14. The Thread Pool & Asynchronous Tasks

14.1 Queues

There is a flaw in our server since the introduction of the sorted set data type: the deletion of keys. If the size of a sorted set is huge, it can take a long time to free its nodes and the server is stalled during the destruction of the key. This can be easily fixed by using multi-threading to move the destructor away from the main thread.

Firstly, we introduce the "thread pool", which is literally a pool of threads. The thread from the pool consumes tasks from a queue and executes them. It is trivial to code a multi-producer multi-consumer queue using `pthread` APIs. (Although there is only a single producer in our case.)

The relevant `pthread` primitives are `pthread_mutex_t` and `pthread_cond_t`; they are called the mutex and the condition variable respectively. If you are unfamiliar with them, it is advised to get some education on multi-threading after reading this chapter. (Such as manpages of the `pthread` APIs, textbooks on operating systems, online courses, etc.)

Here is a really short introduction to the two `pthread` primitives:

- The queue is accessed by multiple threads (both the producer and consumers), so it needs the protection of a mutex, obviously.
- The consumer threads should be sleeping when idle, and only be waken up when the queue is not empty, this is the job of the condition variable.

14.2 The Implementation

The thread pool data type is defined as follows:

```
struct Work {
    void (*f)(void *) = NULL;
    void *arg = NULL;
};

struct TheadPool {
    std::vector<pthread_t> threads;
    std::deque<Work> queue;
    pthread_mutex_t mu;
    pthread_cond_t not_empty;
};
```

The `thread_pool_init` is for initialization and starting threads. `pthread` types are initialized by `pthread_xxx_init` functions and the `pthread_create` starts a thread with the target function `worker`.

```
void thread_pool_init(TheadPool *tp, size_t num_threads) {
    assert(num_threads > 0);

    int rv = pthread_mutex_init(&tp->mu, NULL);
    assert(rv == 0);
    rv = pthread_cond_init(&tp->not_empty, NULL);
    assert(rv == 0);

    tp->threads.resize(num_threads);
    for (size_t i = 0; i < num_threads; ++i) {
        int rv = pthread_create(&tp->threads[i], NULL, &worker, tp);
        assert(rv == 0);
    }
}
```

The consumer code:

```
static void *worker(void *arg) {
    TheadPool *tp = (TheadPool *)arg;
    while (true) {
        pthread_mutex_lock(&tp->mu);
        // wait for the condition: a non-empty queue
        while (tp->queue.empty()) {
            pthread_cond_wait(&tp->not_empty, &tp->mu);
```

```
        }

        // got the job
        Work w = tp->queue.front();
        tp->queue.pop_front();
        pthread_mutex_unlock(&tp->mu);

        // do the work
        w.f(w.arg);
    }
    return NULL;
}
```

The producer code:

```
void thread_pool_queue(TheadPool *tp, void (*f)(void *), void *arg) {
    Work w;
    w.f = f;
    w.arg = arg;

    pthread_mutex_lock(&tp->mu);
    tp->queue.push_back(w);
    pthread_cond_signal(&tp->not_empty);
    pthread_mutex_unlock(&tp->mu);
}
```

14.3 pthread APIs

The explanation:

1. For both the producer and consumers, the queue access code is surrounded by the `pthread_mutex_lock` and the `pthread_mutex_unlock`, only one thread can access the queue at once.
2. After a consumer acquired the mutex, check the queue:
    - If the queue is not empty, grab a job from the queue, release the mutex and do the work.
    - Otherwise, release the mutex and go to sleep, the sleep can be wakened later by the condition variable. This is accomplished via a single `pthread_cond_wait` call.
3. After the producer puts a job into the queue, the producer calls the `pthread_cond_signal` to wake up a potentially sleeping consumer.

4. After a consumer woken up from the `pthread_cond_wait`, the mutex is held again automatically. The consumer must check for the condition *again* after waking up, if the condition (a non-empty queue) is not satisfied, go back to sleep.

The use of the condition variable needs some more explanations: The `pthread_cond_wait` function is *always* inside a loop checking for the condition. This is because the condition could be changed by other consumers before the wakening consumer grabs the mutex; the mutex is not transferred from the signaler to the to-be-waked consumer! It is probably a mistake if you see a condition variable used without a loop.

A concrete sequence to help you understand the use of condition variables:

1. The producer signals.
2. The producer releases the mutex.
3. Some consumer grabs the mutex and empties the queue.
4. A consumer wakes up from the producer's signal and grabs the mutex, but the queue is empty!

Note that the `pthread_cond_signal` doesn't need to be protected by the mutex, singaling after releasing the mutex is also correct.

14.4 Integrating with the Server

The thread pool is done. Let's add that to our server:

```
// global variables
static struct {
    HMap db;
    // a map of all client connections, keyed by fd
    std::vector<Conn *> fd2conn;
    // timers for idle connections
    DList idle_list;
    // timers for TTLs
    std::vector<HeapItem> heap;
    // the thread pool
    TheadPool tp;
} g_data;
```

Inside the `main` function:

```
    // some initializations
    dlist_init(&g_data.idle_list);
    thread_pool_init(&g_data.tp, 4);
```

The `entry_del` function is modified: It will put the destruction of large sorted sets into the thread pool. And the thread pool is only for the large ones since multi-threading has some overheads too.

```
// deallocate the key immediately
static void entry_destroy(Entry *ent) {
    switch (ent->type) {
    case T_ZSET:
        zset_dispose(ent->zset);
        delete ent->zset;
        break;
    }
    delete ent;
}

static void entry_del_async(void *arg) {
    entry_destroy((Entry *)arg);
}

// dispose the entry after it got detached from the key space
static void entry_del(Entry *ent) {
    entry_set_ttl(ent, -1);

    const size_t k_large_container_size = 10000;
    bool too_big = false;
    switch (ent->type) {
    case T_ZSET:
        too_big = hm_size(&ent->zset->hmap) > k_large_container_size;
        break;
    }

    if (too_big) {
        thread_pool_queue(&g_data.tp, &entry_del_async, ent);
    } else {
        entry_destroy(ent);
```

```
    }
}
```

Exercises:

1. The semaphore is often introduced as a multi-threading primitive instead of the condition variable and the mutex. Try to implement the thread pool using the semaphore.
2. Some fun exercises to help you understand these primitives further:
   1. Implement the mutex using the semaphore. (Trivial)
   2. Implement the semaphore using the condition variable. (Easy)
   3. Implement the condition variable using only mutexes. (Intermediate)
   4. Now that you know these primitives are somewhat equivalent, why should you prefer one to another?

Source code:

- 14_server.cpp
- avl.cpp
- avl.h
- common.h
- hashtable.cpp
- hashtable.h
- heap.cpp
- heap.h
- list.h
- thread_pool.cpp
- thread_pool.h
- zset.cpp
- zset.h

Check it out