if:

  ```
  (watch.info & info_mask) == info_filter
  ```

  This could be used, for example, to ignore events that are not exactly on the watched point in a mount tree.
- `subtype_filter` is a bitmask indicating the subtypes that are of interest. Bit 0 of subtype_filter[0] corresponds to subtype 0, bit 1 to subtype 1, and so on.

If the argument to the ioctl() is NULL, then the filters will be removed and all events from the watched sources will come through.

Userspace Code Example

A buffer is created with something like the following:

```
pipe2(fds, O_TMPFILE);
ioctl(fds[1], IOC_WATCH_QUEUE_SET_SIZE, 256);
```

It can then be set to receive keyring change notifications:

```
keyctl(KEYCTL_WATCH_KEY, KEY_SPEC_SESSION_KEYRING, fds[1], 0x01);
```

The notifications can then be consumed by something like the following:

```
static void consumer(int rfd, struct watch_queue_buffer *buf)
{
        unsigned char buffer[128];
        ssize_t buf_len;

        while (buf_len = read(rfd, buffer, sizeof(buffer)),
               buf_len > 0
               ) {
                void *p = buffer;
                void *end = buffer + buf_len;
                while (p < end) {
                        union {
                                struct watch_notification n;
                                unsigned char buf1[128];
                        } n;
                        size_t largest, len;

                        largest = end - p;
                        if (largest > 128)
                                largest = 128;
                        memcpy(&n, p, largest);

                        len = (n->info & WATCH_INFO_LENGTH) >>
                                WATCH_INFO_LENGTH__SHIFT;
                        if (len == 0 || len > largest)
                                return;

                        switch (n.n.type) {
                        case WATCH_TYPE_META:
                                got_meta(&n.n);
                        case WATCH_TYPE_KEY_NOTIFY:
                                saw_key_change(&n.n);
                                break;
                        }

                        p += len;
                }
        }
}
```