The errseq_t datatype

An errseq_t is a way of recording errors in one place, and allowing any number of "subscribers" to tell whether it has changed since a previous point where it was sampled.

The initial use case for this is tracking errors for file synchronization syscalls (fsync, fdatasync, msync and sync_file_range), but it may be usable in other situations.

It's implemented as an unsigned 32-bit value. The low order bits are designated to hold an error code (between 1 and MAX_ERRNO). The upper bits are used as a counter. This is done with atomics instead of locking so that these functions can be called from any context.

Note that there is a risk of collisions if new errors are being recorded frequently, since we have so few bits to use as a counter.

To mitigate this, the bit between the error value and counter is used as a flag to tell whether the value has been sampled since a new value was recorded. That allows us to avoid bumping the counter if no one has sampled it since the last time an error was recorded.

Thus we end up with a value that looks something like this:

31..13	12	11..0
counter	SF	errno

The general idea is for "watchers" to sample an errseq_t value and keep it as a running cursor. That value can later be used to tell whether any new errors have occurred since that sampling was done, and atomically record the state at the time that it was checked. This allows us to record errors in one place, and then have a number of "watchers" that can tell whether the value has changed since they last checked it.

A new errseq_t should always be zeroed out. An errseq_t value of all zeroes is the special (but common) case where there has never been an error. An all zero value thus serves as the "epoch" if one wishes to know whether there has ever been an error set since it was first initialized.

API usage

Let me tell you a story about a worker drone. Now, he's a good worker overall, but the company is a little...management heavy. He has to report to 77 supervisors today, and tomorrow the "big boss" is coming in from out of town and he's sure to test the poor fellow too.

They're all handing him work to do -- so much he can't keep track of who handed him what, but that's not really a big problem. The supervisors just want to know when he's finished all of the work they've handed him so far and whether he made any mistakes since they last asked.

He might have made the mistake on work they didn't actually hand him, but he can't keep track of things at that level of detail, all he can remember is the most recent mistake that he made.

Here's our worker_drone representation:

```
struct worker_drone {
        errseq_t        wd_err; /* for recording errors */
};
```

Every day, the worker_drone starts out with a blank slate:

```
struct worker_drone wd;

wd.wd_err = (errseq_t)0;
```

The supervisors come in and get an initial read for the day. They don't care about anything that happened before their watch begins:

```
struct supervisor {
        errseq_t        s_wd_err; /* private "cursor" for wd_err */
        spinlock_t      s_wd_err_lock; /* protects s_wd_err */
}

struct supervisor       su;

su.s_wd_err = errseq_sample(&wd.wd_err);
spin_lock_init(&su.s_wd_err_lock);
```

Now they start handing him tasks to do. Every few minutes they ask him to finish up all of the work they've handed him so far. Then they ask him whether he made any mistakes on any of it:

```
spin_lock(&su.su_wd_err_lock);
err = errseq_check_and_advance(&wd.wd_err, &su.s_wd_err);
spin_unlock(&su.su_wd_err_lock);
```

Up to this point, that just keeps returning 0.

Now, the owners of this company are quite miserly and have given him substandard equipment with which to do his job. Occasionally it glitches and he makes a mistake. He sighs a heavy sigh, and marks it down:

```
errseq_set(&wd.wd_err, -EIO);
```

...and then gets back to work. The supervisors eventually poll again and they each get the error when they next check. Subsequent calls will return 0, until another error is recorded, at which point it's reported to each of them once.

Note that the supervisors can't tell how many mistakes he made, only whether one was made since they last checked, and the latest value recorded.

Occasionally the big boss comes in for a spot check and asks the worker to do a one-off job for him. He's not really watching the worker full-time like the supervisors, but he does need to know whether a mistake occurred while his job was processing.

He can just sample the current errseq_t in the worker, and then use that to tell whether an error has occurred later:

```
errseq_t since = errseq_sample(&wd.wd_err);
/* submit some work and wait for it to complete */
err = errseq_check(&wd.wd_err, since);
```

Since he's just going to discard "since" after that point, he doesn't need to advance it here. He also doesn't need any locking since it's not usable by anyone else.

Serializing errseq_t cursor updates

Note that the errseq_t API does not protect the errseq_t cursor during a check_and_advance_operation. Only the canonical error code is handled atomically. In a situation where more than one task might be using the same errseq_t cursor at the same time, it's important to serialize updates to that cursor.

If that's not done, then it's possible for the cursor to go backward in which case the same error could be reported more than once.

Because of this, it's often advantageous to first do an errseq_check to see if anything has changed, and only later do an errseq_check_and_advance after taking the lock. e.g.:

```
if (errseq_check(&wd.wd_err, READ_ONCE(su.s_wd_err)) {
        /* su.s_wd_err is protected by s_wd_err_lock */
        spin_lock(&su.s_wd_err_lock);
        err = errseq_check_and_advance(&wd.wd_err, &su.s_wd_err);
        spin_unlock(&su.s_wd_err_lock);
}
```

That avoids the spinlock in the common case where nothing has changed since the last time it was checked.

Functions

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\core-api\(linux-master)(Documentation)(core-api)errseq.rst, line 159)

Unknown directive type "kernel-doc".

```
.. kernel-doc:: lib/errseq.c
```