Pthreads_Report_36

109062233蘇裕恆：

50%trace code 50% implementation

109062320朱季葳：

50%trace code 50% implementation

Implementation

```
int main(int argc, char** argv) {
    assert(argc == 4);

    int n = atoi(argv[1]);
    std::string input_file_name(argv[2]);
    std::string output_file_name(argv[3]);

    // TODO: implements main function
    TSQueue<Item*>* input_queue = new TSQueue<Item*>(READER_QUEUE_SIZE);
    TSQueue<Item*>* worker_queue = new TSQueue<Item*>(WORKER_QUEUE_SIZE);
    TSQueue<Item*>* output_queue = new TSQueue<Item*>(WRITER_QUEUE_SIZE);

    Transformer* transformer = new Transformer;
    Reader* reader = new Reader(n, input_file_name, input_queue);
    Writer* writer = new Writer(n, output_file_name, output_queue);

    Producer* p1 = new Producer(input_queue, worker_queue, transformer);
    Producer* p2 = new Producer(input_queue, worker_queue, transformer);
    Producer* p3 = new Producer(input_queue, worker_queue, transformer);
    Producer* p4 = new Producer(input_queue, worker_queue, transformer);
```

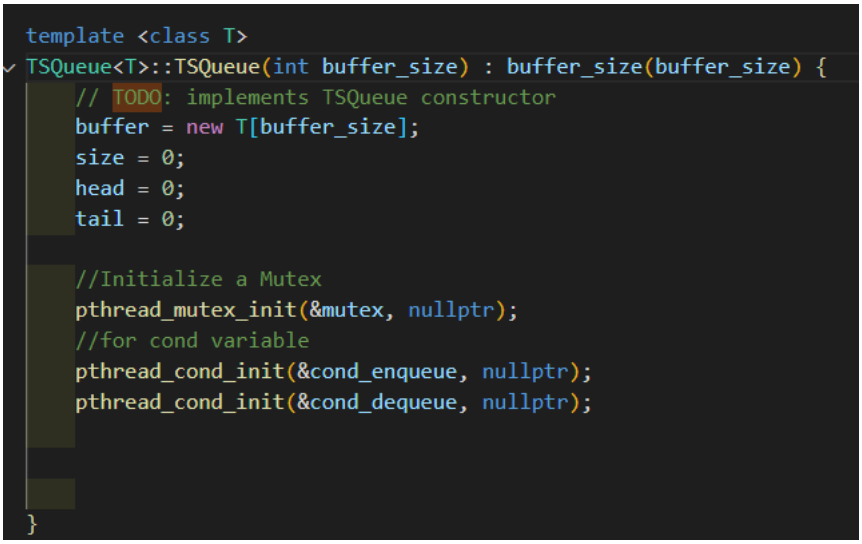
在main裡面 因為他總共有三個queue (input queue , working queue , output queue) 分別是由 reader , (consumer & producer) , writer 掌握。
其他包括了transformer、reader、writer、producer、consumerController 等，並將variables傳入對應得constructor。

```
42
43
44      reader->start();
45      writer->start();
46
47      p1->start();
48      p2->start();
49      p3->start();
50      p4->start();
51      consumer_ctrler->start();
52
53      reader->join();
54      writer->join();
55
```

接著將這些 thread 呼叫 start()開始execute，最後需要呼叫 reader 和 writer 的 join()，等待這 2 個 thread 運行完畢，在最後 main 結束後，並將所有東西 delete 掉。

ts_queue

因此在一開始，我們需要將三個 TSqueue new 出來，而在 TSQueue，裡面 因為要確保其正確性 因此我們需要使用 Pthread

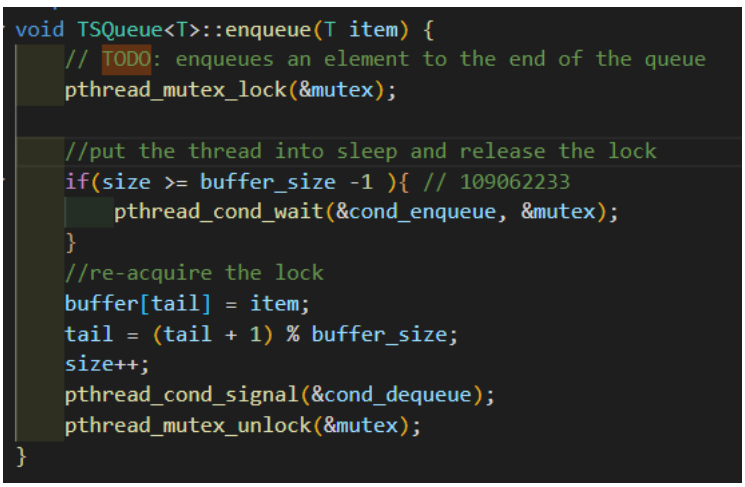


```
template <class T>
TSQueue<T>::TSQueue(int buffer_size) : buffer_size(buffer_size) {
    // TODO: implements TSQueue constructor
    buffer = new T[buffer_size];
    size = 0;
    head = 0;
    tail = 0;

    //Initialize a Mutex
    pthread_mutex_init(&mutex, nullptr);
    //for cond variable
    pthread_cond_init(&cond_enqueue, nullptr);
    pthread_cond_init(&cond_dequeue, nullptr);


}
```

可以看到 一開始我們需要initialate一些同步化工具，包括了mutex與condition variables。 想當然爾，我們在最後的時候必須將他們destruct跟destroy掉，

```
        delete[] buffer;
        size = 0;
        head = 0;
        tail = 0;

        pthread_mutex_destroy(&mutex);
        pthread_cond_destroy(&cond_enqueue);
        pthread_cond_destroy(&cond_dequeue);
```

接著我們看一下究竟是怎麼確保enqueue跟dequeue是不會產生race condition的。



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

    //put the thread into sleep and release the lock
    if(size >= buffer_size -1 ){ // 109062233
        pthread_cond_wait(&cond_enqueue, &mutex);
    }
    //re-acquire the lock
    buffer[tail] = item;
    tail = (tail + 1) % buffer_size;
    size++;
    pthread_cond_signal(&cond_dequeue);
    pthread_mutex_unlock(&mutex);
}
```

可以看到 我們一開始將mutex lock 住，代表進入critical section。如果他這時候的size >= (buffer_size - 1) 我們就要將他put to sleep。
若沒有的話，我們就會將buffer[tail]存入資訊，並將tail移動位置且將size++。注意，我們在這邊也會去試著將dequeue的codition variable signal。 因為若是dequeue的thread也被put to sleep的話，條件就滿足可以讓她dequeue了。

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

    if(size <= 0){
        pthread_cond_wait(&cond_dequeue, &mutex);
    }
    T ret_T;
    ret_T = buffer[head];
    head = (head + 1) % buffer_size;
    size--;

    pthread_cond_signal(&cond_enqueue);

    pthread_mutex_unlock(&mutex);

    return ret_T;
}
```

同樣的，在dequeue裡面，想法是相同的，如果他這時候的size <= 0 我們就要將他put to sleep。若沒有的話，我們就會將buffer[head]的資訊取出，並將head移動位置且將size−。注意，也會去試著將equeue的codition variable signal。

```cpp
template <class T>
int TSQueue<T>::get_size() {
    // TODO: returns the size of the queue
    return size;
}
```

同時，因為ConsumerController會需要依據size 來create process，因此我們會需要一個getSize() 函數來確保可以得到他size的資訊。

writer

```cpp
void Writer::start() {
    // TODO: starts a Writer thread
    pthread_create(&t, 0, Writer::process, (void*)this);
}

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

    }
    return nullptr;
}
```

在Wirter中，在start()中我們需要呼叫 pthread_create()，參數放入
void* Writer::process(void* arg)。即可以完成。
而在process裡面我們則需要不斷從output_queue call dequeue()取出 item，並將其寫入到file，直到所有原本給定的lines並結束。

consumer_controller

```
void ConsumerController::start() {
    // TODO: starts a ConsumerController thread
    pthread_create(&t, 0, ConsumerController::process, (void*) this);
}
```

在ConsumerController中，在start()中我們需要呼叫pthread_create()，參數放入ConsumerController::process(void* arg)。即可以完成。

```
void* ConsumerController::process(void* arg) {
    // TODO: implements the ConsumerController's work
    //usleep
    //https://blog.csdn.net/wangquan1992/article/details/103612109
    //chrono
    //https://www.pluralsight.com/blog/software-development/how-to-measure-execution-time-intervals-in-c--
    ConsumerController* consumer_ctrler = (ConsumerController*)arg;
    while(1){
        int worker_queue_size = consumer_ctrler->worker_queue->get_size();
        int high_thres = consumer_ctrler->high_threshold;
        int low_thres = consumer_ctrler->low_threshold;
        if(worker_queue_size > high_thres){
            Consumer* new_consumer = new Consumer(consumer_ctrler->worker_queue,
                                                  consumer_ctrler->writer_queue,
                                                  consumer_ctrler->transformer);
            new_consumer->start();
            consumer_ctrler->consumers.push_back(new_consumer);
            printf("Scaling up consumers from %d to %d\n", consumer_ctrler->consumers.size() - 1,
             consumer_ctrler->consumers.size());
        }
        else if((worker_queue_size < low_thres) && (consumer_ctrler->consumers.size() > 1)){
            consumer_ctrler->consumers.back()->cancel();
            consumer_ctrler->consumers.pop_back();
            printf("Scaling down consumers from %d to %d\n", consumer_ctrler->consumers.size() + 1,
            consumer_ctrler->consumers.size());
        }

        usleep(consumer_ctrler->check_period);
    }
    return nullptr;
```

在process中，我們需要periodically的看他的size並且決定是要將consumer增加或是將comsumer減少。

```
1   (WORKER_QUEUE_SIZE * CONSUMER_CONTROLLER_LOW_THRESHOLD_PERCENTAGE / 100),
2   (WORKER_QUEUE_SIZE * CONSUMER_CONTROLLER_HIGH_THRESHOLD_PERCENTAGE / 100));
```

我們可以看到 上述就是我們傳出的 int low_threshold, int high_threshold，而如果超過的話就會將一個consumer create出來並且給予他一樣的file。之後會call start並將他push進去comsumer vector之中。
接著我們就印出助教所需要的訊息。
如果少於threshold 的話並且就會將一個consumer delete (cancel) 並且將它從comsumer vector之中 pop出來。
最後 因為我們要用到periodic的方式，因此我們使用的usleep()來達成。

consumer

start() -> same

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

cancel() 將is_cancel設為true。之後就會將這個process刪除掉。

```cpp
void* Consumer::process(void* arg) {
    Consumer* consumer = (Consumer*)arg;
    //https://blog.csdn.net/hslinux/article/details/7929182
    //A cancellation request is deferred
    //until the thread next calls a function that is a cancellation point
    pthread_setcanceltype(PTHREAD_CANCEL_DEFERRED, nullptr);

    while (!consumer->is_cancel) {
        //disable cancellation for a while,
        //so that we don't immediately react to a cancellation request
        pthread_setcancelstate(PTHREAD_CANCEL_DISABLE, nullptr);

        // TODO: implements the Consumer's work
        Item* it = consumer->worker_queue->dequeue();
        it -> val = consumer->transformer->consumer_transform(it->opcode, it->val);
        consumer->output_queue->enqueue(it);


        //re-enable cancellation
        pthread_setcancelstate(PTHREAD_CANCEL_ENABLE, nullptr);
    }

    delete consumer;

    return nullptr;
}
```

如果他今天沒有被cancel的話，就會掠過while loop並將consumer delete掉。
在這邊，為了避免process在執行途中被刪掉，我們使用了幾個pthread中的
pthread_setcanceltype(PTHREAD_CANCEL_DEFERRED, nullptr);
pthread_setcancelstate(PTHREAD_CANCEL_DISABLE, nullptr);
pthread_setcancelstate(PTHREAD_CANCEL_ENABLE, nullptr);
來確保他不會被interrupt。

producer

```cpp
void Producer::start() {
    // TODO: starts a Producer thread
    pthread_create(&t, 0, Producer::process, (void*) this);
}

void* Producer::process(void* arg) {
    // TODO: implements the Producer's work
    Producer* producer = (Producer*)arg;
    pthread_setcanceltype(PTHREAD_CANCEL_DEFERRED, nullptr);

    while(1){
        pthread_setcancelstate(PTHREAD_CANCEL_DISABLE, nullptr);
        Item* it = producer->input_queue->dequeue();
        it -> val = producer->transformer->producer_transform(it->opcode, it->val);
        producer->worker_queue->enqueue(it);
        pthread_setcancelstate(PTHREAD_CANCEL_ENABLE, nullptr);
    }

    return nullptr;
}
```

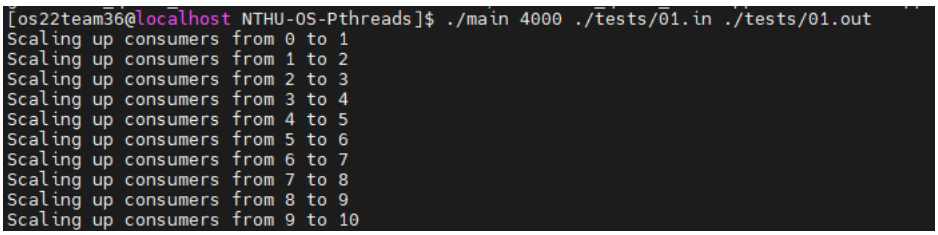
跟writer類似，但是他的做法是不斷地從input queue裡面找出item並且將item藉由producer transdormtransform()傳入item的 opcode , val來決定新的val值，最後並將它寫入worker queue 等待consumer。

Experiment

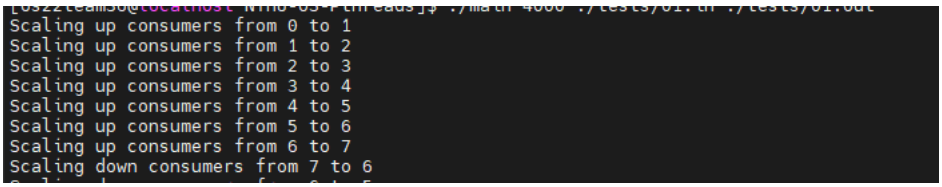
1. Different values of CONSUMER_CONTROLLER_CHECK_PERIOD.

我們在evaulate的時候使用 testcase01 並取用4000行input作為baseline

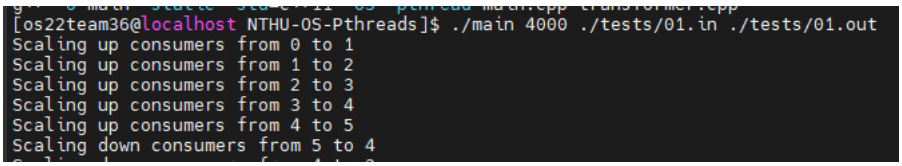
testcase 100000 :



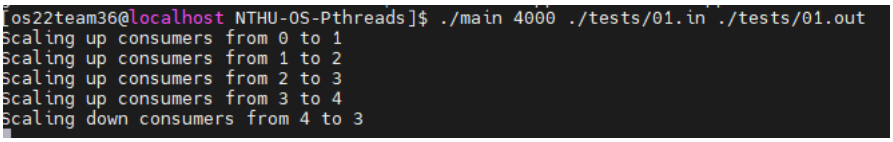
testcase 200000 :



testcase 400000 :



testcase 800000 :

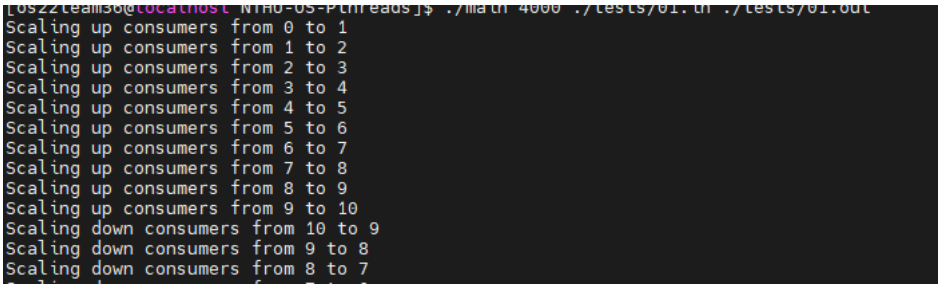


我們可以看到 因為越短的時間意味著他能更快速的依據目前的load來做 反應，因此若是越小則comsumers數量越多。

2. Different values of CONSUMER_CONTROLLER_LOW_THRESHOLD_PERCENTAGE and CONSUMER_CONTROLLER_HIGH_THRESHOLD_PERCENTAGE.

low / high

testcase 20 / 80 :



testcase 0 / 80 :

```
[os22team36@localhost NTHU-OS-Pthreads]$ ./main 4000 ./tests/01.in ./tests/01.out
Scaling up consumers from 0 to 1
Scaling up consumers from 1 to 2
Scaling up consumers from 2 to 3
Scaling up consumers from 3 to 4
Scaling up consumers from 4 to 5
Scaling up consumers from 5 to 6
Scaling up consumers from 6 to 7
Scaling up consumers from 7 to 8
Scaling up consumers from 8 to 9
Scaling up consumers from 9 to 10
[os22team36@localhost NTHU-OS-Pthreads]$ ./scripts/verify --output ./tests/01.out --
```

我們可以看到 若將 CONSUMER_CONTROLLER_LOW_THRESHOLD_PERCENTAGEW挑整下降，因為他的門檻不會到，因此不會則就不會做到scaling down ，若是把它提高 則沒有甚麼不一樣，

testcase 20 / 80 :

```
Scaling up consumers from 0 to 1
Scaling up consumers from 1 to 2
Scaling up consumers from 2 to 3
Scaling up consumers from 3 to 4
Scaling up consumers from 4 to 5
Scaling up consumers from 5 to 6
Scaling up consumers from 6 to 7
Scaling up consumers from 7 to 8
Scaling up consumers from 8 to 9
Scaling up consumers from 9 to 10
Scaling down consumers from 10 to 9
Scaling down consumers from 9 to 8
Scaling down consumers from 8 to 7
```

testcase 20 / 90 :

```
[os22team36@localhost NTHU-OS-Pthreads]$ ./main 4000 ./tests/01.in ./tests/01.out
Scaling up consumers from 0 to 1
Scaling up consumers from 1 to 2
Scaling up consumers from 2 to 3
Scaling up consumers from 3 to 4
Scaling up consumers from 4 to 5
Scaling up consumers from 5 to 6
Scaling up consumers from 6 to 7
Scaling up consumers from 7 to 8
Scaling up consumers from 8 to 9
Scaling down consumers from 9 to 8
Scaling down consumers from 8 to 7
```

testcase 20 / 30 :

```
[os22team36@localhost NTHU-OS-Pthreads]$ ./main 4000 ./tests/01.in ./tests/01.out
Scaling up consumers from 0 to 1
Scaling up consumers from 1 to 2
Scaling up consumers from 2 to 3
Scaling up consumers from 3 to 4
Scaling up consumers from 4 to 5
Scaling up consumers from 5 to 6
Scaling up consumers from 6 to 7
Scaling up consumers from 7 to 8
Scaling up consumers from 8 to 9
Scaling up consumers from 9 to 10
Scaling up consumers from 10 to 11
Scaling down consumers from 11 to 10
Scaling down consumers from 10 to 9
Scaling down consumers from 9 to 8
```

我們可以看到 若將 CONSUMER_CONTROLLER_HIGH_THRESHOLD_PERCENTAGE提高則就，會造成比較晚的comsumer增加與比較少的comsumer，因為他的threshold被提高了。若是調為30，scaling up 則會跑到11，因為他較早開始scale up。

3. Different values of WORKER_QUEUE_SIZE.

testcase 200 :

```
Scaling up consumers from 0 to 1
Scaling up consumers from 1 to 2
Scaling up consumers from 2 to 3
Scaling up consumers from 3 to 4
Scaling up consumers from 4 to 5
Scaling up consumers from 5 to 6
Scaling up consumers from 6 to 7
Scaling up consumers from 7 to 8
Scaling up consumers from 8 to 9
Scaling up consumers from 9 to 10
Scaling down consumers from 10 to 9
Scaling down consumers from 9 to 8
Scaling down consumers from 8 to 7
```

testcase 400 :

```
[os22team36@localhost NTHU-OS-Pthreads]$ ./main 4000 ./tests/01.in ./tests/01.out
Scaling up consumers from 0 to 1
Scaling up consumers from 1 to 2
Scaling up consumers from 2 to 3
Scaling up consumers from 3 to 4
Scaling up consumers from 4 to 5
Scaling up consumers from 5 to 6
Scaling up consumers from 6 to 7
Scaling up consumers from 7 to 8
Scaling up consumers from 8 to 9
Scaling down consumers from 9 to 8
Scaling down consumers from 8 to 7
Scaling down consumers from 7 to 6
Scaling down consumers from 6 to 5
```

testcase 800 :

```
[os22team36@localhost NTHU-OS-Pthreads]$ ./main 4000 ./tests/01.in ./tests/01.out
Scaling up consumers from 0 to 1
Scaling up consumers from 1 to 2
Scaling up consumers from 2 to 3
Scaling up consumers from 3 to 4
Scaling up consumers from 4 to 5
Scaling up consumers from 5 to 6
Scaling up consumers from 6 to 7
Scaling up consumers from 7 to 8
Scaling down consumers from 8 to 7
Scaling down consumers from 7 to 6
Scaling down consumers from 6 to 5
```

我們可以看到，當worker queue變多高的時候，因為變相的他的threshold達到的量及變高，因此越是高，他所產生生出的consumer的max就會變小。

4. What happens if WRITER_QUEUE_SIZE is very small?

   testcase 1 :

```
[os22team36@localhost NTHU-OS-Pthreads]$ time ./main 400 ./tests/01.in ./tests/01.out
Scaling up consumers from 0 to 1
Scaling up consumers from 1 to 2
Scaling up consumers from 2 to 3
Scaling up consumers from 3 to 4
Scaling up consumers from 4 to 5
Scaling up consumers from 5 to 6
Scaling up consumers from 6 to 7
Scaling down consumers from 7 to 6
```

   testcase 4000 :

```
[os22team36@localhost NTHU-OS-Pthreads]$ time ./main 400 ./tests/01.in ./tests/01.out
Scaling up consumers from 0 to 1
Scaling up consumers from 1 to 2
Scaling up consumers from 2 to 3
Scaling down consumers from 3 to 2
Scaling down consumers from 2 to 1
```

因為整個程式會被bound在output，因此他的worker queue會較擁擠，造成consumer的thread數量較高。

5. What happens if READER_QUEUE_SIZE is very small?

   testcase 1 :

```
[os22team36@localhost NTHU-OS-Pthreads]$ time ./main 4000 ./tests/01.in ./tests/01.out
Scaling up consumers from 0 to 1
Scaling up consumers from 1 to 2
Scaling up consumers from 2 to 3
Scaling up consumers from 3 to 4
Scaling up consumers from 4 to 5
Scaling up consumers from 5 to 6
Scaling up consumers from 6 to 7
Scaling up consumers from 7 to 8
Scaling up consumers from 8 to 9
Scaling down consumers from 9 to 8
Scaling down consumers from 8 to 7
```

因為整個程式會因為input size = 1 ，因此他的worker queue可能增加的不是那麼快，造成他的comsumer只增加到 9 就停下來了。

difficulties & feedback

在一開始的時候因為只有在課堂上聽過老師講述pthread library的應用以及一些實作的範例，但是並沒有實際操作過，所以一開始花了蠻多時間在研究library的API要傳入的東西，不過幸虧在Spec上面有明確的指出大概要做些什麼事情，所以在這次的作業上面比較沒有遇到太大的困難。