A Brief Overview of the Re-Implementation of the Android Push Event Stream Model

Dustin McAfee

1 Introduction

The work described, here, is an ongoing attempt of a re-implementation of an existing model, the Android Event Push Stream Model (ESM), put forth in a paper by Dr. Vander Zanden and Dr. Marz at the University of Tennessee, Knoxville, [1], and described in another paper submitted to the Journal of Mobile Devices by the same authors [2], and even further in Dr. Marz PhD dissertation, titled, "Reducing Power Consumption and Latency in Mobile Devices using a Push Event Stream Model, Kernel Display Server, and GUI Scheduler" [3]. This re-implementation is without access to the original code-base created by Dr. Marz, and is not an attempt to re-implement neither the Kernel Display Server as described in [2] and [3], nor the GUI Scheduler in [3].

1.1 Related Work

All of the relevant related work pertaining to implementing a new push event stream model by modifying the Linux kernel and Android operating system is introduced in the three papers mentioned in the above Section (Section 1). This re-implementation is originally goaled toward following the algorithms described in [1], including the functionality of the kernel-level functions "esm_wait", "esm_register", "esm_interpret", and "esm_dispatch", along with all of the data structures that are described in these functions. The other two papers, [2] and [3], describe how this model, implemented with the Kernel Display Server (and GUI Scheduler in [3]), is a more power efficient model and reduces latency for mobile devices.

1.2 Structure

In the next Section (Section 2), the basic functionality of the Push Event Stream Model (ESM) is compared to the current poll model that is used in modern devices, and the data structures used for the ESM are introduced. Section 2.1 describes the first attempt in implementing this model in detail, which involves using architecture-dependent assembly code to load the user-stack with a user-defined input handler, and context-switching from the bottom half of an interrupt handler to the user's input handler. Section 2.2 describes a simpler method of copying the events to user-space, along with an user-defined input handler identifier, so that the kernel is not responsible for using architecture-dependent code to execute the user's input handler. Section **??** describes the current progress and obstacles of this re-implementation.

2 Methods

The Android input stack is comprised of multiple layers for polling input and demultiplexing it to the user processes (applications). The polling interface to the Linux kernel that the Android operating system uses is epoll (see frameworks/native/services/inputflinger/EventHub.cpp in the Android source code, and fs/eventpoll.c in the Linux kernel source code). The Linux kernel running inside of Android uses the evdev input event interface (see drivers/input/evdev.c in the Linux kernel source code). After an input event fires an interrupt in the Linux kernel, evdev is responsible for directing the input event to the correct input device file (located in /dev/input/).

The Android operating system is using epoll to consistently poll all input devices that are of any interest to the system (input devices that have been registered by a user application or system application). Only one epoll instance is used to poll the input devices, so the goal is to be able to replace the one instance of epoll and its call to "epoll_wait" and calls to "epoll_register" to a new ESM model API, with calls to "esm_wait" and "esm_register". A big obstacle to note, is that the epoll API uses file descriptors to not only identify the input devices, but also identify itself. In the ESM, the input event is being streamed from the bottom half of an interrupt handler (deferred work) to the user, possibly even before it is being written to the input device file.

Dr. Marz describes in [1] how "esm_wait", "esm_register", "esm_interpret", and "esm_dispatch" are implemented, along with a mapping from the user-defined input handler and the input events that are registered. The mapping differs between Sections 2.1 and 2.2, and so the mapping from Section 2.1 will be discussed here, and the differences will be explained in the later Section.

A 'struct application_l' (application list item) structure is created for each input event type and code, which contains the input handler virtual address that is registered. The structures that share the same input type and code with different input handlers are shared in the same linked list, as to organize the lists by the type/code of the input event. So that, when "esm_register" is called, a new 'application_l' is allocated and added to the appropriate linked list, or if "esm_register" is called with the argument to de-register an input handler, then the appropriate 'appliation_l' is de-allocated and removed from the linked list.

The idea is that the user application can register an input type to a given input handler address, so that after it calls ”esm_wait”, it will receive these events without having to poll a device file. So, in order to record registered events that happen before the user calls ”esm_wait”, another linked list is added to the process control block structure, ’struct task_struct’ (see linux/sched.h). The linked list structure added, simply contains a list of input values that are registered to the task and pending to the ESM model. This list must act as a queue in order to push the events to the user in order.

One more major addition to the Linux kernel is a new task state, ’TASK_EV_WAIT’, as described in [1]. This is to ensure that the system knows which tasks are waiting for events using ESM, so that, any call to wake_up’ (or, specifically, ’wake_up_state’) must be given the argument ’TASK_EV_WAIT’ in order to wake the process up, notifying it that there are input events in its queue. In Dr. Marz’s implementation of the GUI Scheduler, dynamic clock tics are set so that when the system is doing nothing but waiting on input events (no wake-locks in use), the clock timer will not interrupt and wake up any processes that otherwise should be asleep waiting for events [3]. This is not a part of the re-implementation, as of yet.

After an interrupt occurs in the Linux kernel for an input event, it is passed to evdev. A call to ”esm_interpret” is used here to intercept this event just before it is passed to be written to the character device. In this re-implementation, ”esm_interpret” iterates through each ’application_l’ type to find the corresponding linked list that contains the registered input event and event handlers, and creates deferred work (work queue) that calls ”esm_dispatch” for each input handler interested in the input event.

A call to ”esm_dispatch” will enqueue the input event if the task is not in ’TASK_EV_WAIT’, otherwise, it is responsible for pushing the event into userspace. The original implementation by Dr. Marz uses a combination of the Kernel Display Server and the Android Push Event Stream Model to not only push the event into userspace, but also context switch from the kernel to the user process’s registered input handler, somehow allowing the process to return back into the call to ”esm_wait” after the event was handled. Since this original code-base is not open source, this part of the re-implementation is where most of the complications come from. In fact, Section 2.2 is a simpler model that strays away from this structure a little in order to simplify the way that the input handler is called.

2.1 ESM Load User Stack

The first attempt in re-implementing ESM in the Linux kernel involved registering single events to the kernel and loading the user process’s stack with its own input handler and the value of the triggered input event. So, for each interested input event (left mouse click, right mouse click, specific keyboard buttons, etc.), the user process calls ”esm_register” with the virtual address of the input handler (which belongs to the user process) that should handle the event. This creates an item in a linked list that contains the virtual address of the input handler, the interested input event, and a pointer to the process control block that is interested in the input event (and owns the input handler).

After registering for events, the user process calls ”esm_wait”, which puts the process into the new task state ’TASK_EV_WAIT’ and schedules it. When an interested event occurs (via interrupt), ”esm_interpret” is called, which calls ”esm_dispatch” for each interested event. ”esm_dispatch” is responsible for dispatching the input event to the input handler (taking the process out of ’TASK_EV_WAIT’), and in this case, calls a special architecture dependent context-switch function that loads the user process’s stack with its own input handler and input event and switches the context to that user process. This architecture dependent code was only written for x86/x86_64, and never properly worked; there are protections set in place by either the hardware or the kernel from switching out of kernel mode to user mode that halted the kernel when attempting this method.

However, the main reason this was abandoned, is because it required re-writing assembly code for x86 and arm devices (32-bit and 64-bit). Another reason to try a different approach, is that the epoll API does not call the input handler from kernel space, and in fact, Android relies on its middleware to read that input events after epoll returns, and figure out what to do with the input events thereafter. This is a big reason (I think) why Dr. Marz moved much of the Android middleware to the kernel when developing the Kernel Display Server [2]; it is because the Android middleware actually has to re-figure out what to do with input events after receiving them from the kernel, and because of this, much of the kernel input driver code is re-written inside of the Android HAL (Hardware Abstraction Layer).

2.2 ESM Copy to User

Without having to re-implement Dr. Marz’s KDS, it does not make much sense to register single events at a time to the user process, or even call the input event handler from kernel space. In fact, the goal becomes much simpler: Remove the call to epoll from the Android inputflinger and replace it with a similar API that does not poll. This means registering groups of input events based on input device file descriptors, and somehow delivering the interested events to the user after a call to ”esm_wait”. The main goal of removing the polling loop voids the point of waiting on character device files, and so this means there will be no writing to, or reading from character devices inside of the ESM. However, epoll uses open input device file descriptors to register interested input events to the user process. To overcome this, ”esm_register” takes a user buffer of a ’struct input_id’, which is a unique identifier of an input device. This input ID structure consolidates the ’struct application_l’ type (Section 2) to a single linked-list of input IDs and Process Control Block pointers. It is known in the Linux kernel upon an interrupt which device generated the input event, which is passed to ”esm_interpret” (from evdev). In this case, ”esm_dispatch” actually just wakes the process up (if it is in ’TASK_EV_WAIT’, scheduled in ”esm_wait”) and ”esm_wait” copies the events to userspace (to a ’void __user*’ supplied buffer argument) before returning.

For a test case, a single thread user application is developed that uses the ESM API to wait on, collect, and deliver

input events to corresponding handlers. The ESM successfully delivers multiple events without having read from any open file descriptors (however, it does have to make calls to ioctl to read the input id's). Successful tests were performed on this single threaded program for collecting events before and after the call to "esm_wait". However, opening the input character device required root access, and the Android device needed to be rooted to do so (this is not a problem since the intent is to implement this into the middleware, which runs with enough permission to read from the input devices).

The single thread case is a success, but the Android inputflinger service is multi-threaded, with one thread that registers/de-registers input devices, and another dedicated thread that runs "EventHub::GetEvents", which is the function that calls "epoll_wait".

2.3 Implementation in the Android Middleware

When the ESM interface is installed into EventHub, the events get registered to a thread separate from the thread that calls "esm_wait" (separate threads do not share Process Control Blocks in the Linux kernel); the thread that waits for events in "EventHub::GetEvents" goes to sleep and never has any events registered to its Process Control Block. Therefore, it never wakes up when any events happen.

To solve this issue, a new system call is created, named "esm_ctl". This function is given two process IDs, and is called right before "esm_wait". It transfers the thread's registered input devices and queued input values to another thread, allowing the callee of "esm_wait" to have the correct registered and queued events.

However, to completely replace "EventHub"'s dependency on the epoll interface, another argument must be supplied to "esm_register" and "esm_wait". The epoll interface uses what is defined as a 'struct epoll_event', which is mainly used by epoll to mask events from file-descriptors, along with error reporting, and for the user to store device specific data. "EventHub" sets the 'events' member to "EPOLLIN", which sets the input event mask to take all input events from the input device. When there is an error such as a hang-up event in epoll, then epoll is liable to change this value to indicate the error; however, this has not been implemented in this push model as of yet.

One other member of the 'epoll_event' structure is used by the Android middleware, which is in the 'data' field (of type 'epoll_data_t'). This field holds a value that is used to index an array of input device structures defined by the middleware. So, it is important to add an association of the supplied 'struct epoll_event' to the supplied 'struct input_id' (Section 2.2) inside of the ESM API, specifically, inside of 'struct application_l' (See Section 2).

One minor discrepancy between the ESM API and the epoll API used by the Android middleware is that the ESM API uses 'struct input_value' structures, and the epoll API uses 'struct input_event' structures. The difference is that the 'struct input_event' structures contain a timestamp member. This can be easily remedied by replacing all of the 'struct input_value' instances in the ESM API with 'struct input_event' instances, and including the timestamp in the 'struct input_event' when invoking "esm_interpret" from the evdev interface. Because of this, the Android middleware (specifically inputflinger) spits out warnings about the wrong clock used for the input devices when input arrives.

3 Results

Using this new API, all calls to the epoll interface are replaced with calls to the ESM interface in an AOSP 7.1 build, and tested in an x86_64 emulator. There are no inputflinger process crashes that are seen in the logs, and the input is delivered in a timely manner.

There is one issue that has yet to be addressed. During streaming of input events in the Linux kernel, a special kind of event (with type, code, and value equal to 0) is generated to mark the end of an event. This is called an "EV_SYN" event, which is used as a marker to separate events in time or space (e.g. multi-touch protocol). These events are streamed to the Android middleware, which expects the Linux kernel to have already handled these events. The result is the input events are a little choppy, with the USB keyboard (or hardware keyboard connected to the AOSP emulator on a laptop) having the most issues with single key presses acting as multiple key presses. This issue could likely be solved by moving the invocation to "esm_interpret" to a later function in the Linux input stack, where the synchronization events have already been handled, but may not be as trivial to solve as the previous issues that have been discussed. This implies that the location of the interception of events for the ESM model may be a little too early, and could benefit much by having the Linux input stack process the events further before passing them to user space.

One major thing worth noting to end this discussion, is that the source code for the ESM API likely contains artifacts from the previous sections of this paper that were never re-written or removed. A fine example of this is that the input event queue that is stored in the Process Control Block would likely be able to be taken out and put into a global event queue, since the Android middleware is only using the one instance of ESM. This would make the "esm_ctl" call obsolete, as well. However, keeping the input event queues stored inside of the PCB allows for multiple instances of the ESM for different processes (or threads) in the system.

References

[1] S. Marz and B. V. Zanden, "Reducing power consumption and latency in mobile devices using an event stream model," *ACM Trans. Embed. Comput. Syst.*, vol. 16, pp. 11:1–11:24, Oct. 2016.

[2] S. Marz, B. T. Vander Zanden, and W. Gao, "Reducing event latency and power consumption in mobile devices by using a kernel-level display server," *IEEE Transactions on Mobile Computing*, pp. 1–1, 2018.

[3] S. Marz, "Reducing power consumption and latency in mobile devices by using a gui scheduler," 2018.