Internal input event handling in the Linux kernel and the Android userspace (https://seasonofcode.com/posts/internal-input-event-handling-in-the-linux-kernel-and-the-android-userspace.html)

By Chuan Ji | May 14, 2011 |
Tags: Android (https://seasonofcode.com/tag/android.html), Featured (https://seasonofcode.com/tag/featured.html) |
Comments (23) (https://seasonofcode.com/posts/internal-input-event-handling-in-the-linux-kernel-and-the-android-userspace.html#disqus_thread)

While figuring out hardware buttons for my NITDroid project, I had the opportunity of exploring the way Linux and Android handle input events internally before passing them through to the user application. This post traces the propagation of an input event from the Linux kernel through the Android userspace as far as I understand it. Although the principles are likely the same for essentially any input device, I will be drawing on my investigations of the drivers for the LM8323 hardware keyboard (`drivers/input/keyboard/lm8323.c`) and the TSC2005 touchscreen (`drivers/input/touchscreen/tsc2005.c`) which are both found inside the Nokia N810.

I. Inside the Linux kernel

Firstly, Linux exposes externally a uniform input event interface for each device as `/dev/input/eventX` where `X` is an integer. This means these "devices" can be polled in the same way and the events they produce are in the same uniform format. To accomplish this, Linux has a standard set of routines that every device driver uses to register / unregister the hardware it manages and publish input events it receives.

When the driver module of an input device is first loaded into the kernel, its initialization routine usually sets up some sort of probing to detect the presence of the types of hardware it is supposed to manage. This probing is of course device-specific; however, if it is successful, the module will eventually invoke the function `input_register_device(…)` in `include/linux/input.h` which sets up a file representing the physical device as `/dev/input/eventX` where `X` is some integer. The module will also register a function to handle IRQs originating from the hardware it manages via `request_irq(…)` (`include/linux/interrupt.h`) so that the module will be notified when the user interacts with the physical device it manages.

When the user physically interacts with the hardware (for instance by pushing / releasing a key or exerting / lifting pressure on the touchscreen), an IRQ is fired and Linux invokes the IRQ handler registered by the

corresponding device driver. However, IRQ handlers by custom must return quickly as they essentially block the entire system when executing and thus cannot perform any lengthy processing; typically, therefore, an IRQ handler would merely 1) save the data carried by the IRQ, 2) ask the kernel to schedule a method that would process the event later on when we have exited IRQ mode, and 3) tell the kernel we have handled the IRQ and exit. This could be very straightforward, as the IRQ handler in the driver for the LM8323 keyboard inside the N810:

```
/*
 * We cannot use I2C in interrupt context, so we just schedule work.
 */
static irqreturn_t lm8323_irq(int irq, void *data)
{
        struct lm8323_chip *lm = data;

        schedule_work(&lm->work);

        return IRQ_HANDLED;
}
```

It could also be more complex as the one in the driver of the TSC2005 touchscreen controller (`tsc2005_ts_irq_handler(…)`) as it integrates into the SPI framework (which I have never looked into...).

Some time later, the kernel executes the scheduled method to process the recently saved event. Invariably, this method would report the event in a standard format by calling one or more of the `input_*` functions in `include/linux/input.h` ; these include `input_event(…)` (general purpose), `input_report_key(…)` (for key down and key up events), `input_report_abs(…)` (for position events e.g. from a touchscreen) among others. Note that the `input_report_*(…)` functions are really just convenience functions that call `input_event(…)` internally, as defined in `include/linux/input.h` . It is likely that a lot of processing happens before the event is published via these methods; the LM8323 driver for instance does an internal key code mapping step and the TSC2005 driver goes through this crazy arithmetic involving Ohms (to calculate a pressure index from resistance data?). Furthermore, one physical IRQ could correspond to multiple published input events, and vice versa. Finally, when all event publishing is finished, the event processing method calls `input_sync(…)` to flush the event out. The event is now ready to be accessed by the userspace at `/dev/input/eventX` .

II. Inside the Android userspace

When the Android GUI starts up, an instance of the class `WindowManagerService` (`frameworks/base /services/java/com/android/server/WindowManagerservice.java`) is created. This class, when constructed, initializes the member field

```
final KeyQ mQueue;
```

where `KeyQ`, defined as a private class inside the same file, extends Android's basic input handling class, the abstract class `KeyInputQueue` (`frameworks/base/services/java/com/android/server/KeyInputQueue.java` and `frameworks/base/services/jni/com_android_server_KeyInputQueue.cpp`). As `mQueue` is instantiated, it of course calls the constructor of `KeyInputQueue`; the latter, inconspicuously, starts an anonymous thread it owns that is at the heart of the event handling system in Android:

```
Thread mThread = new Thread("InputDeviceReader") {
    public void run() {
        ...
        RawInputEvent ev = new RawInputEvent();
        while (true) {
            try {
                readEvent(ev);  // block, doesn't release the monitor

                boolean send = false;
                ...
                if (ev.type == RawInputEvent.EV_DEVICE_ADDED) {
                    ...
                } else if (ev.type == RawInputEvent.EV_DEVICE_REMOVED) {
                    ...
                } else {
                    di = getInputDevice(ev.deviceId);
                    ...
                    // first crack at it
                    send = preprocessEvent(di, ev);
                }
                ...
                if (!send) {
                    continue;
                }
                synchronized (mFirst) {
                    ...
                    // Is it a key event?
                    if (type == RawInputEvent.EV_KEY &&
                            (classes&RawInputEvent.CLASS_KEYBOARD) != 0 &&
                            (scancode < RawInputEvent.BTN_FIRST ||
                                    scancode > RawInputEvent.BTN_LAST)) {
                        boolean down;
                        if (ev.value != 0) {
                            down = true;
                            di.mKeyDownTime = curTime;
                        } else {
                            down = false;
                        }
                        int keycode = rotateKeyCodeLocked(ev.keycode);
                        addLocked(di, curTimeNano, ev.flags,
                                RawInputEvent.CLASS_KEYBOARD,
                                newKeyEvent(di, di.mKeyDownTime, curTime, down,
                                        keycode, 0, scancode,
                                        ((ev.flags & WindowManagerPolicy.FLAG_WOKE_HERE) !=
                                         ? KeyEvent.FLAG_WOKE_HERE : 0));
                    } else if (ev.type == RawInputEvent.EV_KEY) {
                        ...
                    } else if (ev.type == RawInputEvent.EV_ABS &&
                            (classes&RawInputEvent.CLASS_TOUCHSCREEN_MT) != 0) {
                        // Process position events from multitouch protocol.
                        ...
                    } else if (ev.type == RawInputEvent.EV_ABS &&
```

```
                            (classes&RawInputEvent.CLASS_TOUCHSCREEN) != 0) {
                        // Process position events from single touch protocol.
                        ...
                    } else if (ev.type == RawInputEvent.EV_REL &&
                            (classes&RawInputEvent.CLASS_TRACKBALL) != 0) {
                        // Process movement events from trackball (mouse) protocol.
                        ...
                    }
                    ...
                }

            } catch (RuntimeException exc) {
                Slog.e(TAG, "InputReaderThread uncaught exception", exc);
            }
        }
    }
};
```

I have removed most of this ~350 lined function that is irrelevant to our discussion and reformatted the code for easier reading. The key idea is that this independent thread will

1. Read an event
2. Call the `preprocess(…)` method of its derived class, offering the latter a chance to prevent the event from being propagated further

3.Add it to the event queue owned by the class

This `InputDeviceReader` thread started by `WindowManagerService` (indirectly via `KeyInputQueue's constructor`) is thus THE event loop of the Android UI.

But we are still missing the link from the kernel to this `InputDeviceReader`. What exactly is this magical `readEvent(…)`? It turns out that this is actually a native method implemented in the C++ half of `KeyInputQueue`：

```
static Mutex gLock;
static sp<EventHub> gHub;

static jboolean
android_server_KeyInputQueue_readEvent(JNIEnv* env, jobject clazz,
                                       jobject event)
{
    gLock.lock();
    sp<EventHub> hub = gHub;
    if (hub == NULL) {
        hub = new EventHub;
        gHub = hub;
    }
    gLock.unlock();

    ...
    bool res = hub->getEvent(&deviceId, &type, &scancode, &keycode,
            &flags, &value, &when);
    ...

    return res;
}
```

Ah, so `readEvent` is really just a proxy for `EventHub::getEvent(…)`. If we proceed to look up `EventHub` in `frameworks/base/libs/ui/EventHub.cpp`, we find

```
int EventHub::scan_dir(const char *dirname)
{
    ...
    dir = opendir(dirname);
    ...
    while((de = readdir(dir))) {
        ...
        open_device(devname);
    }
    closedir(dir);
    return 0;
}
...

static const char *device_path = "/dev/input";
...

bool EventHub::openPlatformInput(void)
{
    ...
    res = scan_dir(device_path);
    ...
    return true;
}

bool EventHub::getEvent(int32_t* outDeviceId, int32_t* outType,
        int32_t* outScancode, int32_t* outKeycode, uint32_t *outFlags,
        int32_t* outValue, nsecs_t* outWhen)
{
    ...
    if (!mOpened) {
        mError = openPlatformInput() ? NO_ERROR : UNKNOWN_ERROR;
        mOpened = true;
    }

    while(1) {
        // First, report any devices that had last been added/removed.
        if (mClosingDevices != NULL) {
            ...
            *outType = DEVICE_REMOVED;
            delete device;
            return true;
        }
        if (mOpeningDevices != NULL) {
            ...
            *outType = DEVICE_ADDED;
            return true;
        }

        ...
        pollres = poll(mFDs, mFDCount, -1);
        ...
```

```
        // mFDs[0] is used for inotify, so process regular events starting at mFDs[1]
        for(i = 1; i < mFDCount; i++) {
            if(mFDs[i].revents) {
                if(mFDs[i].revents & POLLIN) {
                    res = read(mFDs[i].fd, &iev, sizeof(iev));
                    if (res == sizeof(iev)) {
                        ...
                        *outType = iev.type;
                        *outScancode = iev.code;
                        if (iev.type == EV_KEY) {
                            err = mDevices[i]->layoutMap->map(iev.code, outKeycode, outFlag
                            ...
                        } else {
                            *outKeycode = iev.code;
                        }
                        ...
                        return true;
                    } else {
                        // Error handling
                        ...
                        continue;
                    }
                }
            }
        }
        ...
    }
}
```

Again, most of the details have been stripped out from the above code, but we now see how `readEvent()` in `KeyInputQueue` is getting these events from Linux: on first call, `EventHub::getEvent` scans the directory `/dev/input` for input devices, opens them and saves their file descriptors in an array called `mFDs`. Then whenever it is called again, it tries to read from each of these input devices by simply calling the `read(2)` (http://linux.die.net/man/2/read) Linux system call.

OK, now we know how an event propagates through `EventHub::getEvent(…)` to `KeyInputQueue::readEvent(…)` then to `InputDeviceReader.run(…)` where it could get queued inside `WindowManagerService.mQueue` (which, as a reminder, extends the otherwise abstract `KeyInputQueue`). But what happens then? How does that event get to the client application?

Well, it turns out that `WindowManagerService` has yet another private member class that handles just that:

```
private final class InputDispatcherThread extends Thread {
    ...
    @Override public void run() {
        while (true) {
            try {
                process();
            } catch (Exception e) {
                Slog.e(TAG, "Exception in input dispatcher", e);
            }
        }
    }

    private void process() {
        android.os.Process.setThreadPriority(
                android.os.Process.THREAD_PRIORITY_URGENT_DISPLAY);
        ...
        while (true) {
            ...
            // Retrieve next event, waiting only as long as the next
            // repeat timeout.  If the configuration has changed, then
            // don't wait at all -- we'll report the change as soon as
            // we have processed all events.
            QueuedEvent ev = mQueue.getEvent(
                (int)((!configChanged && curTime < nextKeyTime)
                        ? (nextKeyTime-curTime) : 0));
            ...
            try {
                if (ev != null) {
                    curTime = SystemClock.uptimeMillis();
                    int eventType;
                    if (ev.classType == RawInputEvent.CLASS_TOUCHSCREEN) {
                        eventType = eventType((MotionEvent)ev.event);
                    } else if (ev.classType == RawInputEvent.CLASS_KEYBOARD ||
                                ev.classType == RawInputEvent.CLASS_TRACKBALL) {
                        eventType = LocalPowerManager.BUTTON_EVENT;
                    } else {
                        eventType = LocalPowerManager.OTHER_EVENT;
                    }
                    ...
                    switch (ev.classType) {
                        case RawInputEvent.CLASS_KEYBOARD:
                            KeyEvent ke = (KeyEvent)ev.event;
                            if (ke.isDown()) {
                                lastKey = ke;
                                downTime = curTime;
                                keyRepeatCount = 0;
                                lastKeyTime = curTime;
                                nextKeyTime = lastKeyTime
                                        + ViewConfiguration.getLongPressTimeout();
                            } else {
                                lastKey = null;
                                downTime = 0;
```

```
                                // Arbitrary long timeout.
                                lastKeyTime = curTime;
                                nextKeyTime = curTime + LONG_WAIT;
                            }
                            dispatchKey((KeyEvent)ev.event, 0, 0);
                            mQueue.recycleEvent(ev);
                            break;
                        case RawInputEvent.CLASS_TOUCHSCREEN:
                            dispatchPointer(ev, (MotionEvent)ev.event, 0, 0);
                            break;
                        case RawInputEvent.CLASS_TRACKBALL:
                            dispatchTrackball(ev, (MotionEvent)ev.event, 0, 0);
                            break;
                        case RawInputEvent.CLASS_CONFIGURATION_CHANGED:
                            configChanged = true;
                            break;
                        default:
                            mQueue.recycleEvent(ev);
                            break;
                    }
                } else if (configChanged) {
                    ...
                } else if (lastKey != null) {
                    ...
                } else {
                    ...
                }
            } catch (Exception e) {
                Slog.e(TAG,
                    "Input thread received uncaught exception: " + e, e);
            }
        }
    }
}
```

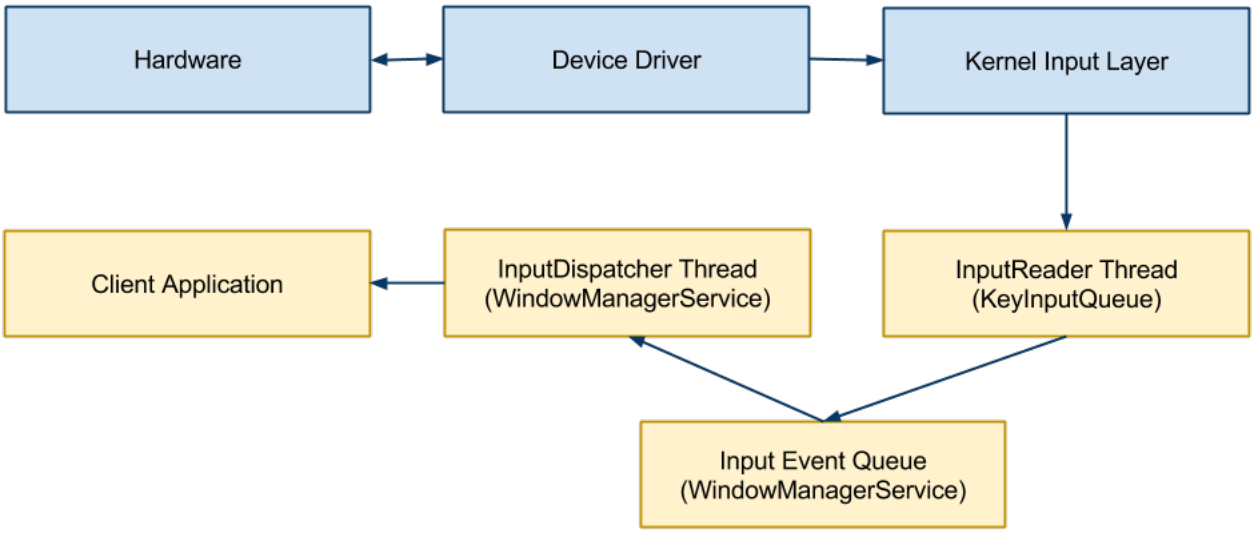
As we can see, this thread started by `WindowManagerService` is very simple; all it does is

1. Grabs events queued into `WindowManagerService.mQueue`
2. Calls `WindowManagerService.dispatchKey(…)` when appropriate.

If we next inspect `WindowManagerService.dispatchKey(…)`, we would see that it checks the currently focused window, and calls `android.view.IWindow.dispatchKey(…)` on that window. The event is now in the user space.

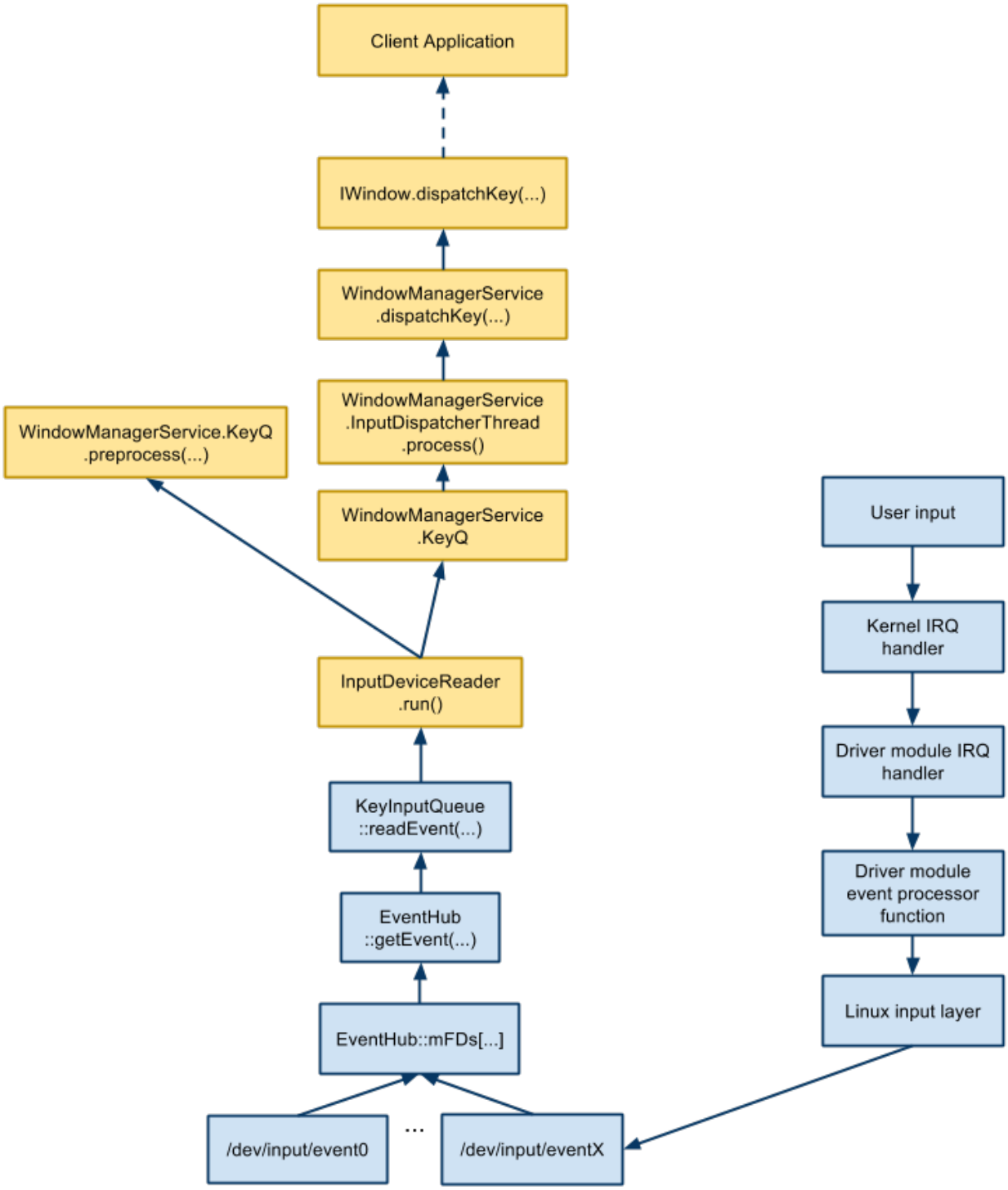
I put together some nice diagrams that illustrate these interactions. The conceptual model:

Event propagation flow on Android - simplified version



The full model:

Event propagation flow on Android



The yellow boxes are Java implementations, and the blue boxes native or other.

By Chuan Ji | May 14, 2011 |
Tags: Android (https://seasonofcode.com/tag/android.html), Featured (https://seasonofcode.com/tag/featured.html) |

Comments (23) (https://seasonofcode.com/posts/internal-input-event-handling-in-the-linux-kernel-and-the-android-userspace.html#disqus_thread)
|
Permalink (https://seasonofcode.com/posts/internal-input-event-handling-in-the-linux-kernel-and-the-android-userspace.html)

About the author

I am a software engineer by profession and a passionate technology geek in my free time.

jichu4n (https://github.com/jichu4n) @jichu4n (https://twitter.com/jichu4n) Chuan Ji (https://www.linkedin.com/in/chuanji) Chuan Ji (https://plus.google.com/115396580584561637180) ji@chu4n.com (mailto:ji@chu4n.com)

(https://plus.google.com/115396580584561637180)

Raghavendra • 4 years ago

Really informative description from top to the bottom. But I see that in the latest release of Android (mine is 4.2.2) whatever window manager service has done, it now done by InputReader.cpp.

1 • Reply • Share ›

manju reddy • 2 months ago

Hey Thanks for the good info. Do you know how to enable touch events for secondary display in android from eventHub.cpp ?. Since now the display is mirrored. Touch on both display works irrelevant of display selected. I want Primary Touch Display events to be handled separate and Second display touch events to be handled separate.

• Reply • Share ›

吴迪 • 3 months ago

Good article.
But the input event flow had changed a lot now since this article was written on 2011.
For the new input event system, a good pdf here
http://newandroidbook.com/f...

• Reply • Share ›

ran • a year ago

Great work, Now,
I'm trying to understand how to turn off display. On doing input keyevent 26, it turns off display, but I don't understand what low level device is called. I need to do it in low level somehow. /sys/class/...
Thanks for any idea

• Reply • Share ›

Nachiappan Chidambaram N • 4 years ago

Wonderful work! I appreciate it.

From the above it's quite clear what happens with input devices, i.e. when an user gives an input, it interrupts the processor through the flow shown in this post. Do you have any pointers/ideas regd. what happens w.r.t. output devices like display, audio, etc ?

• Reply • Share ›

Guest • 4 years ago