# Exhibit A

# KS2009: How Google uses Linux

By **Jonathan Corbet**
October 21, 2009

LWN's 2009 Kernel Summit coverage

There may be no single organization which runs more Linux systems than Google. But the kernel development community knows little about how Google uses Linux and what sort of problems are encountered there. Google's Mike Waychison traveled to Tokyo to help shed some light on this situation; the result was an interesting view on what it takes to run Linux in this extremely demanding setting.

Mike started the talk by giving the developers a good laugh: it seems that Google manages its kernel code with Perforce. He apologized for that. There is a single tree that all developers commit to. About every 17 months, Google rebases its work to a current mainline release; what follows is a long struggle to make everything work again. Once that's done, internal "feature" releases happen about every six months.

This way of doing things is far from ideal; it means that Google lags far behind the mainline and has a hard time talking with the kernel development community about its problems.

There are about 30 engineers working on Google's kernel. Currently they tend to check their changes into the tree, then forget about them for the next 18 months. This leads to some real maintenance issues; developers often have little idea of what's actually in Google's tree until it breaks.

And there's a lot in that tree. Google started with the 2.4.18 kernel - but they patched over 2000 files, inserting 492,000 lines of code. Among other things, they backported 64-bit support into that kernel. Eventually they moved to 2.6.11, primarily because they needed SATA support. A 2.6.18-based kernel followed, and they are now working on preparing a 2.6.26-based kernel for deployment in the near future. They are currently carrying 1208 patches to 2.6.26, inserting almost 300,000 lines of code. Roughly 25% of those patches, Mike estimates, are backports of newer features.

There are plans to change all of this; Google's kernel group is trying to get to a point where they can work better with the kernel community. They're moving to git for source code management, and developers will maintain their changes in their own trees. Those trees will be rebased to mainline kernel releases every quarter; that should, it is hoped, motivate developers to make their code more maintainable and more closely aligned with the upstream kernel.

Linus asked: why aren't these patches upstream? Is it because Google is embarrassed by them, or is it secret stuff that they don't want to disclose, or is it a matter of internal process problems? The answer was simply "yes." Some of this code is ugly stuff which has been carried forward from the 2.4.18 kernel. There are also doubts internally about how much of this stuff will be actually useful to the rest of the world. But, perhaps, maybe about half of this code could be upstreamed eventually.

As much as 3/4 of Google's code consists of changes to the core kernel; device support is a relatively small

part of the total.

Google has a number of "pain points" which make working with the community harder. Keeping up with the upstream kernel is hard - it simply moves too fast. There is also a real problem with developers posting a patch, then being asked to rework it in a way which turns it into a much larger project. Alan Cox had a simple response to that one: people will always ask for more, but sometimes the right thing to do is to simply tell them "no."

In the area of CPU scheduling, Google found the move to the completely fair scheduler to be painful. In fact, it was such a problem that they finally forward-ported the old O(1) scheduler and can run it in 2.6.26. Changes in the semantics of sched_yield() created grief, especially with the user-space locking that Google uses. High-priority threads can make a mess of load balancing, even if they run for very short periods of time. And load balancing matters: Google runs something like 5000 threads on systems with 16-32 cores.

On the memory management side, newer kernels changed the management of dirty bits, leading to overly aggressive writeout. The system could easily get into a situation where lots of small I/O operations generated by kswapd would fill the request queues, starving other writeback; this particular problem should be fixed by the per-BDI writeback changes in 2.6.32.

As noted above, Google runs systems with lots of threads - not an uncommon mode of operation in general. One thing they found is that sending signals to a large thread group can lead to a lot of run queue lock contention. They also have trouble with contention for the `mmap_sem` semaphore; one sleeping reader can block a writer which, in turn, blocks other readers, bringing the whole thing to a halt. The kernel needs to be fixed to not wait for I/O with that semaphore held.

Google makes a lot of use of the out-of-memory (OOM) killer to pare back overloaded systems. That can create trouble, though, when processes holding mutexes encounter the OOM killer. Mike wonders why the kernel tries so hard, rather than just failing allocation requests when memory gets too tight.

So what is Google doing with all that code in the kernel? They try very hard to get the most out of every machine they have, so they cram a lot of work onto each. This work is segmented into three classes: "latency sensitive," which gets short-term resource guarantees, "production batch" which has guarantees over longer periods, and "best effort" which gets no guarantees at all. This separation of classes is done partly through the separation of each machine into a large number of fake "NUMA nodes." Specific jobs are then assigned to one or more of those nodes. One thing added by Google is "NUMA-aware VFS LRUs" - virtual memory management which focuses on specific NUMA nodes. Nick Piggin remarked that he has been working on something like that and would have liked to have seen Google's code.

There is a special SCHED_GIDLE scheduling class which is a truly idle class; if there is no spare CPU available, jobs in that class will not run at all. To avoid priority inversion problems, SCHED_GIDLE processes have their priority temporarily increased whenever they sleep in the kernel (but not if they are preempted in user space). Networking is managed with the HTB queueing discipline, augmented with a bunch of bandwidth control logic. For disks, they are working on proportional I/O scheduling.

Beyond that, a lot of Google's code is there for monitoring. They monitor all disk and network traffic, record it, and use it for analyzing their operations later on. Hooks have been added to let them associate all disk I/O back to applications - including asynchronous writeback I/O. Mike was asked if they could use tracepoints for this task; the answer was "yes," but, naturally enough, Google is using its own scheme now.

Google has a lot of important goals for 2010; they include:

- They are excited about CPU limits; these are intended to give priority access to latency-sensitive

tasks while still keeping those tasks from taking over the system entirely.

- RPC-aware CPU scheduling; this involves inspection of incoming RPC traffic to determine which process will wake up in response and how important that wakeup is.

- A related initiative is delayed scheduling. For most threads, latency is not all that important. But the kernel tries to run them immediately when RPC messages come in; these messages tend not to be evenly distributed across CPUs, leading to serious load balancing problems. So threads can be tagged for delayed scheduling; when a wakeup arrives, they are not immediately put onto the run queue. Instead, the wait until the next global load balancing operation before becoming truly runnable.

- Idle cycle injection: high-bandwidth power management so they can run their machines right on the edge of melting down - but not beyond.

- Better memory controllers are on the list, including accounting for kernel memory use.

- "Offline memory." Mike noted that it is increasingly hard to buy memory which actually works, especially if you want to go cheap. So they need to be able to set bad pages aside. The HWPOISON work may help them in this area.

- They need dynamic huge pages, which can be assembled and broken down on demand.

- On the networking side, there is a desire to improve support for receive-side scaling - directing incoming traffic to specific queues. They need to be able to account for software interrupt time and attribute it to specific tasks - networking processing can often involve large amounts of softirq processing. They've been working on better congestion control; the algorithms they have come up with are "not Internet safe" but work well in the data center. And "TCP pacing" slows down outgoing traffic to avoid overloading switches.

- For storage, there is a lot of interest in reducing block-layer overhead so it can keep up with high-speed flash. Using flash for disk acceleration in the block layer is on the list. They're looking at in-kernel flash translation layers, though it was suggested that it might be better to handle that logic directly in the filesystem.

Mike concluded with a couple of "interesting problems." One of those is that Google would like a way to pin filesystem metadata in memory. The problem here is being able to bound the time required to service I/O requests. The time required to read a block from disk is known, but if the relevant metadata is not in memory, more than one disk I/O operation may be required. That slows things down in undesirable ways. Google is currently getting around this by reading file data directly from raw disk devices in user space, but they would like to stop doing that.

The other problem was lowering the system call overhead for providing caching advice (with `fadvise()`) to the kernel. It's not clear exactly what the problem was here.

All told, it was seen as one of the more successful sessions, with the kernel community learning a lot about one of its biggest customers. If Google's plans to become more community-oriented come to fruition, the result should be a better kernel for all.

Next: Performance regressions

---