Pthreads Programming Assignment

Team 67

108062128 林育丞

108062135 呂佳恩

分工表

林育丞	Research, Question 回答整理, Report 整理
呂佳恩	Implementation, Report 整理

‘

(1) Implementation

1. ts_queue.hpp

We start our implementation from the file ts_queue.hpp since we find it reasonable to first finish the structure of it. After we finish the structure, using it in other files would be more intuitive

```
TSQueue<T>::TSQueue(int buffer_size) : buffer_size(buffer_size) {
    // TODO(V): implements TSQueue constructor

    buffer = new T[buffer_size];
    size = 0;
    head = 0;
    tail = 0;
    pthread_mutex_init(&mutex, NULL);
    pthread_cond_init(&cond_enqueue, NULL);
    pthread_cond_init(&cond_dequeue, NULL);
}
```

For the constructor, we initialize some values that are required, and call pthread initialization methods for some conditions and mutex locks.
For the destructor, we do the opposite as the constructor and call the destroy method for the locks and conditions.

```
void TSQueue<T>::enqueue(T item) {
    // TODO(V): enqueues an element to the end of the queue
    pthread_mutex_lock(&mutex);

    while(size >= buffer_size)
        pthread_cond_wait(&cond_enqueue, &mutex);

    buffer[tail] = item;
    tail = (tail+1) % buffer_size;
    size+=1;

    pthread_cond_signal(&cond_dequeue);
    pthread_mutex_unlock(&mutex);
}
```

For the enqueue, we have to lock the operation first, then we check if there are any free space, if there is none, lock the operation in the while loop until a space is available. If there is free space, we put the item into the tail of the buffer, then move the tail, Since we want to wrap around to the front at the end, we use mod operation to calculate. And then increment the size of the buffer.

```
T TSQueue<T>::dequeue() {
    // TODO(V): dequeues the first element of the queue
    pthread_mutex_lock(&mutex);

    while(size <= 0)
        pthread_cond_wait(&cond_dequeue, &mutex);

    T item = buffer[head];
    head = (head+1)%buffer_size;
    size-=1;

    pthread_cond_signal(&cond_enqueue);
    pthread_mutex_unlock(&mutex);
    return item;
}
```

Dequeue is somewhat similar, however, we move the head into the next space and return the head item from the buffer.
The get_size( ) simply returns the size.

2. writer.hpp

The writer::start( ) is fairly simple and repetitive as we do the same operation of pthread_create in other instances such as in comsumer.hpp.

```
void* Writer::process(void* arg) {
    // TODO(V): implements the Writer's work
    Writer *writer = (Writer*)arg;
    while(writer->expected_lines--){
        Item *item = new Item;
        item = writer->output_queue->dequeue();
        writer->ofs << *item;
        delete item;
    }
    return nullptr;
}
```

The writer::process reads the file one line after the other and then outputs to file

3. Producer.hpp

We implement the start similar to writer with pthread_create.

```
void* Producer::process(void* arg) {
    // TODO(V): implements the Producer's work
    Producer *producer = (Producer *)arg;
    while(1){
        if(producer->input_queue->get_size()>0){
            Item *item = new Item;
            item = producer->input_queue->dequeue();
            unsigned long long trans = producer->transformer->producer_transf
            item->val = trans;
            producer->worker_queue->enqueue(item);
            delete item;
        }
```

Since the producer needs to run from the start to the end, always checking if there are item needing production, we use a everlasting while loop to implement it, if the size of input_que is not 0, we extract the item and transform it, then put it in the worker_queue.

4. Consumer.hpp

Similar to Producer and writer, we implement start with pthread_create

```
int Consumer::cancel() {
    // TODO(V): cancels the consumer thread
    is_cancel = true;
    return pthread_cancel(t);
}
```

Cancel is implemented similarly as a thread in thread.h, however, we have to set the is_cancel to true.

```
if (consumer->worker_queue->get_size() != 0){
    Item *item = new Item;
    item = consumer->worker_queue->dequeue();
    unsigned long long trans = consumer->transformer->consumer_transform(item->opcode, item->val);
    item->val = trans;
    consumer->output_queue->enqueue(item);
    delete item;
}
```

We dequeue and item from the worker queue and perform a transform and then put it in the output queue.

5. Consumer_controller.hpp

```
// TODO(V): implements the ConsumerController's work
// controls the number of consumers
// # of items in the worker queue exceeds 80%, # of consumer thread++
// else if fall behind 20%, # of consumer thread--
ConsumerController *controller = (ConsumerController *)arg;
struct timeval time_start, time_end;
gettimeofday(&time_start, NULL);
gettimeofday(&time_end, NULL);
while (1)
{
    gettimeofday(&time_start, NULL);
    int sec = time_start.tv_sec - time_end.tv_sec;
    int usec = time_start.tv_usec - time_end.tv_usec;
    int tot_time = (sec * 1000000) + usec;
    if (tot_time >= controller->check_period)
    {
        gettimeofday(&time_end, NULL);
        if (controller->worker_queue->get_size() > controller->high_threshold)
        {
            Consumer *new_con = new Consumer(controller->worker_queue, controller->writer_queue, controller->transformer);
            printf("Scaling up consumers from %d to %d.\n", controller->consumers.size(), controller->consumers.size() + 1);
            new_con->start();
            controller->consumers.push_back(new_con);
        }
        else if (controller->worker_queue->get_size() < controller->low_threshold && controller->consumers.size() > 1)
        {
            Consumer *del_con = controller->consumers.back();
            controller->consumers.pop_back();
            int cancel_result = del_con->cancel();
            if (!cancel_result)
            {
                printf("Scaling down consumers from %d to %d.\n", controller->consumers.size() + 1, controller->consumers.size());
            }
            else
            {
                printf("Cancel Failed");
            }
        }
        gettimeofday(&time_start, NULL);
    }
    gettimeofday(&time_start, NULL);
}
```

Start( ) is still the same as before, implemented with pthread_create.

We use a struct included in sys/time.h header file that helps us record the time frame. If the time that elapsed exceeds the checkperiod, we perform the operation, otherwise, we continue to iterate through the while loop.

If the size of the worker queue exceeds 80%, we have to increase the number of consumer threads, however, if the total falls below the 20% mark, we have to decrease the number of consumer threads.

We print different messages according to the operation.

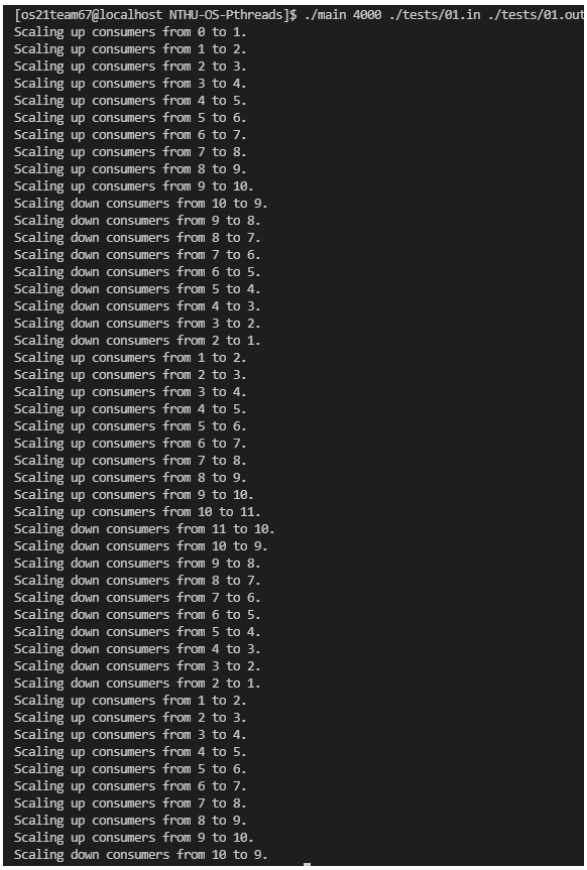
6. Main.cpp

```
// TODO: implements main function
int low_threshold = WORKER_QUEUE_SIZE * CONSUMER_CONTROLLER_LOW_THRESHOLD_PERCENTAGE /100;
int high_threshold = WORKER_QUEUE_SIZE * CONSUMER_CONTROLLER_HIGH_THRESHOLD_PERCENTAGE /100
Transformer* transformer = new Transformer;

TSQueue<Item*>* Input_Queue = new TSQueue<Item *>;
TSQueue<Item*>* Worker_Queue = new TSQueue<Item *>;
TSQueue<Item*>* Writer_Queue= new TSQueue<Item *>;

Reader* reader = new Reader(n, input_file_name, Input_Queue);
Writer* writer = new Writer(n, output_file_name, Writer_Queue);
Producer* producer[4];

ConsumerController* controller = new ConsumerController(Worker_Queue, Writer_Queue, transfo
reader->start();
writer->start();

controller->start();

for(int i = 0; i < 4; i++){
    producer[i]= new Producer(Input_Queue, Worker_Queue, transformer);
    producer[i]->start();
}

reader->join();
writer->join();

for(int i = 0; i < 4; i++){
    delete producer[i];
}
delete writer;
delete reader;
delete transformer;
delete Input_Queue;
delete Worker_Queue;
delete Writer_Queue;
delete controller;
return 0;
```

Initialize different values and start worker, producer, writer, consumer_controller, according to the spec.

2. Experiment

We run some different test, which includes all the required ones and a few others. We arrived at some conclusions with a few data points, the data are measured in time(seconds).



1. Different values of CONSUMER_CONTROLLER_CHECK_PERIOD
   The default value provided was 1000000 milliseconds, we changed it into 10, which was a drastic change, we believed that this could result in a huge difference if the values of CONSUMER_CONTROLLER_CHECK_PERIOD effects the execution time.
   With n = 200, our time improved from 7.7seconds to 6.3 seconds, which was a 17.87% decrease of time. However, we also noticed that there were way more scaling operations, which causes overhead, we believe that there is a sweet spot that the check_period could be set to in order to maximize the performance.

2. Different values of THRESHOLD_HIGH-THRESHOLD_LOW
   The original ratio provided was an 80-20 pair.
   We changed the High-Low into the following pairs 90-10, 80-20, 70-30, 60-40
   With n = 200, we observed an average execution time of 8 , 7.7, 7.5, 7.2 seconds, the 60-40 ratio has a 6.49% decrease in runtime, this value is barely noticeable, and can even considered error.
   When we observe the scaling operation, there was little difference regarding the scaling operations, we believe that the THRESHOLD values play little to no part in the performance.
3. Different values of WORKER_QUEUE_SIZE
   The initial value was 200.
   We changed values into 10, 100, 200, 1000 to observe the performance.
   The execution time regarding this would be 4.3, 5.7, 6.2, 7.5.
   We then observe the consumer values output by our code, we can se that if the job count start small and increase steadily, the scaling operation does not vary much, however, if the QUEUE_SIZE is small and the number of jobs change drastically, the scaling operation has to happen very often.
4. WRITER_QUEUE_SIZE is small.
   The original value set is at 4000, we drop it down to 4.
   The time recorded was 7.7 for 4000 queue size and 7.2 for the 4
   I feel like in this testcase, it doesn't matter much around the WRITER_QUEUE_SIZE, however, if we think about it, if the writer_queue_size is small, it gets full easily, and other writers have to wait for their chance, and might cause trouble if they wait for too long.
5. READER_QUEUE_SIZE is small
   The original value is set at 200, we try to drop them to 1, and we try to raise it to 2000 to make the difference dramatical.
   For 2000, the execution time was around 10seconds, for 200, it was around 7.4, and for 1, it was around 7.5
   We discover that it might be better to have the READER_QUEUE value set smaller, but not too small.

3. Difficulties

This project was hard since there were little to no resources that we could take as a reference online, therefore, we spent most of our time trying to understand how the implementation was expected to perform, after that, each element of the code was fairly easy, however, we found out that there might be strange bugs that occurred and we have to rethink our approach to the question.

4. FEEDBACK

I would really hope that the deadline would be moved 2 weeks earlier or a few days later since it was the same time as MP4, also, we had several midterm and other assignments crammed together.