When 2MB turns into 512KB

One frequently asked question I get is why a storage device is doing differently sized IO than what the application issued. If the application design spec depends on bigger sized IO hitting the media, getting several smaller requests can pose a performance problem. To prevent this from happening, it's important to understand precisely what goes on from system call to device driver. Some of the below gets into kernel details that the application developer doesn't necessarily *need* to know about, but the information can come in handy when debugging discrepancies in IO patterns on the application and hardware side. Sometimes the issue is with the kernel itself, not the application.

IO Units

Regardless of the method used to submit the IO, at some point the kernel will start assembling it in page-sized chunks. The container used to hold these pages is called a struct bio, and this is where we will potentially run into the first limiter of IO size. A bio can only hold 256 pages. Most systems will run with a page size of 4KB, which results in a bio holding at most 1MB of data. Luckily, for storage devices, the unit of transfer is not a single bio. The block layer will assemble multiple bio units into a single struct request. A request can hold an arbitrary number bio structures, so ideally this should not impose a limitation on the resulting IO size.

Plugging

When the kernel is assembling and submitting strings of bio units, it's critical that the block layer does not start sending these to the device driver until they are fully assembled. If that happens, then we can get a series of IO requests that aren't fully assembled yet. This can turn what should have been a nice series of 512KB-sized sequential IO requests into an arbitrary sequence of smaller IOs. To prevent this from happening, the block layer utilizes a concept called plugging. Plugging is a task-based state. Before the kernel starts submitting a series of IO on behalf of an application, it initiates a plug sequence. When it's done submitting IO, it signals the end of a plug sequence. The latter flushes the IO that the task has queued and submits it to the block layer. This effectively prevents premature submission of IO. Plug sequences can

be observed through blktrace as 'P' (plug) events, and as 'U' (unplug) events. Normally unplugging will happen explicitly when the task is done submitting IO. Occasionally it can happen implicitly, if the task ends up being scheduled out. In the event that this happens, the kernel will auto-flush the plug to ensure that IO doesn't sit idle while a task is sleeping.

IO Scheduler merging

Assembling smaller bio units into a bigger request is done jointly by the block layer and the IO scheduler. There are various types of merging that is supported. The most basic func-

tionality will be performed by the block layer, and the IO scheduler can augment that with more sophisticated merging. An example of that would be last-hit merging. This is done by the block layer, and simply checks if a new bio unit can be merged to the previous request merge. This will catch most of the plug merges from the plug sequence mentioned above. However, since this last merge state is per-device and the plug state is per-task, there are cases where interleaved IO from multiple tasks could leave it less effective. Because of that, IO schedulers will do explicit lookup merges to check if we can merge with an existing request, either at the front or at the back of it. This means that the effectiveness of merging can be at the mercy of the selected IO scheduler. Some setups turn off IO scheduling by setting it to **none** or **noop**, which can have an impact on the resulting IO sizes since it can result in decreased merging. **deadline** or **mq-deadline** should catch all merge opportunities.

Device limitations

If the kernel supports unbounded IO sizes, then why are you still not seeing the sizes you expect on the backend? The answer could be that your hardware has IO size restrictions that are limiting the maximum size it can support in a single command. Luckily the kernel exports this kind of information for you. As an example, let's look at a standard NVMe device. All of the relevant parameters can be found in the block sysfs directory for the device, which in this case is `/sys/block/nvme0n1/queue/` :

```
max_hw_sectors_kb:128
max_sectors_kb:128
max_segments:33
max_segment_size:65536
```

This tells us that the maximum size the device can support is 128KB (`max_hw_sectors_kb`) and the maximum size that the kernel allows is 128KB (`max_sectors_kb`). Additionally, the DMA engine is limited to 33 segments of IO, each with a max size of 64KB. The latter are the scatter/gather (SG) properties of the device. If the IO being submitted isn't physically contiguous, multiple SG entries are needed to describe the IO to the hardware. If the SG limitations are sufficiently low, this

can be a source of frustration in terms seemingly random device IO sizes from uniformly large application IO sizes. This is especially the case for buffered IO, where the kernel is the allocator of the backing pages. If the kernel memory space is fragmented, we might not be getting nice sequences of sequential pages. From the NVMe example above, this is not

going to be a concern, since it supports 33 segments. Assuming 4KB page size and IO sizes, that's a minimum of 132KB in a request, exceeding the device limit of 128KB. But let's look at a different example from the SCSI world:

```
max_hw_sectors_kb:16383
max_sectors_kb:1280
max_segments:128
```

`max_segment_size:65536`

Here the hardware advertises almost 16MB of IO size. While the kernel defaults to 1280KB as the max IO size, that's a tunable that can be modified to anything between 4KB and the `max_hw_sectors_kb` setting. Looking at the SG restrictions on this device, it supports a maximum of 128 segments, with each segment being up to 64KB in size. Assuming no contiguous pages, that's a max of 4*128 == 512KB per request. Contrary to the NVMe example, for this device we're effectively limited by the SG restrictions. In this kind of setup, we cannot reliably guarantee more than a 512KB request size.

Performance impact

Now you're disappointed that you can't do more than 512KB reliably, since you were planning and hoping for 2MB requests. How big of a performance impact is 4x512KB compared to 1x2MB? Unfortunately there's no perfect answer to this question. In most cases, the performance impact will be negligible. Obviously we'll be doing four times as many commands. If we're dealing with rotating storage, command overhead is generally not a concern, since IOPS rates are low. For flash devices, unless we're anywhere near IOPS limited, it's also not going to be a problem. The biggest concern is rotating storage, and ensuring that the IOs go out on the wire as close to each other as possible. We don't want a sequential piece of 2MB IO to grow a seek, since this would shatter our performance. The requests will generally go out in sequence, and the storage device can effectively combine these internally. As all modern devices support command queuing, a 2MB queue depth of 1 will just turn into a 512KB queue depth of 4. As long as we stay below the effective queue depth of the device, hopefully it won't be too much of a performance impact.

Some IO schedulers will actively try to limit the device queue depth, CFQ being an example of that. CFQ has tunables to deal with this (`quantum`, `slice_async_rq`), but I would recommend first trying deadline if you see IO limitations being hit and resulting performance issues because of it. Similarly, kyber utilizes dispatch tokens to control queue depth.

As long as the requests go out in a single batch and aren't subject to reordering, hopefully performance will be in the expected range even for split requests. Observed device IOPS will differ, of course, and you might need to verify with blktrace that driver request issues and device completions look as expected.