What's new with io_uring

It's now been about 6 months since the first upstream kernel (v5.1) was released with io_uring support. As with any new API and feature, the initial version is just the jumping off point. Once folks start converting existing applications to the API, or start writing new ones against it, inevitably feature requests pop up. This short note will attempt to introduce some of the more important additions to io_uring since its inception.
If you're not familiar with io_uring, I'd invite you to read the introductory document I wrote back in April. It's been updated a few times since I posted it in a note here, the canonical version can always be found here:
http://kernel.dk/io_uring.pdf

NEW COMMANDS

Most of the new features are inevitably new op-codes for io_uring. These add new core functionality, and most of them are just mirrors of regular synchronous system calls. For actual command definitions, I'd encourage the reader to clone liburing which has prep helpers for setting each of these up. Clone it here:

git://git.kernel.dk/liburing

In no particular order of importance, the new commands are:

`IORING_OP_SYNC_FILE_RANGE.` This command adds support for executing `sync_file_range(2)` in an async manner. It supports all the features that the sync system call does.

`IORING_OP_SENDMSG` and `IORING_OP_RECVMSG`. It was always possible to do regular `IORING_OP_READV` and `IORING_OP_WRITEV` on sockets, but that was the only way to do networked IO with io_uring. Now we support async versions of `sendmsg(2)` and `recvmsg(2)` as well. These will execute inline, if possible, otherwise done in the background if they would block the submitting application.

`IORING_OP_ACCEPT`. Like the send/recvmsg calls, this provides support for async accept4(2) system calls. This is the first system call io_uring supports that creates new file descriptors, and a lot of work had to go into supporting code for this to happen. This also enables us to add open(2) support in the future.

`IORING_OP_TIMEOUT`. This command is special in that it doesn't mirror an existing system call, rather it adds support for triggering a timeout condition in the CQ ring to wake an application sleeping on events. The timeout is one of two events - a number of completions, or a specific timeout (absolute or relative). Whatever event triggers first will queue a completion event in the CQ ring and wake up waiters. liburing uses timeouts internally to provide support for `io_uring_wait_cqe_timeout()`, but applications can also use them specifically for whatever timeout need they have. Applications may delete existing timeouts before they occur with `IORING_OP_TIMEOUT_REMOVE`. This op-code will remove an existing timeout.

`IORING_OP_ASYNC_CANCEL`. We now finally have support for cancelling existing async work! Folks familiar with aio/libaio may say that this has been the case for a long time with io_cancel(2), but that has never been implemented. It works with the poll command for aio, but that's it. In io_uring, this works with any read/write request, accept, send/recvmsg, etc. There's an important distinction to make here with the different kinds of commands. A read/write on a regular file will generally be waiting for IO completion in an uninterruptible state. This means it'll ignore any signals or attempts to cancel it, as these operations are uncancellable. io_uring can cancel these operations if they haven't yet been started. If they have been started, cancellations on these will fail. Network IO will generally be waiting interruptibly, and can hence be cancelled at any time. The completion event for this request will have a result of `0` if done successfully, `-EALREADY` if the operation is already in progress, and `-ENOENT` if the original request specified cannot be found. For cancellation requests that return `-EALREADY`, io_uring may or may not cause this request to be

stopped sooner. For blocking IO, the original request will complete as it originally would have. For IO that is cancellable, it will terminate sooner if at all possible.

COMMAND EXECUTION CHANGES

Two new features have been implemented which modify the submission queue pipeline for io_uring. By default, since submission and completion both utilize shared rings, any command that is queued up in the SQ ring will be seen in order by the kernel for execution. Commands may complete out of order, and often do, but submissions are always done in the order in which they were placed in the SQ ring. Two new features modify that.

`IOSQE_IO_DRAIN`. This is a flag set in the `sqe->flags` member. If set on a command, submission of this command will be deferred until previously issued commands have completed. As such, it provides a drain like functionality for the SQ ring.

`IOSQE_IO_LINK`. This is a flag set in the `sqe->flags` member. If set, the next SQE in the ring will depend on this SQE. A dependent SQE will not be started until the parent SQE has completed. If the parent SQE fails, then a dependent SQE will be failed without being started. Link chains can be arbitrarily long, the chain spans any new SQE that continues to have the `IOSQE_IO_LINK` flag set. Once an SQE is encountered that does not have this flag set, that defines the end of the chain. This features allows to form dependencies between individual SQEs. liburing has an example of a (simplified) cp(1) implementation that uses dependent SQEs to make read/write chains. If a read from fileX is successful, a write to fileY is automatically done. You can view this chain as: {{ READ, fileX, offsetX, bytesX},{WRITE, fileY, offsetY, bytesX}}.

MISCELLANEOUS

eventfd. io_uring now also supports eventfd notifications on the ring itself, for applications that want to use eventfd for notification of completion events.

The registered file set support has been expanded a lot. It's now no longer limited to 1024 files, but supports 64K registered files. It also supports sparse file sets, where a large set will have fd == -1 set. This is significant because we now also support file set updates, where an application can update a number of files at a specific offset in the table explicitly. Before this change, the only way to update/change a file set was to unregister the existing one, then register a new one.

By default, io_uring will size the CQ ring as twice the size of the SQ ring. This is done because of how SQE lifetimes are very short, they are consumed as soon as the kernel has seen them. This means that an application can drive a much higher count of in-flight requests than the SQ size would seem to indicate. To avoid easily overflowing the CQ ring, we double the size of the ring to allow some slack. There are certain use cases that need a MUCH larger CQ ring than SQ ring. Previously they had to use a big SQ ring to accommodate that, but that is inefficient in terms of memory utilization. io_uring now supports independently sizing the CQ ring, making it possible to have an (eg) SQ ring that's 128 entries big, but a CQ ring that's 32K. If an application wants to influence the CQ ring size independently, it must set `IORING_SETUP_CQSIZE` in the `io_uring_params` structure passed in to ring creation (`io_uring_setup(2)` for the system call, or `io_uring_queue_init_params(3)` for liburing) and set `params->cq_entries` to the desired size. The CQ ring size must be at least the same as the SQ ring, and it must also be a power-of-two just like the SQ ring size.

io_uring has also now divorced itself from the kernel workqueue infrastructure. This is a purely internal change that isn't visible in the API. There are numerous reasons for why that was necessary, anyone who's interested in the detailed (current and future) justification can read the two commits here and here. The important take-away is that it enables several of the features mentioned in this note.