SLOW WORK ITEM EXECUTION THREAD POOL

By: David Howells <dhowells@redhat.com>

The slow work item execution thread pool is a pool of threads for performing things that take a relatively long time, such as making mkdir calls. Typically, when processing something, these items will spend a lot of time blocking a thread on I/O, thus making that thread unavailable for doing other work.

The standard workqueue model is unsuitable for this class of work item as that limits the owner to a single thread or a single thread per CPU. For some tasks, however, more threads - or fewer - are required.

There is just one pool per system. It contains no threads unless something wants to use it - and that something must register its interest first. When the pool is active, the number of threads it contains is dynamic, varying between a maximum and minimum setting, depending on the load.

CLASSES OF WORK ITEM

This pool support two classes of work items:

 (*) Slow work items.

 (*) Very slow work items.

The former are expected to finish much quicker than the latter.

An operation of the very slow class may do a batch combination of several lookups, mkdirs, and a create for instance.

An operation of the ordinarily slow class may, for example, write stuff or expand files, provided the time taken to do so isn't too long.

Operations of both types may sleep during execution, thus tying up the thread loaned to it.

A further class of work item is available, based on the slow work item class:

 (*) Delayed slow work items.

These are slow work items that have a timer to defer queueing of the item for a while.

THREAD-TO-CLASS ALLOCATION

Not all the threads in the pool are available to work on very slow work items. The number will be between one and one fewer than the number of active threads.

This is configurable (see the "Pool Configuration" section).

All the threads are available to work on ordinarily slow work items, but a percentage of the threads will prefer to work on very slow work items.

The configuration ensures that at least one thread will be available to work on very slow work items, and at least one thread will be available that won't work on very slow work items at all.

USING SLOW WORK ITEMS

Firstly, a module or subsystem wanting to make use of slow work items must register its interest:

```
int ret = slow_work_register_user(struct module *module);
```

This will return 0 if successful, or a -ve error upon failure. The module pointer should be the module interested in using this facility (almost certainly THIS_MODULE).

Slow work items may then be set up by:

(1) Declaring a slow_work struct type variable:

```
#include <linux/slow-work.h>

struct slow_work myitem;
```

(2) Declaring the operations to be used for this item:

```
struct slow_work_ops myitem_ops = {
        .get_ref = myitem_get_ref,
        .put_ref = myitem_put_ref,
        .execute = myitem_execute,
};
```

[*] For a description of the ops, see section "Item Operations".

(3) Initialising the item:

```
slow_work_init(&myitem, &myitem_ops);
```

or:

```
delayed_slow_work_init(&myitem, &myitem_ops);
```

or:

```
vslow_work_init(&myitem, &myitem_ops);
```

depending on its class.

A suitably set up work item can then be enqueued for processing:

```
int ret = slow_work_enqueue(&myitem);
```

This will return a -ve error if the thread pool is unable to gain a reference on the item, 0 otherwise, or (for delayed work):

```
int ret = delayed_slow_work_enqueue(&myitem, my_jiffy_delay);
```

The items are reference counted, so there ought to be no need for a flush operation. But as the reference counting is optional, means to cancel existing work items are also included:

```
cancel_slow_work(&myitem);
cancel_delayed_slow_work(&myitem);
```

can be used to cancel pending work. The above cancel function waits for existing work to have been executed (or prevent execution of them, depending on timing).

When all a module's slow work items have been processed, and the module has no further interest in the facility, it should unregister its interest:

```
slow_work_unregister_user(struct module *module);
```

The module pointer is used to wait for all outstanding work items for that module before completing the unregistration. This prevents the put_ref() code from being taken away before it completes. module should almost certainly be THIS_MODULE.

==========
HELPER FUNCTIONS
==========

The slow-work facility provides a function by which it can be determined whether or not an item is queued for later execution:

```
bool queued = slow_work_is_queued(struct slow_work *work);
```

If it returns false, then the item is not on the queue (it may be executing with a requeue pending). This can be used to work out whether an item on which another depends is on the queue, thus allowing a dependent item to be queued after it.

If the above shows an item on which another depends not to be queued, then the owner of the dependent item might need to wait. However, to avoid locking up the threads unnecessarily be sleeping in them, it can make sense under some circumstances to return the work item to the queue, thus deferring it until some other items have had a chance to make use of the yielded thread.

To yield a thread and defer an item, the work function should simply enqueue the work item again and return. However, this doesn't work if there's nothing

actually on the queue, as the thread just vacated will jump straight back into the item's work function, thus busy waiting on a CPU.

Instead, the item should use the thread to wait for the dependency to go away, but rather than using schedule() or schedule_timeout() to sleep, it should use the following function:

```
bool requeue = slow_work_sleep_till_thread_needed(
                struct slow_work *work,
                signed long *_timeout);
```

This will add a second wait and then sleep, such that it will be woken up if either something appears on the queue that could usefully make use of the thread - and behind which this item can be queued, or if the event the caller set up to wait for happens. True will be returned if something else appeared on the queue and this work function should perhaps return, of false if something else woke it up. The timeout is as for schedule_timeout().

For example:

```
wq = bit_waitqueue(&my_flags, MY_BIT);
init_wait(&wait);
requeue = false;
do {
        prepare_to_wait(wq, &wait, TASK_UNINTERRUPTIBLE);
        if (!test_bit(MY_BIT, &my_flags))
                break;
        requeue = slow_work_sleep_till_thread_needed(&my_work,
                                                     &timeout);
} while (timeout > 0 && !requeue);
finish_wait(wq, &wait);
if (!test_bit(MY_BIT, &my_flags)
        goto do_my_thing;
if (requeue)
        return; // to slow_work
```

==========
ITEM OPERATIONS
==========

Each work item requires a table of operations of type struct slow_work_ops. Only ->execute() is required; the getting and putting of a reference and the describing of an item are all optional.

(*) Get a reference on an item:

```
int (*get_ref)(struct slow_work *work);
```

This allows the thread pool to attempt to pin an item by getting a reference on it. This function should return 0 if the reference was granted, or a -ve error otherwise. If an error is returned, slow_work_enqueue() will fail.

The reference is held whilst the item is queued and whilst it is being executed. The item may then be requeued with the same reference held, or

the reference will be released.

(*) Release a reference on an item:

```
void (*put_ref)(struct slow_work *work);
```

This allows the thread pool to unpin an item by releasing the reference on it. The thread pool will not touch the item again once this has been called.

(*) Execute an item:

```
void (*execute)(struct slow_work *work);
```

This should perform the work required of the item. It may sleep, it may perform disk I/O and it may wait for locks.

(*) View an item through /proc:

```
void (*desc)(struct slow_work *work, struct seq_file *m);
```

If supplied, this should print to 'm' a small string describing the work the item is to do. This should be no more than about 40 characters, and shouldn't include a newline character.

See the 'Viewing executing and queued items' section below.

==================
POOL CONFIGURATION
==================

The slow-work thread pool has a number of configurables:

(*) /proc/sys/kernel/slow-work/min-threads

The minimum number of threads that should be in the pool whilst it is in use. This may be anywhere between 2 and max-threads.

(*) /proc/sys/kernel/slow-work/max-threads

The maximum number of threads that should in the pool. This may be anywhere between min-threads and 255 or NR_CPUS * 2, whichever is greater.

(*) /proc/sys/kernel/slow-work/vslow-percentage

The percentage of active threads in the pool that may be used to execute very slow work items. This may be between 1 and 99. The resultant number is bounded to between 1 and one fewer than the number of active threads. This ensures there is always at least one thread that can process very slow work items, and always at least one thread that won't.

==================================
VIEWING EXECUTING AND QUEUED ITEMS
==================================

If CONFIG_SLOW_WORK_DEBUG is enabled, a debugfs file is made available:

```
/sys/kernel/debug/slow_work/runqueue
```

through which the list of work items being executed and the queues of items to be executed may be viewed. The owner of a work item is given the chance to add some information of its own.

The contents look something like the following:

```
THR PID   ITEM ADDR        FL MARK  DESC
=== ===== ================ == ===== ==========
  0  3005 ffff880023f52348  a 952ms FSC: OBJ17d3: LOOK
  1  3006 ffff880024e33668  2 160ms FSC: OBJ17e5 OP60d3b: Write1/Store fl=2
  2  3165 ffff8800296dd180  a 424ms FSC: OBJ17e4: LOOK
  3  4089 ffff8800262c8d78  a 212ms FSC: OBJ17ea: CRTN
  4  4090 ffff88002792bed8  2 388ms FSC: OBJ17e8 OP60d36: Write1/Store fl=2
  5  4092 ffff88002a0ef308  2 388ms FSC: OBJ17e7 OP60d2e: Write1/Store fl=2
  6  4094 ffff88002abaf4b8  2 132ms FSC: OBJ17e2 OP60d4e: Write1/Store fl=2
  7  4095 ffff88002bb188e0  a 388ms FSC: OBJ17e9: CRTN
vsq     - ffff880023d99668  1 308ms FSC: OBJ17e0 OP60f91: Write1/EnQ fl=2
vsq     - ffff8800295d1740  1 212ms FSC: OBJ16be OP4d4b6: Write1/EnQ fl=2
vsq     - ffff880025ba3308  1 160ms FSC: OBJ179a OP58dec: Write1/EnQ fl=2
vsq     - ffff880024ec83e0  1 160ms FSC: OBJ17ae OP599f2: Write1/EnQ fl=2
vsq     - ffff880026618e00  1 160ms FSC: OBJ17e6 OP60d33: Write1/EnQ fl=2
vsq     - ffff880025a2a4b8  1 132ms FSC: OBJ16a2 OP4d583: Write1/EnQ fl=2
vsq     - ffff880023cbe6d8  9 212ms FSC: OBJ17eb: LOOK
vsq     - ffff880024d37590  9 212ms FSC: OBJ17ec: LOOK
vsq     - ffff880027746cb0  9 212ms FSC: OBJ17ed: LOOK
vsq     - ffff880024d37ae8  9 212ms FSC: OBJ17ee: LOOK
vsq     - ffff880024d37cb0  9 212ms FSC: OBJ17ef: LOOK
vsq     - ffff880025036550  9 212ms FSC: OBJ17f0: LOOK
vsq     - ffff8800250368e0  9 212ms FSC: OBJ17f1: LOOK
vsq     - ffff880025036aa8  9 212ms FSC: OBJ17f2: LOOK
```

In the 'THR' column, executing items show the thread they're occupying and queued threads indicate which queue they're on. 'PID' shows the process ID of a slow-work thread that's executing something. 'FL' shows the work item flags. 'MARK' indicates how long since an item was queued or began executing. Lastly, the 'DESC' column permits the owner of an item to give some information.