

# Pthread Report

Team 27 : 112062232賴允中、111070070林標松

## 分工表

組員	Report編寫	Implementation	Experiment
賴允中	Y	N	Y
林標松	Y	Y	N

## Table of Contents

1. Implementation ----- 2
3. Experiment ----- 24
3. Reflection ----- 33

# 1. Implementation

## 1. TSQueue

- *TSQueue::TSQueue*

在constructor裡，要去初始化 TSQueue class 裡有的 private 變數。

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

    // 初始化相關變數
    buffer = new T[buffer_size];
    size = 0;
    head = tail = 0;
    // 初始化 mutex 及 condition variable
    pthread_mutex_init(&mutex, 0);
    pthread_cond_init(&cond_enqueue, 0);
    pthread_cond_init(&cond_dequeue, 0);
}
```

首先，為 buffer 配出長度為傳入參數 buffer\_size 的陣列，並把 size 初始為 0（一開始 buffer 應該要是空的）。接著，因為採用 queue 來管理 buffer，初始化會用到的變數 head 和 tail 為 0。最後是 mutex lock 的部分，使用 `pthread_mutex_init()`，和 `pthread_cond_init()`，分別初始化 lock 和 condition variable。

- *TSQueue::~TSQueue*

在 destructor 裡，除了釋放分配的 buffer 空間，還要將 mutex lock 以及兩個 condition variable 銷毀，分別使用 `pthread_mutex_destroy()` 以及 `pthread_cond_destroy()`。

```
template <class T>
TSQueue<T>::~TSQueue() {
    // TODO: implements TSQueue destructor

    // 釋放 buffer 空間
    delete [] buffer;
    // 刪除 mutex 和 condition variable
    pthread_mutex_destroy(&mutex);
    pthread_cond_destroy(&cond_enqueue);
    pthread_cond_destroy(&cond_dequeue);
}
```

- *TSQueue::enqueue*

此函數負責將 item 加入 TSQueue。

```
template <class T>
void TSQueue<T>::enqueue(T item) {
    // TODO: enqueues an element to the end of the queue

    // 進入 critical section
    pthread_mutex_lock(&mutex);
    // buffer 滿了，不能再寫東西
    while(size == buffer_size){
        pthread_cond_wait(&cond_enqueue, &mutex);
    }
    // size < buffer_size
    buffer[tail] = item;
    tail = (tail + 1) % buffer_size;
    size++;
    // 此時 buffer 內有寫入東西了，有東西可以 dequeue 了
    pthread_cond_signal(&cond_dequeue);
    // 離開 critical section
    pthread_mutex_unlock(&mutex);
}
```

因為此函數會更動到 buffer 空間以及 size 兩個共享變數（寫入方 enqueue 以及取用方 dequeue 共享），所以需要將「寫入」這個動作放在 critical section 裡。

當有 thread 要寫入東西到 buffer 時，需要先取得 lock 才能進行寫入動作。這樣可確保當有 thread 在寫入 buffer 時，其他 thread 無法更動 buffer，達到 mutual exclusion，一次只有一個執行緒能夠修改或取用 buffer，以及修改 size 變數，避免同步問題。

首先利用 `pthread_mutex_lock()` 來嘗試取得 lock，成功後才嘗試進行寫入。寫入前還要檢查目前 buffer 是否已滿，如果是滿的，則無法將東西寫入 buffer。

我們利用 while 迴圈來檢查狀態，如果此時 buffer 的 size（目前 buffer 內 element 的數量）和 buffer\_size (maximum buffer size) 相同，則呼叫 `pthread_cond_wait()`，將此 thread 放入 cond\_enqueue 這個 condition variable 的 waiting queue，並讓此 thread 進入等待狀態，同時還會將其持有的 mutex lock 釋放，避免發生 deadlock。

利用 while 迴圈是為了達到不斷檢查的目的，當今天此 thread 被喚醒了，但很不幸被其他 thread 先把 mutex lock 搶走了並進行寫入動作造成 buffer 又滿了，還是需要將此 thread 再次放到 waiting queue，put to sleep 等待再次被喚醒。（**注意：所有關於 condition variable 的操作都需要在持有 mutex lock 下執行**）

若目前 size 小於 buffer\_size，代表還有空間能將東西放入，就可以執行寫入動作。將 item 寫進 buffer 的尾端，並將 tail 更新 +1。注意 buffer 是 circular queue，所以要除以 buffer\_size 取餘數。接著將 size+1，更新 buffer 內元素的數量。

當寫入動作完成時，還要呼叫 `pthread_cond_signal(&cond_dequeue)`。因為現在 buffer 中已經有元素可以取用了，應該要將剛才想從 buffer 取值，但因為 buffer 為空而被 put to sleep 的 thread 喚醒。此函數會喚醒 `cond_dequeue` 裡的 waiting queue 中的一個 thread，若沒有元素在等待 (waiting queue 為空)，此函數不會發生任何事。（**注意：所有關於 condition variable 的操作都需要在持有 mutex lock 下執行**）

最後，離開 critical section，利用 `pthread_mutex_unlock()` 將 mutex lock 釋放。

- ***TSQueue::dequeue***

此函數可取得 buffer 內的 head element 並回傳。

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

    //進入 critcal section
    pthread_mutex_lock(&mutex);
    // buffer 空的，不能拿取東西
    while(size == 0){
        pthread_cond_wait(&cond_dequeue, &mutex);
    }
    // size > 0
    T item_to_take = buffer[head];
    head = (head + 1) % buffer_size;
    size--;
    // 此時buffer內有東西被拿走了，有空間可以enqueue了
    pthread_cond_signal(&cond_enqueue);
    // 離開critical section
    pthread_mutex_unlock(&mutex);
    // 回傳 item_to_take
    return item_to_take;
}
```

此函數實作邏輯與 `enqueue()` 相似。

因為此函數會更動到 buffer 空間以及 size 兩個共享變數（寫入方 enqueue 以及取用方 dequeue 共享），所以需要將「拿取」這個動作放在 critical section 裡。

當有人要拿取 buffer 內的元素時，需要先取得 lock 才能進行取得動作。這樣可確保當有 thread 在拿取 buffer 元素時，其他 thread 無法更動 buffer，達到 mutual exclusion，一次只有一個執行緒能夠修改或取用 buffer，以及修改 size 變數，避免同步問題。

首先利用 `pthread_mutex_lock()` 來嘗試取得 lock，成功後才嘗試進行拿取。拿取前還要先檢查目前 buffer 是否為空，如果是空的，代表 buffer 內沒有東西可以取用。

我們利用 while 迴圈來檢查狀態，如果此時 buffer 的 size（目前 buffer 內 element 的數量）為 0，則呼叫 `pthread_cond_wait()`，將此 thread 放入 cond\_dequeue 這個 condition variable 的 waiting queue，並讓此 thread 進入等待狀態，同時還會將其持有的 mutex lock 釋放，避免發生 deadlock。

利用 while 迴圈是為了達到不斷檢查的目的，當今天此 thread 被喚醒了，但很不幸被其他 thread 先把 mutex lock 搶走了並把 buffer 最後一個元素拿走，導致 size 又變回 0，還是需要將此 thread 再次放到 waiting queue，put to sleep 等待再次被喚醒。（**注意：所有關於 condition variable 的操作都需要在持有 mutex lock 下執行**）

若目前  $\text{size} > 0$ ，代表 buffer 內還有東西能夠拿取，就可以執行取用動作。將 buffer 內的 head element 存進 item\_to\_take 變數，並將 head 更新  $+1$ 。注意 buffer 是 circular queue，所以要除以 buffer\_size 取餘數。接著將 size-1，更新 buffer 內元素的數量。

當拿取動作完成時，還要呼叫 `pthread_cond_signal(&cond_enqueue)`。因為我們剛剛拿走一個東西了，所以現在有空間可以寫入東西了，應該要將剛才想寫入 buffer，但因為 buffer 滿了而被 put to sleep 的 thread 喚醒。此函數會喚醒 `cond_enqueue` 裡的 waiting queue 中的一個 thread，若沒有元素在等待（waiting queue 為空），此函數不會發生任何事。（注意：所有關於 condition variable 的操作都需要在持有 mutex lock 下執行）

最後，離開 critical section，利用 `pthread_mutex_unlock()` 將 mutex lock 釋放，並回傳剛剛拿取的 item\_to\_take 元素。

- *TSQueue::get\_size*

此函數只是回傳目前 buffer 內元素的數量，因為不會修改到任何 shared variable，所以不用被 critical section 保護，直接回傳 TSQueue 物件的 size 變數值即可。

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

## 2. Producer

- *Producer::Start*

此函數呼叫 `pthread_create()` 以建立一個 producer thread，並指定 thread 要執行的函數為 `Producer::process()`，然後傳入自己為 argument。感謝有 reader 當範例，讓我們可以套用相同邏輯。

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

這邊的 t 變數是 Thread class 中的一個 protected 成員，Writer 類別因繼承自 Thread，可以直接存取該變數。t 的型別為 `pthread_t`，表示執行緒的唯一識別碼（類似進程的 PID），用於執行緒的操作管理。

- *Producer::process*

Producer 的工作是不斷地去 Input Queue 拿資料，並透過 `Transformer::producer_transformer()` 轉換後，把帶著新 value 值的資料放到 Worker Queue 裡。

```

void* Producer::process(void* arg) {
    // TODO: implements the Producer's work
    Producer* producer = (Producer*)arg;
    // producer不斷地把Input Queue資料，放到 Worker Queue
    while(true)
    {
        // 如果 Input Queue裡有東西才做搬移的動作
        if(producer->input_queue->get_size() > 0)
        {
            Item* item = producer->input_queue->dequeue();
            // call Transformer::producer_transform，且只改變val
            item->val = producer->transformer->producer_transform
                (item->opcode, item->val);
            // 把transform後的item放到Worker Queue
            producer->worker_queue->enqueue(item);
        }
    }
    return nullptr;
}

```

用一個無限迴圈使 producer 持續進行搬移資料的動作。

首先，先透過 `TSQueue::get_size()` 來檢查此時 Input Queue 裡面的元素數量是否  $>0$ ，若是則代表 Input Queue 裡有元素可以讓 producer 搬移。

檢查後進行搬移動作，先呼叫 `TSQueue::dequeue()` 從 Input Queue 裡拿取一個資料，並呼叫 `Transformer::producer_transformer()` 來把資料的 value 做轉換得到新 value (詳細轉換過程定義在 `transformer.cpp`，可以觀察到要傳入兩個參數，item 的 opcode 和 item 的 value，透過 opcode switch case，獲取其對應的 TransformSpec 值，再透過 TransformSpec 裡的 a, b, m, iteration 去計算新的 value 值。iteration 代表總共要做幾次運算，a, b, m 則是用來計算 value 的)，並利用 `TSQueue::enqueue()` 把帶著新 value 的資料放到 Input Queue 裡。

### 3. Consumer

- *Consumer::Start*

此函數呼叫 `pthread_create()` 建立一個 consumer thread，並指定 thread 要執行的函數為 `Consumer::process()`，然後傳入自己為 argument。感謝有 reader 當範例，讓我們可以套用相同邏輯。

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

- *Consumer::cancel*

此函數來於取消 consumer thread，並覆寫了 Thread class 中的 `cancel()`。故當 consumer thread 呼叫 `cancel()` 時，實際執行的是此函數。覆寫 `cancel()` 的目的是為了控制 Consumer class 中的 `is_cancel` 變數，讓 `Consumer::process()` 可以確認 consumer thread 是否呼叫了 `cancel()`，即時退出迴圈。

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

首先將此 consumer 物件的 `is_cancel` 設為 `true`，讓 `process()` 知道。

接著呼叫 `pthread_cancel()`，對此 consumer thread 發送取消請求，並在取消點時執行取消操作。

- *Consumer::process*

Consumer 的工作是不斷地去 Worker Queue 拿資料，並透過 [Transformer::consumer\\_transformer\(\)](#) 轉換後，把帶著新 value 的資料放到 Output Queue 裡。邏輯和 [producer::process\(\)](#) 相似。

```
void* Consumer::process(void* arg) {
    Consumer* consumer = (Consumer*)arg;

    pthread_setcanceltype(PTHREAD_CANCEL_DEFERRED, nullptr); // 使用deferred的cancel方式

    while (!consumer->is_cancel) {
        pthread_setcancelstate(PTHREAD_CANCEL_DISABLE, nullptr); // disable cancel request

        // TODO: implements the Consumer's work
        // 如果 Worker Queue裡有東西才做搬移的動作
        if(consumer->worker_queue->get_size() > 0)
        {
            Item* item = consumer->worker_queue->dequeue();
            // call Transformer::consumer_transform ,只改變val
            item->val = consumer->transformer->consumer_transform(item->opcode, item->val);
            // 把transform後的item放到Output Queue
            consumer->output_queue->enqueue(item);
        }

        pthread_setcancelstate(PTHREAD_CANCEL_ENABLE, nullptr); // enable cancel request
    }

    delete consumer;

    return nullptr;
}
```

可以先發現助教已經幫我們寫好 cancel 的宣告類型和保護機制。我們只需實作 consumer 的工作內容。

[pthread\\_setcanceltype\(\)](#) 用以設定此 thread (consumer) 的取消類型。在POSIX 裡有兩種 cancel type，一種是 asynchronous（立即cancel），另一種是這邊所用的 deferred（到達取消點時，才進行cancel），而這也是系統默認的取消類型，因為讓 thread 運行到一個段落才被 cancel，比較安全。

接著進入 while 迴圈，若 `is_cancel` 為 `False`，代表此 thread 可以繼續進行 consumer 的工作。反之則代表此 thread 要被 cancel，不應繼續執行 consumer 的工作，要跳出迴圈，執行完 `delete consumer` 並 `return`，等待 cancel 操作完成。

而 consumer 的工作內容，也和 producer 相似。首先，透過呼叫 `TSQueue::get_size()` 來檢查此時 Worker Queue 裡的元素數量是否  $> 0$ ，若是則代表 Worker Queue 裡有元素可以讓 consumer 取用。

檢查後進行搬移動作，先呼叫 `TSQueue::dequeue()` 從 Worker Queue 裡拿取一個資料，並呼叫 `Transformer::producer_transformer()` 來把資料的 value 做轉換得到新 value（詳細轉換過程定義在 `transformer.cpp`，可以觀察到要傳入兩個參數，item 的 opcode 和 item 的 value，透過 opcode switch case，獲取其對應的 TransformSpec 值，再透過 TransformSpec 裡的 a, b, m, iteration 去計算新的value值。iteration 代表總共要做幾次運算，a, b, m 則是用來計算 value 的）。並利用 `TSQueue::enqueue()` 把帶著新 value 的資料放到 Output Queue 裡。

這裡可以發現，在 consumer 工作前後，會呼叫 `pthread_setcancelstate()` 確認此 thread 是否可以處理取消請求。

`PTHREAD_CANCEL_DISABLE` 代表會先擋置 cancel 請求，繼續運行。而 `PTHREAD_CANCEL_ENABLE` 代表允許執行對此 thread 的 cancel 請求。

因此，consumer 在工作階段時都不會被 consumer controller cancel 掉，目的是確保資料 item 在被 dequeue, transform, enqueue 的過程中不會被打斷，造成資料搬移錯誤。cancel 的請求會等到 cancel state 重啟後，才被執行，結束此 consumer thread。

## 4. ConsumerController:

- 在 class 裡增加 `max_worker_queue_size` 變數

```
// 加上 Worker Queue 的 MAX size，即定義在main.cpp裡的WORKER_QUEUE_SIZE  
int max_worker_queue_size;
```

增加此變數，是因為需要知道 Worker Queue 最大的 size，才能算出此時 Worker Queue 的 loading (`current_size / max_worker_queue_size`)。而此變數就是定義在 main.cpp 的 WORKER\_QUEUE\_SIZE。

```
ConsumerController::ConsumerController(  
    TSQueue<Item*>* worker_queue,  
    TSQueue<Item*>* writer_queue,  
    Transformer* transformer,  
    int check_period,  
    int low_threshold,  
    int high_threshold,  
    int max_worker_queue_size  
) : worker_queue(worker_queue),  
    writer_queue(writer_queue),  
    transformer(transformer),  
    check_period(check_period),  
    low_threshold(low_threshold),  
    high_threshold(high_threshold),  
    max_worker_queue_size(max_worker_queue_size) {  
}
```

在 constructor 裡也要加上新的傳入參數：`max_worker_queue_size`。

- *ConsumerController::Start*

此函數呼叫 `pthread_create()` 建立一個 consumer thread，並指定 thread 要執行的函數為 `Consumer::process()`，然後傳入自己為 argument。感謝有 reader 當範例，讓我們可以套用相同邏輯。

此函數呼叫 `pthread_create()` 建立一個 consumer controller thread，並指定 thread 要執行的函數為 `ConsumerController::process()`，然後傳入自己為 argument。感謝有 reader 當範例，讓我們可以套用相同邏輯。

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

- *ConsumerController::Process*

ConsumerController 的工作是在 runtime 時，定期檢查 Worker Queue 的狀態，並動態調整 consumer 的數量。

```
void* ConsumerController::process(void* arg) {
    // TODO: implements the ConsumerController's work
    ConsumerController* controller = (ConsumerController*)arg;

    // 計算 high_threshold_ratio 和 low_threshold_ratio
    double high_threshold_ratio = (double)controller->high_threshold / 100;
    double low_threshold_ratio = (double)controller->low_threshold / 100;
```

首先把 arg (即ConsumerController物件本身) 存到 controller 指標。

接著利用 ConsumerController 裡的 high\_threshold 和 low\_threshold 變數計算 high threshold 和 low threshold 的比率。

```
//定期檢查 Worker Queue狀況
while(true)
{
    // 計算現在 Worker Queue的 element ratio
    double current_size = (double)controller->worker_queue->get_size() / (double)
    controller->max_worker_queue_size;
```

計算完臨界佔比後，進入 while 迴圈進行定期檢查的動作。先計算出目前 Worker Queue 的佔比，才能比較是否高於或是低於臨界值。

利用 `TSQueue::get_size()` 獲取 Worker Queue 此時的 element 數量，除以 `max_worker_queue_size` 得到目前 Worker Queue 的 loading rate。

```
if(current_size > high_threshold_ratio)
{
    // create a new consumer thread
    Consumer* new_consumer = new Consumer(controller->worker_queue, controller->writer_queue,
    controller->transformer);
    // starts new consumer thread
    new_consumer->start();
    // 把 new consumer 加入 consumer list
    controller->consumers.push_back(new_consumer);
    // output message
    std::cout << "Scaling up consumers from " << controller->consumers.size() - 1 << " to " <<
    controller->consumers.size() << '\n';
}
```

計算完目前 Worker Queue 的 loading rate 後，利用此比率進行比較。

若此比率高於 `high_threshold_ratio`，則創建一個新的 consumer thread，並呼叫 `Consumer::start()` 啟動 consumer 工作。

此外，我們還需要維護 running consumer list，使用 vector 的 `push_back()` 將新 consumer 加入 list 的尾端。

最後根據 Spec 規定，在 Scaling up 時需要印出訊息，這裡將 consumer 的前後數量印出，數量可由 vector 的 `size()` 得到。

```

// 要確保至少有一個consumer直到程式結束
else if((current_size < low_threshold_ratio) && (controller->consumers.size() > 1))
{
    // cancel the newest consumer thread
    controller->consumers.back()->cancel();
    // 呼叫 pthread_join()來等待consumer執行結束，確保其工作都做完了
    controller->consumers.back()->join();
    // 把 consumer 從 consumer list移除
    controller->consumers.pop_back();
    // output message
    std::cout << "Scaling down consumers from " << controller->consumers.size() + 1 << " to " <<
    controller->consumers.size() << '\n';
}

```

若目前 Worker Queue 的 loading rate 低於 low\_threshold\_ratio，則要 cancel 掉最新的 consumer，即 consumer list 的最後一個元素。

這邊要注意，spec 規定至少要有一個 consumer thread 直到 program 結束，因此還需加上一個條件，檢查此時 running consumer list 的元素數量是否  $> 1$ ，是的話才可以進行 cancel，否則若沒有 consumer 程式可能會卡住。

開始進行 cancel consumer，newest consumer 可透過 vector 的 `back()` 取得。首先，呼叫 newest consumer 的 `cancel()`，cancel 動作會到取消點時進行釋放。隨後呼叫 `join()` 以等待 consumer 做完剩下的工作，即 `process()` 裡 while 迴圈外面的 delete consumer，釋放記憶體空間。

這裡一樣需要維護 running consumer list，使用 vector 的 `pop_back()` 移除此 canceled consumer。

根據 Spec 規定，在 Scaling down 時也需要印出訊息，這裡一樣將 consumer 的前後數量印出，數量可由 vector 的 `size()` 得到。

```

    // 利用usleep()來達到固定的period檢查 Worker Queue
    //因單位是microsecond，所以使用usleep()，此函數單位已經是微秒了
    usleep(controller->check_period);
}

```

最後，為實現定期檢查 Worker Queue，並滿足 Spec 規定的單位 microsecond，我使用 `usleep()` 來達成。此函數的參數單位即是微秒，因此只需傳入在 `main.cpp` 中定義的 `check_period` 即可。透過 `usleep()`，consumer controller 能夠在休眠指定的時間後，定期喚醒並檢查 Worker Queue 的狀態。

完整程式如下：

```

void* ConsumerController::process(void* arg) {
    // TODO: implements the ConsumerController's work
    ConsumerController* controller = (ConsumerController*)arg;

    // 計算 high_threshold_ratio 和 low_threshold_ratio
    double high_threshold_ratio = (double)controller->high_threshold / 100;
    double low_threshold_ratio = (double)controller->low_threshold / 100;

    //定期檢查 Worker Queue狀況
    while(true)
    {
        // 計算現在 Worker Queue的 element ratio
        double current_size = (double)controller->worker_queue->get_size() / (double)
            controller->max_worker_queue_size;

        if(current_size > high_threshold_ratio)
        {
            // create a new consumer thread
            Consumer* new_consumer = new Consumer(controller->worker_queue, controller->writer_queue,
                controller->transformer);
            // starts new consumer thread
            new_consumer->start();
            // 把 new consumer 加入 consumer list
            controller->consumers.push_back(new_consumer);
            // output message
            std::cout << "Scaling up consumers from " << controller->consumers.size() - 1 << " to " <<
            controller->consumers.size() << '\n';
        }
        // 要確保至少有一個consumer直到程式結束
        else if((current_size < low_threshold_ratio) && (controller->consumers.size() > 1))
        {
            // cancel the newest consumer thread
            controller->consumers.back()->cancel();
            // 呼叫 pthread_join()來等待consumer執行結束，確保其工作都做完了
            controller->consumers.back()->join();
            // 把 consumer 從 consumer list移除
            controller->consumers.pop_back();
            // output message
            std::cout << "Scaling down consumers from " << controller->consumers.size() + 1 << " to " <<
            controller->consumers.size() << '\n';
        }
        // 利用usleep()來達到固定的period檢查 Worker Queue
        //因單位是microsecond，所以使用usleep()，此函數單位已經是微秒了
        usleep(controller->check_period);
    }
}

```

## 5. Writer:

- *Writer::Start*

此函數呼叫 `pthread_create()` 建立一個 writer thread，並指定 thread 要做的函數是 `Writer::process()`，然後傳入自己為 argument。感謝有 reader 當範例，讓我們可以套用相同邏輯。

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

- *Writer::Process*

此函數可取出 Output Queue 裡的 item，並寫入 output file，重複執行直到資料完入完畢 (`expected_lines == 0`)，最後回傳 `nullptr`。感謝有 reader 當範例，讓我們可以套用類似邏輯。

```
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 while 迴圈裡，透過 `TSQueue::dequeue()`，從 Output Queue 取出資料，並寫入 output file（利用定義在 `item.hpp` 裡的 `<<` 運算子重載，將 `item` 物件裡的 key, value, opcode 照格式輸出到文件中）。

## 6. main.cpp:

- 定義會用到的參數

```
#define READER_QUEUE_SIZE 200
#define WORKER_QUEUE_SIZE 200
#define WRITER_QUEUE_SIZE 4000
#define CONSUMER_CONTROLLER_LOW_THRESHOLD_PERCENTAGE 20
#define CONSUMER_CONTROLLER_HIGH_THRESHOLD_PERCENTAGE 80
#define CONSUMER_CONTROLLER_CHECK_PERIOD 1000000

// 新增的定義，方便實驗使用
#define PRODUCER_COUNT 4
```

除了原本定義好的參數，我們新增了一個參數 PRODUCER\_COUNT (producer 的數量)，為了額外實驗使用。

- *main function* 實作

main function 會啟動資料搬移，即 input → transform → output 的過程。所以會根據上面 #define 的各個參數，初始化 Input Queue, Worker Queue, Output Queue，以及 reader, producer, consumer controller, writer 這四個執行緒，還有 transformer。最後，要等待 reader 和 writer 都完成它們的工作（reader 要把 input file 裡所有的 item 都讀完，writer 要把所有 item 都寫到 output file 裡），main program 才能結束。

```
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]);
```

首先，根據輸入的命令行，解析 expected\_line 的數量（即 reader 和 writer 要讀和寫的行數），以及 input file name（reader 要去讀取的檔案）和 output file name（writer要去寫入的檔案）。

例如，測資00的輸入命令是

```
./main 200 ./tests/00.in ./tests/00.out
```

代表 expected\_line 有 200 行，input file 是 tests 資料夾裡的 00.in 檔案，output file 是 tests 資料夾裡的 00.out 檔案。

```
// TODO: implements main function
//先宣告會用到的 Queue 和 transformer
TSQueue<Item*> input_queue(READER_QUEUE_SIZE);
TSQueue<Item*> worker_queue(WORKER_QUEUE_SIZE);
TSQueue<Item*> output_queue(WRITER_QUEUE_SIZE);
Transformer transformer;
```

解析完命令後，初始化會用到的 Queue 以及 thread。先初始化 Input Queue、Worker Queue 以及 Output Queue，傳入 constructor 的參數 (max\_buffer\_size)是使用前面事先定義好#define的參數。

這裡順便初始化 transformer。

然後是 threads。

```
// 建構 producer 物件並放到 producer list裡
std::vector<Producer> producer_list;
for(int i = 0; i < PRODUCER_COUNT; i++){
    Producer p(&input_queue, &worker_queue, &transformer);
    producer_list.push_back(p);
}
```

這裡建構 producer 物件，根據前面定義的參數 PRODUCER\_COUNT 建構等量的 producer。傳入 constructor 的參數分別是 Input Queue、Worker Queue 以及 transformer，這些都是 producer 在搬移資料時會存取的物件。

其工作是去 Input Queue 裡取出 item，經過 transformer 後，把轉換後的 item 放到 Worker Queue 裡。這裡，我宣告了一個 vector 來儲存 producer list，代表正在 running 的 producer thread 有哪些。

```
//建構 Reader、Writer、ConsumerController
Reader reader(n, input_file_name, &input_queue);
Writer writer(n, output_file_name, &output_queue);
ConsumerController consumerController(&worker_queue, &output_queue, &transformer,
CONSUMER_CONTROLLER_CHECK_PERIOD, CONSUMER_CONTROLLER_LOW_THRESHOLD_PERCENTAGE,
CONSUMER_CONTROLLER_HIGH_THRESHOLD_PERCENTAGE, WORKER_QUEUE_SIZE);
```

接著建構 reader、writer 以及 consumer controller。

首先是 reader，傳入 constructor 的參數是 n (expected\_line)、input file name 和 Input Queue，這些都是 reader 工作時會存取的物件和變數。其任務是去 input file 裡讀取所有 item，放入 Input Queue。

再來是 writer，傳入 constructor 的參數是 n (expected\_line)、output file name 和 Output Queue，這些都是 writer 工作時會存取的物件和變數。其任務是去 Output Queue 裡讀取所有 item，寫入output file。

最後是 consumer controller，傳入 constructor 的參數是 Worker Queue、Output Queue、transformer（這三個是在創建consumer時會用到的）、Check Period、Low Threshold、High Threshold 以及 Max Worker Queue Size。其任務是定期檢查 Worker Queue 的 loading rate，若超過臨界值 (high threshold) 就創建新的 consumer thread，反之，低於臨界值 (low threshold) 就刪除最新的 consumer thread。

```
// 開始thread工作
reader.start();
writer.start();
for(int i = 0; i < PRODUCER_COUNT; i++){
    producer_list[i].start();
}
consumerController.start();
```

待所有結構以及執行緒物件都建構完後，分別呼叫其對應的 `start()`，讓這些執行緒開始工作。

```
// 等待 reader 和 writer thread 執行完畢才結束 main program
reader.join();
writer.join();

return 0;
}
```

最後，因為 main program 要等待 reader 和 writer 的工作完成才能結束，因此使用 `join()` 來等待它們把自己 thread 內的工作都做完。

完整 code 如下：

```
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
    // 先宣告會用到的 Queue 和 transformer
    TSQueue<Item*> input_queue(READER_QUEUE_SIZE);
    TSQueue<Item*> worker_queue(WORKER_QUEUE_SIZE);
    TSQueue<Item*> output_queue(WRITER_QUEUE_SIZE);
    Transformer transformer;

    // 建構 producer 物件並放到 producer list 裡
    std::vector<Producer> producer_list;
    for(int i = 0; i < PRODUCER_COUNT; i++){
        Producer p(&input_queue, &worker_queue, &transformer);
        producer_list.push_back(p);
    }

    // 建構 Reader、Writer、ConsumerController
    Reader reader(n, input_file_name, &input_queue);
    Writer writer(n, output_file_name, &output_queue);
    ConsumerController consumerController(&worker_queue, &output_queue, &transformer,
                                         CONSUMER_CONTROLLER_CHECK_PERIOD, CONSUMER_CONTROLLER_LOW_THRESHOLD_PERCENTAGE,
                                         CONSUMER_CONTROLLER_HIGH_THRESHOLD_PERCENTAGE, WORKER_QUEUE_SIZE);

    // 開始 thread 工作
    reader.start();
    writer.start();
    for(int i = 0; i < PRODUCER_COUNT; i++){
        producer_list[i].start();
    }
    consumerController.start();

    // 等待 reader 和 writer thread 執行完畢才結束 main program
    reader.join();
    writer.join();

    return 0;
}
```

### 3. Experiment

為了測試方便，我寫了一個腳本幫我把從生成 transformer 到驗證正確性的步驟全部做完，運行時間的量測方式使用 linux 的 time 指令。

```
class ConsumerController : public Thread {  
public:  
    // constructor  
    ConsumerController(  
        TSQueue<Item*>* worker_queue,  
        TSQueue<Item*>* writer_queue,  
        Transformer* transformer,  
        int check_period,  
        int low_threshold,  
        int high_threshold,  
        int max_worker_queue_size  
    );  
  
    // destructor  
    ~ConsumerController();  
  
    virtual void start();  
  
    int scale_up_count = 0;  
    int scale_down_count = 0;
```

為了方便觀察 scale up / down 的次數，我在 ConsumerController class 中加入這兩個變數，並在每次發生 scale up / down 時更新它們。

```
std::cout << "Scale up count: " << consumer_controller.scale_up_count << std::endl;  
std::cout << "Scale down count: " << consumer_controller.scale_down_count << std::endl;
```

在 main function 結束前，將統計完的數字輸出。

```
Scaling up consumers from 0 to 1
Scaling up consumers from 1 to 2
Scaling down consumers from 2 to 1
Scale up count: 2
Scale down count: 1

real    0m7.305s
user    0m16.690s
sys     0m0.003s
```

這是 test00 在原始參數下的測試結果。我們的實驗結果都會以 test00 做測試，並以此數據為基準做比較。

```
Scale up count: 28
Scale down count: 18

real    0m59.733s
user    7m10.004s
sys     0m0.143s
```

這是 test01 在原始參數下的測試結果（scale up / down 次數太多故沒有節錄）。若等等的實驗出現比較難以分析的結果，會再用 test01 做進一步的測試。

## 1. Different values of CONSUMER\_CONTROLLER\_CHECK\_PERIOD

- 提高 check period ( $1000000 \rightarrow 10000000$ )

```
Scaling up consumers from 0 to 1
Scale up count: 1
Scale down count: 0

real    0m18.279s
user    0m16.534s
sys     0m0.012s
```

- 降低 check period ( $1000000 \rightarrow 100000$ )

```
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 down consumers from 6 to 5
Scaling down consumers from 5 to 4
Scaling down consumers from 4 to 3
Scaling down consumers from 3 to 2
Scale up count: 6
Scale down count: 4

real    0m3.460s
user    0m16.565s
sys     0m0.011s
```

可以觀察到，check period 越高，consumer controller 對於 worker queue 滿載率的反應越不靈敏（scale up / down 次數減少），導致其調度 consumer 數量的能力差，故執行速度變慢。反之則可以快速調配 consumers 數量（scale up / down 次數增加），故執行速度變快。

## 2. Different values of

CONSUMER\_CONTROLLER\_LOW\_THRESHOLD\_PERCENTAGE &  
CONSUMER\_CONTROLLER\_HIGH\_THRESHOLD\_PERCENTAGE

- 降低 **high threshold (low = 20, high = 60)**

```
Scaling up consumers from 0 to 1
Scaling up consumers from 1 to 2
Scaling up consumers from 2 to 3
Scale up count: 3
Scale down count: 0

real    0m5.777s
user    0m16.537s
sys     0m0.012s
```

由於 scale up 的條件更容易被觸發（scale up 次數增加），故執行時間變快。

- 提高 **high threshold (low = 20, high = 90)**

```
Scaling up consumers from 0 to 1
Scale up count: 1
Scale down count: 0

real    0m10.303s
user    0m16.596s
sys     0m0.011s
```

由於 scale up 的條件更難被觸發（scale up 次數減少），故執行時間變慢。

- 降低 **low threshold (low = 10, high = 80)**

```
Scaling up consumers from 0 to 1
Scaling up consumers from 1 to 2
Scale up count: 2
Scale down count: 0

real    0m6.654s
user    0m16.581s
sys     0m0.016s
```

由於 scale down 的條件更難被觸發（scale down次數減少），故執行時間變快。

- 提高 low threshold (low = 40, high = 80)

```
Scaling up consumers from 0 to 1
Scaling up consumers from 1 to 2
Scaling down consumers from 2 to 1
Scale up count: 2
Scale down count: 1

real    0m8.263s
user    0m16.578s
sys     0m0.009s
```

由於 scale down 的條件更容易被觸發，故執行時間變慢。

但經實驗發現， test00 的行數可能不足以觀察到提高 low threshold 對於 scale up / down 次數的影響（但執行時間仍然有差距），因此這裡再使用 test01 做測試。

- 提高 low threshold (low = 70, high = 80) (使用 test01 測試)

```
Scale up count: 28
Scale down count: 20

real    1m0.951s
user    7m20.288s
sys     0m0.151s
```

這裡就可以看出 scale down 次數確實增加了，且執行時間也確實變慢。

### 3. Different values of WORKER\_QUEUE\_SIZE.

- 降低 worker queue size 至 100

```
Scaling up consumers from 0 to 1
Scaling up consumers from 1 to 2
Scaling up consumers from 2 to 3

real    0m4.794s
user    0m16.655s
sys     0m0.010s
```

降低 worker queue size 後，其滿載率在所含元素數量相同時變高了，更容易超過 high threshold，所以 consumer 更容易 scale up，故速度變快。

- 提高 worker queue size 至 400

```
Step 4: Running the main program...
```

提高 worker queue size 後，其滿載率在所含元素數量相同時變低了，更不容易超過 high threshold，所以 consumer 更不容易 scale up，速度會變慢。甚至在提高到 400 這種極端的情況下，由於 test00 只有 200 行，滿載率最高只會到 50%，完全達不到新增 consumer 的閾值，所以程式完全沒在動。

- 提高 worker queue size 至 400 (使用 test01 測試)

```
Scale up count: 24
Scale down count: 16

real    1m1.864s
user    7m11.820s
sys     0m0.140s
```

用 test01 再做一次測試，可以發現執行速度確實變慢。

#### 4. What happens if WRITER\_QUEUE\_SIZE is very small?

- 降低 writer queue size 至 1

```
Scaling up consumers from 0 to 1
Scaling up consumers from 1 to 2
Scaling down consumers from 2 to 1
Scale up count: 2
Scale down count: 1

real    0m7.254s
user    0m16.532s
sys     0m0.017s
```

可發現與基準值下的表現幾乎一樣，代表 writer queue size 應該不影響結果。

為求精確，再使用 test01 測試。

- 降低 writer queue size 至 1 (使用 test01 測試)

```
Scale up count: 28
Scale down count: 18

real    0m59.729s
user    7m10.295s
sys     0m0.137s
```

確實不影響結果。

## 5. What happens if READER\_QUEUE\_SIZE is very small?

- 降低 reader queue size 至 1

```
Scaling up consumers from 0 to 1
Scaling up consumers from 1 to 2
Scaling down consumers from 2 to 1
Scale up count: 2
Scale down count: 1

real    0m7.262s
user    0m16.566s
sys     0m0.015s
```

可發現與基準值下的表現幾乎一樣，代表 reader queue size 應該也不影響結果。

為求精確，再使用 test01 測試。

- 降低 reader queue size 至 1 (使用 test01 測試)

```
Scale up count: 28
Scale down count: 18

real    0m59.730s
user    7m10.199s
sys     0m0.151s
```

確實不影響結果。

## 6. Different value of PRODUCER\_COUNT. (額外實驗)

- 提高 producer count 至 20

```
Scaling up consumers from 0 to 1
Scaling up consumers from 1 to 2
Scaling down consumers from 2 to 1
Scale up count: 2
Scale down count: 1

real    0m6.261s
user    0m16.589s
sys     0m0.022s
```

- 提高 producer count 至 20 (使用 test01 測試)

```
Scale up count: 18
Scale down count: 1

real    0m23.492s
user    7m13.545s
sys     0m0.167s
```

- 降低 producer count 至 1

```
Scaling up consumers from 0 to 1
Scaling up consumers from 1 to 2
Scaling down consumers from 2 to 1
Scale up count: 2
Scale down count: 1

real    0m12.249s
user    0m16.561s
sys     0m0.008s
```

- 降低 producer count 至 1 (使用 test01 測試)

```
Scale up count: 10
Scale down count: 9

real    3m27.744s
user    7m10.067s
sys     0m0.162s
```

觀察結果，producer count 越大，執行速度越快，反之速度越慢。

### 3. Reflection

林櫻松：本次我主要負責實作的部分，在實作時整體上沒有太大的困難，多虧了有 reader 已經實作完的架構可以參考，真正有遇到困難應該是在 consumer controller 的 `process()` 實作裡面，當要 cancel consumer 時，consumer 會在哪個取消點被 cancel 困擾我許久，深怕 consumer 還沒做完 while 迴圈外的事情就被停止了，導致記憶體漏洞，後來查詢資料以及詢問助教，才發現只要在 cancel 後面加一個 `join()` 當作取消點，就能確保 consumer 把工作都做完才被 cancel。而在 consumer 的 `process()` 那裡也學到了兩個 pthread 的函數，`pthread_setcanceltype` 以及 `pthread_setcancelstate`，讓我對 cancel thread 的細節設定更加瞭解。雖然此作業只是基本的同步化實作問題，但是實作完成後對 pthread 的基本操作有獲得超出課堂上所吸收到的知識，也瞭解到平行化程式要注意的細節很多。

賴允中：本次我主要負責實驗的部分。要做出有參考意義的實驗必須要經過大量的測試，為了讓實驗結果不要過於偏頗，我把幾乎所有能調整的方向都試過了。得到大概的結論後，為求精確又對某些項目再做了更嚴謹的測試（更大的測資）。這次作業讓我學到很多平行程式的概念，原本單純上課聽一聽其實沒什麼想法（尤其是 `join()` 的部分），自己做一遍才知道原來要考慮的東西很多，也比較有概念了。