Kprobe Assignment

Date Due: March 30, 2008

The goal of this assignment is to build a Linux AT/PS2 scancode logging module. The module will use the kprobe instrumentation mechanism to collect scancodes and export them using debugfs to be read by a user application. What the application does with the scancodes is outside the scope of the assignment. Presumably it logs them to disk or may take some other action.

For this description, we will use the in-kernel data structure terms to avoid any confusion. The VFS term translation is as follows: inode = file on disk, dentry = name of file or directory, file = open file descriptor.

Your module must:

- Use the `kprobe`s instrumentation mechanism. `kprobe`, `jprobe`, or `kretprobe` are acceptable.
- Buffer 32 scancodes in a ring buffer. Please overflow at this point, I don't want to type all day or run arbitrary qemu scripts.
- Export your buffer as an inode available through the dentry `"atkbd/scancodes"` within the root of the debugfs filesystem.
- When an application `open`s the `scancodes` inode, the first (index: zero) byte of the newly opened file should be the oldest byte remaining in ring buffer. Bytes which were not written by logging scancodes should not appear here.
- When an application attempts to `read` a byte which is no longer in the buffer, that byte will be read as a zero. (The zero scancode indicates a fatal keyboard error, and is a reasonable result. There are better choices, but this is a bit more legible.)
- Applications do **not** evict bytes from the scancode buffer by reading them. This is **not** a producer/consumer model.
- Any number of applications must be able to `open` the `scancodes` inode, obtaining different open files, each with their own offset into the buffer. This should work for the same application opening the inode multiple times.
- Applications must have a method of waiting for new scancodes to arrive and read them.
- Do **not** worry about shared open files. The Unix specification requires that applications sharing open file descriptors manage their own synchronization. However, individual open files must operate independently.
- **Do** support 64-bit file offsets.
- Do **not** worry about overflowing 64-bit indexes. Assume that less than 2^{63} characters will be typed.
- **Do** allow applications to `seek` in the file.
- **Do** clean up any dynamic memory you have allocated, if any, at `open` during `close`.
- Assume you are running on a single-processor system. You may implement a SMP mechanism if so inclined.
- Do **not** worry about spurious keyboards. Assume that the keyboard and interrupt system are working correctly.
- Do **not** over-engineer this problem. In particular, kernel top-half and bottom-half drivers are unnecessary and will not be graded.

Building your kernel module on cs418:

On CS418, copy the directory /home/kprobe/build to your home directory using:

```
$ cp -r /home/kprobe/build ~/
```

You may wish to commit this to a version control system at this point. We recommend mercurial

$ hg init && hg commit -m “initial import”

The file `atkbd_scancodes.c` is a Linux kernel module skeleton which you can edit. To create the kernel module, run make.

```
$ make
```

To import your kernel module into qemu type:

```
$ make qemu
```

This will create a new qemu disk image for you to run, generate a modules disk image loaded with your module and some test programs, and launch qemu.

The root password to the image is “kprobes”.

Your kernel modules are installed in /mnt/modules. To install your kernel module type:

insmod /mnt/modules/atkbd_scancodes.ko

To remove it type:

rmmod atkbd_scancodes

Building your kernel module on another computer:

Simply copy the /home/kprobe directory to another computer. You will need to specify the full path of the local kprobe directory in the Makefile by changing the KPROBE_HOME variable.

You will need approximately 11 Gb of storage and Linux running on x86. We will not support cross compiles. You will need qemu and most of the available Linux development tools.

REMEMBER TO TEST YOUR FINAL SUBMISSION ON CS418. All submissions will be graded on cs418 and must compile and install to receive credit.

We do *NOT* recommend reloading new kernel module versions without rebooting. Loading a newer module version into a running system isn't a problem. However, there is no guarantee that the old, and by definition buggy, driver has not corrupted the system in some yet unnoticed way. In this case, it is more likely that you will be simultaneously debugging two interfering modules instead of one.

Be very careful about using any networking on these systems, including SSH. If you must load new modules into your running system, we would prefer that you do not use SSH. *Instead, please unmount /mnt/modules, rebuild, and remount /mnt/modules.* This will recreate your modules drive.

Do not leave any important files in the qemu guest system. Any changes to the qemu system will be removed when performing a `make clean` or `distclean`. Your modules image will be recreated whenever running qemu. If you want to have files available in the guest, place them into the "my-modules" directory. The default modules filesystem is about 80MB and should have plenty of room for anything. You can change this size in the Makefile by altering the MODULES_BLOCKS variable.

Two applications have been provided for your use. **These are the applications we will use to test your module.** The `reader` application is nearly identical to `tail -f`, insofar as it sequentially reads bytes from a file and pumps them to `stdout`. However, it ignores the reported file length and does not attempt to manage truncated files. It can be used in place of a scancode logging program. If no file is specified it uses `stdin`.

The `tester` program is identical to the reader, except it outputs human readable messages instead of raw bytes. Additionally, it waits for a character from `stdin` before reading from the file. Each call to `read` will report how many bytes were read followed by reports of each byte in character, hexadecimal, and decimal formats. It requires a file and does not fall back to `stdin.`

Hints:

You will need to probe some instruction from drivers/input/keyboard/atkbd.c .

Since you are instrumenting an interrupt handler, you will not be able to use any mutex when writing to your buffer. You will need to implement a transaction. We recommend keeping a counter for the number of scancodes read to your buffer. Before the kernel responds to a read request, it will copy this counter to a volatile location. It then performs the string copy, and when finished, checks to see if the counter has changed mid-copy. If so, it simply re-executes the entire transaction. During a write to the buffer, increment the counter. Since interrupts are disabled, writes happen during a critical section on a uniprocessor.

To get this to work for SMP, you would need to keep a transaction boolean that indicates you are changing data. This is easily implemented using the least significant bit of the counter, and incrementing it twice: once before writing, and once after. A thread reading will be able to tell whether the buffer has changed, or is changing, and rollback the transaction. The only difference in read logic is to mask the least significant bit after reading the counter.

Why don't reading threads starve? A keyboard has a minimum key repeat delay of 250ms, which is far faster than anyone can type. Memory operates at speeds in tens of ns, fast enough to read a handful of scancodes between key events. Of course, this assumes that escape code sequences fit into this 250 ms boundary, or don't overflow the buffer entirely.

Remember that scancodes produce a down code and an up code, so there will be at least two codes for every key press. If a key is held, the repeat will kick on and the keyboard will actively describe what is happening.

This assignment will not require much written code, but will be heavy in development and reading. You will need to read the VFS chapter of Understanding the Linux Kernel. You will want to read a bit about using debugfs [1][2] and you may want to lookup an AT keyboard scancode chart [3] to debug your output. You can learn more about kprobes by reading the Linux documentation[4].

As with any project, please following good bug-reporting guidelines. [5]

Links:

[1] http://lwn.net/Articles/115405/
[2] http://cs418.cs.jhu.edu:8080/source/xref/include/linux/debugfs.h
[3] http://www.quadibloc.com/comp/scan.htm
[4] /home/kprobe/linux-2.6.23.15/Documentation/kprobes.txt on cs418.
[5] http://developer.mozilla.org/en/docs/Bug_writing_guidelines