Linux Block IO—present and future

Jens Axboe
SuSE
`axboe@suse.de`

Abstract

One of the primary focus points of 2.5 was fixing up the bit rotting block layer, and as a result 2.6 now sports a brand new implementation of basically anything that has to do with passing IO around in the kernel, from producer to disk driver. The talk will feature an in-depth look at the IO core system in 2.6 comparing to 2.4, looking at performance, flexibility, and added functionality. The rewrite of the IO scheduler API and the new IO schedulers will get a fair treatment as well.

No 2.6 talk would be complete without 2.7 speculations, so I shall try to predict what changes the future holds for the world of Linux block I/O.

1 2.4 Problems

One of the most widely criticized pieces of code in the 2.4 kernels is, without a doubt, the block layer. It's bit rotted heavily and lacks various features or facilities that modern hardware craves. This has led to many evils, ranging from code duplication in drivers to massive patching of block layer internals in vendor kernels. As a result, vendor trees can easily be considered forks of the 2.4 kernel with respect to the block layer code, with all of the problems that this fact brings with it: 2.4 block layer code base may as well be considered dead, no one develops against it. Hardware vendor drivers include many nasty hacks and `#ifdef's` to work in all of the various 2.4 kernels that are out there, which doesn't exactly enhance code coverage or peer review.

The block layer fork didn't just happen for the fun of it of course, it was a direct result of the various problem observed. Some of these are added features, others are deeper rewrites attempting to solve scalability problems with the block layer core or IO scheduler. In the next sections I will attempt to highlight specific problems in these areas.

1.1 IO Scheduler

The main 2.4 IO scheduler is called `elevator_linus`, named after the benevolent kernel dictator to credit him for some of the ideas used. `elevator_linus` is a one-way scan elevator that always scans in the direction of increasing LBA. It manages latency problems by assigning sequence numbers to new requests, denoting how many new requests (either merges or inserts) may pass this one. The latency value is dependent on data direction, smaller for reads than for writes. Internally, `elevator_linus` uses a double linked list structure (the kernels `struct list_head`) to manage the request structures. When queuing a new IO unit with the IO scheduler, the list is walked to find a suitable insertion (or merge) point yielding an **O(N)** runtime. That in itself is suboptimal in presence of large amounts of IO and to make matters even worse, we repeat this scan if the request free list was empty when we entered the IO scheduler. The latter is not an error condition, it will happen all the time for even moderate amounts of write back against a queue.

1.2 struct buffer_head

The main IO unit in the 2.4 kernel is the `struct buffer_head`. It's a fairly unwieldy structure, used at various kernel layers for different things: caching entity, file system block, and IO unit. As a result, it's suboptimal for either of them.

From the block layer point of view, the two biggest problems is the size of the structure and the limitation in how big a data region it can describe. Being limited by the file system *one block* semantics, it can at most describe a `PAGE_CACHE_SIZE` amount of data. In Linux on x86 hardware that means 4KiB of data. Often it can be even worse: raw io typically uses the soft sector size of a queue (default 1KiB) for submitting io, which means that queuing eg 32KiB of IO will enter the io scheduler 32 times. To work around this limitation and get at least to a page at the time, a 2.4 hack was introduced. This is called `vary_io`. A driver advertising this capability acknowledges that it can manage `buffer_head's` of varying sizes at the same time. File system read-ahead, another frequent user of submitting larger sized io, has no option but to submit the read-ahead window in units of the page size.

1.3 Scalability

With the limit on `buffer_head` IO size and `elevator_linus` runtime, it doesn't take a lot of thinking to discover obvious scalability problems in the Linux 2.4 IO path. To add insult to injury, the entire IO path is guarded by a single, global lock: `io_request_lock`. This lock is held during the entire IO queuing operation, and typically also from the other end when a driver subtracts requests for IO submission. A single global lock is a big enough problem on its own (bigger SMP systems will suffer immensely because of cache line bouncing), but add to that long runtimes and you have a really huge IO scalability problem.

Linux vendors have long shipped lock scalability patches for quite some time to get around this problem. The adopted solution is typically to make the queue lock a pointer to a driver local lock, so the driver has full control of the granularity and scope of the lock. This solution was adopted from the 2.5 kernel, as we'll see later. But this is another case where driver writers often need to differentiate between vendor and vanilla kernels.

1.4 API problems

Looking at the block layer as a whole (including both ends of the spectrum, the producers and consumers of the IO units going through the block layer), it is a typical example of code that has been hacked into existence without much thought to design. When things broke or new features were needed, they had been grafted into the existing mess. No well defined interface exists between file system and block layer, except a few scattered functions. Controlling IO unit flow from IO scheduler to driver was impossible: 2.4 exposes the IO scheduler data structures (the `->queue_head` linked list used for queuing) directly to the driver. This fact alone makes it virtually impossible to implement more clever IO scheduling in 2.4. Even the recently (in the 2.4.20's) added lower latency work was horrible to work with because of this lack of boundaries. Verifying correctness of the code is extremely difficult; peer review of the code likewise, since a reviewer must be intimate with the block layer structures to follow the code.

Another example on lack of clear direction is the partition remapping. In 2.4, it's the driver's responsibility to resolve partition mappings. A given request contains a device and sector offset (i.e. `/dev/hda4`, sector 128) and the driver must map this to an absolute device offset before sending it to the hardware. Not only does this cause duplicate code in the drivers, it also means the IO scheduler has no knowledge of the real device mapping of a particular request. This adversely impacts IO scheduling whenever partitions aren't laid out in strict ascending disk order, since it causes the io scheduler to make the wrong decisions when ordering io.

2 2.6 Block layer

The above observations were the initial kick off for the 2.5 block layer patches. To solve some of these issues the block layer needed to be turned inside out, breaking basically anything-io along the way.

2.1 bio

Given that `struct buffer_head` was one of the problems, it made sense to start from scratch with an IO unit that would be agreeable to the upper layers as well as the drivers. The main criteria for such an IO unit would be something along the lines of:

1. Must be able to contain an arbitrary amount of data, as much as the hardware allows. Or as much that makes *sense* at least, with the option of easily pushing this boundary later.
2. Must work equally well for pages that have a virtual mapping as well as ones that do not.
3. When entering the IO scheduler and driver, IO unit must point to an absolute location on disk.
4. Must be able to stack easily for IO stacks such as raid and device mappers. This includes full redirect stacking like in 2.4, as well as partial redirections.

Once the primary goals for the IO structure were laid out, the `struct bio` was born. It was decided to base the layout on a scatter-gather type setup, with the `bio` containing a map of pages. If the map count was made flexible, items 1 and 2 on the above list were already solved. The actual implementation involved splitting the data container from the `bio` itself into a `struct bio_vec` structure. This was mainly done to ease allocation of the structures so that `sizeof(struct bio)` was always constant. The `bio_vec` structure is simply a tuple of `{page, length, offset}`, and the `bio` can be allocated with room for anything from 1 to `BIO_MAX_PAGES`. Currently Linux defines that as 256 pages, meaning we can support up to 1MiB of data in a single `bio` for a system with 4KiB page size. At the time of implementation, 1MiB was a good deal beyond the point where increasing the IO size further didn't yield better performance or lower CPU usage. It also has the added bonus of making the `bio_vec` fit inside a single page, so we avoid higher order memory allocations (`sizeof(struct bio_vec)` == 12 on 32-bit, 16 on 64-bit) in the IO path. This is an important point, as it eases the pressure on the memory allocator. For swapping or other low memory situations, we ideally want to stress the allocator as little as possible.

Different hardware can support different sizes of io. Traditional parallel ATA can do a maximum of 128KiB per request, qlogicfc SCSI doesn't like more than 32KiB, and lots of high end controllers don't impose a significant limit on max IO size but may restrict the maximum number of segments that one IO may be composed of. Additionally, software raid or device mapper stacks may like special alignment of IO or the guarantee that IO won't cross stripe boundaries. All of this knowledge is either impractical or impossible to statically advertise to submitters of io, so an easy interface for populating a bio with pages was essential if supporting large IO was to become practical. The current solution is int bio_add_page() which attempts to add a single page (full or partial) to a bio. It returns the amount of bytes successfully added. Typical users of this function continue adding pages to a bio until it fails—then it is submitted for IO through submit_bio(), a new bio is allocated and populated until all data has gone out. int bio_add_page() uses statically defined parameters inside the request queue to determine how many pages can be added, and attempts to query a registered merge_bvec_fn for dynamic limits that the block layer cannot know about.

Drivers hooking into the block layer before the IO scheduler[1] deal with struct bio directly, as opposed to the struct request that are output after the IO scheduler. Even though the page addition API guarantees that they never need to be able to deal with a bio that is too big, they still have to manage local splits at sub-page granularity. The API was defined that way to make it easier for IO submitters to manage, so they don't have to deal with sub-page splits. 2.6 block layer defines two ways to deal with this situation—the first is the general clone interface. bio_clone() returns a clone of a bio. A clone is defined as a private copy of the bio itself, but with a shared bio_vec page map list. Drivers can modify the cloned bio and submit it to a different device without duplicating the data. The second interface is tailored specifically to single page splits and was written by kernel raid maintainer Neil Brown. The main function is bio_split() which returns a struct bio_pair describing the two parts of the original bio. The two bio's can then be submitted separately by the driver.

2.2 Partition remapping

Partition remapping is handled inside the IO stack before going to the driver, so that both drivers and IO schedulers have immediate full knowledge of precisely where data should end up. The device unfolding is done automatically by the same piece of code that resolves full bio redirects. The worker function is blk_partition_remap().

2.3 Barriers

Another feature that found its way to some vendor kernels is IO barriers. A barrier is defined as a piece of IO that is guaranteed to:

- Be on platter (or safe storage at least) when completion is signaled.
- Not proceed any previously submitted io.
- Not be proceeded by later submitted io.

The feature is handy for journalled file systems, fsync, and any sort of cache bypassing IO[2] where you want to provide guarantees on data order and correctness. The 2.6 code isn't even complete yet or in the Linus kernels, but it has made its way to Andrew Morton's -mm tree which is generally considered a staging area for features. This section describes the code so far.

The first type of barrier supported is a soft barrier. It isn't of much use for data integrity applications, since it merely implies ordering inside the IO scheduler. It is signaled with the REQ_SOFTBARRIER flag inside struct request. A stronger barrier is the

[1] Also known as at make_request time.

[2] Such types of IO include O_DIRECT or raw.

hard barrier. From the block layer and IO scheduler point of view, it is identical to the soft variant. Drivers need to know about it though, so they can take appropriate measures to correctly honor the barrier. So far the ide driver is the only one supporting a full, hard barrier. The issue was deemed most important for journalled desktop systems, where the lack of barriers and risk of crashes / power loss coupled with ide drives generally always defaulting to write back caching caused significant problems. Since the ATA command set isn't very intelligent in this regard, the ide solution adopted was to issue pre- and post flushes when encountering a barrier.

The hard and soft barrier share the feature that they are both tied to a piece of data (a `bio`, really) and cannot exist outside of data context. Certain applications of barriers would really like to issue a disk flush, where finding out which piece of data to attach it to is hard or impossible. To solve this problem, the 2.6 barrier code added the `blkdev_issue_flush()` function. The block layer part of the code is basically tied to a queue hook, so the driver issues the flush on its own. A helper function is provided for SCSI type devices, using the generic SCSI command transport that the block layer provides in 2.6 (more on this later). Unlike the queued data barriers, a barrier issued with `blkdev_issue_flush()` works on all interesting drivers in 2.6 (IDE, SCSI, SATA). The only missing bits are drivers that don't belong to one of these classes—things like **CISS** and **DAC960**.

2.4 IO Schedulers

As mentioned in section 1.1, there are a number of known problems with the default 2.4 IO scheduler and IO scheduler interface (or lack thereof). The idea to base latency on a unit of data (sectors) rather than a time based unit is hard to tune, or requires auto-tuning at runtime and this never really worked out. Fixing the runtime problems with `elevator_linus` is next to impossible due to the data structure exposing problem. So before being able to tackle any problems in that area, a neat API to the IO scheduler had to be defined.

2.4.1 Defined API

In the spirit of avoiding over-design[3], the API was based on initial adaption of `elevator_linus`, but has since grown quite a bit as newer IO schedulers required more entry points to exploit their features.

The core function of an IO scheduler is, naturally, insertion of new io units and extraction of ditto from drivers. So the first 2 API functions are defined, `next_req_fn` and `add_req_fn`. If you recall from section 1.1, a new IO unit is first attempted merged into an existing request in the IO scheduler queue. And if this fails and the newly allocated request has raced with someone else adding an adjacent IO unit to the queue in the mean time, we also attempt to merge `struct requests`. So 2 more functions were added to cater to these needs, `merge_fn` and `merge_req_fn`. Cleaning up after a successful merge is done through `merge_cleanup_fn`. Finally, a defined IO scheduler can provide init and exit functions, should it need to perform any duties during queue init or shutdown.

The above described the IO scheduler API as of 2.5.1, later on more functions were added to further abstract the IO scheduler away from the block layer core. More details may be found in the `struct elevator_s` in `<linux/elevator.h>` kernel include file.

[3] Some might, rightfully, claim that this is worse than no design

2.4.2 deadline

In kernel 2.5.39, `elevator_linus` was finally replaced by something more appropriate, the *deadline* IO scheduler. The principles behind it are pretty straight forward — new requests are assigned an expiry time in milliseconds, based on data direction. Internally, requests are managed on two different data structures. The sort list, used for inserts and front merge lookups, is based on a red-black tree. This provides **O(log n)** runtime for both insertion and lookups, clearly superior to the doubly linked list. Two FIFO lists exist for tracking request expiry times, using a double linked list. Since strict FIFO behavior is maintained on these two lists, they run in **O(1)** time. For back merges it is important to maintain good performance as well, as they dominate the total merge count due to the layout of files on disk. So **deadline** added a merge hash for back merges, ideally providing **O(1)** runtime for merges. Additionally, **deadline** adds a one-hit merge cache that is checked even before going to the hash. This gets surprisingly good hit rates, serving as much as 90% of the merges even for heavily threaded io.

Implementation details aside, **deadline** continues to build on the fact that the fastest way to access a single drive, is by scanning in the direction of ascending sector. With its superior runtime performance, **deadline** is able to support very large queue depths without suffering a performance loss or spending large amounts of time in the kernel. It also doesn't suffer from latency problems due to increased queue sizes. When a request expires in the FIFO, **deadline** jumps to that disk location and starts serving IO from there. To prevent accidental seek storms (which would further cause us to miss deadlines), **deadline** attempts to serve a number of requests from that location before jumping to the next expired request. This means that the assigned request deadlines are soft, not a specific hard target that must be met.

2.4.3 Anticipatory IO scheduler

While **deadline** works very well for most workloads, it fails to observe the natural dependencies that often exist between synchronous reads. Say you want to list the contents of a directory—that operation isn't merely a single sync read, it consists of a number of reads where only the completion of the final request will give you the directory listing. With **deadline**, you could get decent performance from such a workload in presence of other IO activities by assigning very tight read deadlines. But that isn't very optimal, since the disk will be serving other requests in between the dependent reads causing a potentially disk wide seek every time. On top of that, the tight deadlines will decrease performance on other io streams in the system.

Nick Piggin implemented an anticipatory IO scheduler [Iyer] during 2.5 to explore some interesting research in this area. The main idea behind the anticipatory IO scheduler is a concept called *deceptive idleness*. When a process issues a request and it completes, it might be ready to issue a new request (possibly close by) immediately. Take the directory listing example from above—it might require 3–4 IO operations to complete. When each of them completes, the process[4] is ready to issue the next one almost instantly. But the traditional io scheduler doesn't pay any attention to this fact, the new request must go through the IO scheduler and wait its turn. With **deadline**, you would have to typically wait 500 milliseconds for each read, if the queue is held busy by other processes. The result is poor interactive performance for each process, even though overall throughput might be acceptable or even good.

[4] Or the kernel, on behalf of the process.

Instead of moving on to the next request from an unrelated process immediately, the anticipatory IO scheduler (hence forth known as **AS**) opens a small window of opportunity for that process to submit a new IO request. If that happens, **AS** gives it a new chance and so on. Internally it keeps a decaying histogram of IO *think times* to help the anticipation be as accurate as possible.

Internally, **AS** is quite like **deadline**. It uses the same data structures and algorithms for sorting, lookups, and FIFO. If the think time is set to 0, it is very close to **deadline** in behavior. The only differences are various optimizations that have been applied to either scheduler allowing them to diverge a little. If **AS** is able to reliably predict when waiting for a new request is worthwhile, it gets phenomenal performance with excellent interactiveness. Often the system throughput is sacrificed a little bit, so depending on the workload **AS** might not be the best choice always. The IO storage hardware used, also plays a role in this—a non-queuing ATA hard drive is a much better fit than a SCSI drive with a large queuing depth. The SCSI firmware reorders requests internally, thus often destroying any accounting that **AS** is trying to do.

2.4.4 CFQ

The third new IO scheduler in 2.6 is called **CFQ**. It's loosely based on the ideas on stochastic fair queuing (SFQ [McKenney]). **SFQ** is fair as long as its hashing doesn't collide, and to avoid that, it uses a continually changing hashing function. Collisions can't be completely avoided though, frequency will depend entirely on workload and timing. **CFQ** is an acronym for completely fair queuing, attempting to get around the collision problem that **SFQ** suffers from. To do so, **CFQ** does away with the fixed number of buckets that processes can be placed in. And using regular hashing technique to find the appropriate bucket in case of collisions, fatal collisions are avoided.

CFQ deviates radically from the concepts that **deadline** and **AS** is based on. It doesn't assign deadlines to incoming requests to maintain fairness, instead it attempts to divide bandwidth equally among classes of processes based on some correlation between them. The default is to hash on thread group id, tgid. This means that bandwidth is attempted distributed equally among the processes in the system. Each class has its own request sort and hash list, using red-black trees again for sorting and regular hashing for back merges. When dealing with writes, there is a little catch. A process will almost never be performing its own writes—data is marked dirty in context of the process, but write back usually takes place from the *pdflush* kernel threads. So **CFQ** is actually dividing read bandwidth among processes, while treating each pdflush thread as a separate process. Usually this has very minor impact on write back performance. Latency is much less of an issue with writes, and good throughput is very easy to achieve due to their inherent asynchronous nature.

2.5 Request allocation

Each block driver in the system has at least one `request_queue_t` request queue structure associated with it. The recommended setup is to assign a queue to each logical spindle. In turn, each request queue has a `struct request_list` embedded which holds free `struct request` structures used for queuing io. 2.4 improved on this situation from 2.2, where a single global free list was available to add one per queue instead. This free list was split into two sections of equal size, for reads and writes, to prevent either direction from starving the other[5]. 2.4 statically allocated a big chunk of requests for each queue, all residing in the precious low memory of a machine. The combination of **O(N)** runtime and statically allocated request structures firmly prevented any real world experimentation with large queue depths on 2.4 kernels.

2.6 improves on this situation by dynamically allocating request structures on the fly instead. Each queue still maintains its request free list like in 2.4. However it's also backed by a memory pool[6] to provide deadlock free allocations even during swapping. The more advanced io schedulers in 2.6 usually back each request by its own private request structure, further increasing the memory pressure of each request. Dynamic request allocation lifts some of this pressure as well by pushing that allocation inside two hooks in the IO scheduler API—`set_req_fn` and `put_req_fn`. The latter handles the later freeing of that data structure.

2.6 Plugging

For the longest time, the Linux block layer has used a technique dubbed *plugging* to increase IO throughput. In its simplicity, plugging works sort of like the plug in your tub drain—when IO is queued on an initially empty queue, the queue is plugged. Only when someone asks for the completion of some of the queued IO is the plug yanked out, and io is allowed to drain from the queue. So instead of submitting the first immediately to the driver, the block layer allows a small buildup of requests. There's nothing wrong with the principle of plugging, and it has been shown to work well for a number of workloads. However, the block layer maintains a global list of plugged queues inside the `tq_disk` task queue. There are three main problems with this approach:

1. It's impossible to go backwards from the file system and find the specific queue to unplug.

2. Unplugging one queue through `tq_disk` unplugs all plugged queues.

3. The act of plugging and unplugging touches a global lock.

All of these adversely impact performance. These problems weren't really solved until late in 2.6, when Intel reported a huge scalability problem related to unplugging [Chen] on a 32 processor system. 93% of system time was spent due to contention on `blk_plug_lock`, which is the 2.6 direct equivalent of the 2.4 `tq_disk` embedded lock. The proposed solution was to move the plug lists to a per-CMU structure. While this would solve the contention problems, it still leaves the other 2 items on the above list unsolved.

So work was started to find a solution that would fix all problems at once, and just generally Feel Right. 2.6 contains a link between the block layer and write out paths which is embedded inside the queue, a `struct backing_dev_info`. This structure holds information on read-ahead and queue congestion state. It's also possible to go from a `struct page` to the backing device, which may or may not be a block device. So it would seem an obvious idea to move to a backing device unplugging scheme instead, getting rid of the global `blk_run_queues()` unplugging. That solution would fix all three issues at once—there would be no global way to unplug all devices, only target specific unplugs, and the backing device gives us a mapping from page to queue. The code was rewritten to do just that, and provide unplug functionality going from a specific `struct block_device`, page, or backing device. Code and interface was much superior to the existing code base,

[5] In reality, to prevent writes for consuming all requests.

[6] `mempool_t` interface from Ingo Molnar.

and results were truly amazing. Jeremy Higdon tested on an 8-way IA64 box [Higdon] and got 75–80 thousand IOPS on the stock kernel at 100% CPU utilization, 110 thousand IOPS with the per-CPU Intel patch also at full CPU utilization, and finally 200 thousand IOPS at merely 65% CPU utilization with the backing device unplugging. So not only did the new code provide a huge speed increase on this machine, it also went from being CPU to IO bound.

2.6 also contains some additional logic to unplug a given queue once it reaches the point where waiting longer doesn't make much sense. So where 2.4 will always wait for an explicit unplug, 2.6 can trigger an unplug when one of two conditions are met:

1. The number of queued requests reach a certain limit, `q->unplug_thresh`. This is device tweak able and defaults to 4.
2. When the queue has been idle for `q->unplug_delay`. Also device tweak able, and defaults to 3 milliseconds.

The idea is that once a certain number of requests have accumulated in the queue, it doesn't make much sense to continue waiting for more—there is already an adequate number available to keep the disk happy. The time limit is really a last resort, and should rarely trigger in real life. Observations on various work loads have verified this. More than a handful or two timer unplugs per minute usually indicates a kernel bug.

2.7 SCSI command transport

An annoying aspect of CD writing applications in 2.4 has been the need to use ide-scsi, necessitating the inclusion of the entire SCSI stack for only that application. With the clear majority of the market being ATAPI hardware, this becomes even more silly. ide-scsi isn't without its own class of problems either—it lacks the ability to use DMA on certain writing types. CDDA audio ripping is another application that thrives with ide-scsi, since the native uniform cdrom layer interface is less than optimal (put mildly). It doesn't have DMA capabilities at all.

2.7.1 Enhancing struct request

The problem with 2.4 was the lack of ability to generically send SCSI "like" commands to devices that understand them. Historically, only file system read/write requests could be submitted to a driver. Some drivers made up faked requests for other purposes themselves and put then on the queue for their own consumption, but no defined way of doing this existed. 2.6 adds a new request type, marked by the `REQ_BLOCK_PC` bit. Such a request can be either backed by a `bio` like a file system request, or simply has a data and length field set. For both types, a SCSI command data block is filled inside the request. With this infrastructure in place and appropriate update to drivers to understand these requests, it's a cinch to support a much better direct-to-device interface for burning.

Most applications use the SCSI sg API for talking to devices. Some of them talk directly to the `/dev/sg*` special files, while (most) others use the `SG_IO` ioctl interface. The former requires a yet unfinished driver to transform them into block layer requests, but the latter can be readily intercepted in the kernel and routed directly to the device instead of through the SCSI layer. Helper functions were added to make burning and ripping even faster, providing DMA for all applications and without copying data between kernel and user space at all. So the zero-copy DMA burning was possible, and this even without changing most applications.

3 Linux-2.7

The 2.5 development cycle saw the most massively changed block layer in the history of Linux. Before 2.5 was opened, Linus had clearly expressed that one of the most important things that needed doing, was the block layer update. And indeed, the very first thing merged was the complete bio patch into 2.5.1-pre2. At that time, no more than a handful drivers compiled (let alone worked). The 2.7 changes will be nowhere as severe or drastic. A few of the possible directions will follow in the next few sections.

3.1 IO Priorities

Prioritized IO is a very interesting area that is sure to generate lots of discussion and development. It's one of the missing pieces of the complete resource management puzzle that several groups of people would very much like to solve. People running systems with many users, or machines hosting virtual hosts (or completed virtualized environments) are dying to be able to provide some QOS guarantees. Some work was already done in this area, so far nothing complete has materialized. The CKRM [CKRM] project spear headed by IBM is an attempt to define global resource management, including io. They applied a little work to the **CFQ** IO scheduler to provide equal bandwidth between resource management classes, but at no specific priorities. Currently I have a **CFQ** patch that is 99% complete that provides full priority support, using the IO contexts introduced by **AS** to manage fair sharing over the full time span that a process exists[7]. This works well enough, but only works for that specific IO scheduler. A nicer solution would be to create a scheme that works independently of the io scheduler used. That would require a rethinking of the IO scheduler API.

3.2 IO Scheduler switching

Currently Linux provides no less than 4 IO schedulers—the 3 mentioned, plus a forth dubbed **noop**. The latter is a simple IO scheduler that does no request reordering, no latency management, and always merges whenever it can. Its area of application is mainly highly intelligent hardware with huge queue depths, where regular request reordering doesn't make sense. Selecting a specific IO scheduler can either be done by modifying the source of a driver and putting the appropriate calls in there at queue init time, or globally for any queue by passing the `elevator=xxx` boot parameter. This makes it impossible, or at least very impractical, to benchmark different IO schedulers without many reboots or recompiles. Some way to switch IO schedulers per queue and on the fly is desperately needed. Freezing a queue and letting IO drain from it until it's empty (pinning new IO along the way), and then shutting down the old io scheduler and moving to the new scheduler would not be so hard to do. The queues expose various sysfs variables already, so the logical approach would simply be to:

```
# echo deadline > \
    /sys/block/hda/queue/io_scheduler
```

A simple but effective interface. At least two patches doing something like this were already proposed, but nothing was merged at that time.

4 Final comments

The block layer code in 2.6 has come a long way from the rotted 2.4 code. New features

[7] CFQ currently tears down class structures as soon as it is empty, it doesn't persist over process life time.

bring it more up-to-date with modern hardware, and completely rewritten from scratch core provides much better scalability, performance, and memory usage benefiting any machine from small to really huge. Going back a few years, I heard constant complaints about the block layer and how much it sucked and how outdated it was. These days I rarely hear anything about the current state of affairs, which usually means that it's doing pretty well indeed. 2.7 work will mainly focus on feature additions and driver layer abstractions (our concept of IDE layer, SCSI layer etc will be severely shook up). Nothing that will wreak havoc and turn everything inside out like 2.5 did. Most of the 2.7 work mentioned above is pretty light, and could easily be back ported to 2.6 once it has been completed and tested. Which is also a good sign that nothing really radical or risky is missing. So things are settling down, a sign of stability.

References

[Iyer] Sitaram Iyer and Peter Druschel, *Anticipatory scheduling: A disk scheduling framework to overcome deceptive idleness in synchronous I/O*, 18th ACM Symposium on Operating Systems Principles, `http://www.cs.rice.edu/~ssiyer/r/antsched/antsched.ps.gz`, 2001

[McKenney] Paul E. McKenney, *Stochastic Fairness Queuing*, INFOCOM `http://rdrop.com/users/paulmck/paper/sfq.2002.06.04.pdf`, 1990

[Chen] Kenneth W. Chen, *per-cpu blk_plug_list*, Linux kernel mailing list `http://www.ussg.iu.edu/hypermail/linux/kernel/0403.0/0179.html`, 2004

[Higdon] Jeremy Higdon, *Re: [PATCH] per-backing dev unplugging #2*, Linux kernel mailing list `http://marc.theaimsgroup.com/?l=linux-kernel&m=107941470424309&w=2`, 2004

[CKRM] IBM, *Class-based Kernel Resource Management (CKRM)*, `http://ckrm.sf.net`, 2004

[Bhattacharya] Suparna Bhattacharya, *Notes on the Generic Block Layer Rewrite in Linux 2.5*, General discussion, `Documentation/block/biodoc.txt`, 2002

Proceedings of the
Linux Symposium

Volume One

July 21st–24th, 2004
Ottawa, Ontario
Canada

Conference Organizers

Andrew J. Hutton, *Steamballoon, Inc.*
Stephanie Donovan, *Linux Symposium*
C. Craig Ross, *Linux Symposium*

Review Committee

Jes Sorensen, *Wild Open Source, Inc.*
Matt Domsch, *Dell*
Gerrit Huizenga, *IBM*
Matthew Wilcox, *Hewlett-Packard*
Dirk Hohndel, *Intel*
Val Henson, *Sun Microsystems*
Jamal Hadi Salimi, *Znyx*
Andrew Hutton, *Steamballoon, Inc.*

Proceedings Formatting Team

John W. Lockhart, *Red Hat, Inc.*