

# Pthread Report - 47

## 團隊分工表

成員	Implementation	Experiment	Report
111065542 楊智堯	v	v	v
111065531 郭芳妤	v	v	v

## Part 1 - Implementation

感謝清楚的 spec 以及有 reader.hpp 作為範例，讓實作過程都很順暢！

▼ **ts\_queue.hpp**：首先是最重要的結構 **TSQueue**，整個 process 涉及三個需要以 **TSQueue** 建構 queue

Queue	produced by	consumed by
input_queue	reader (1)	producer (4)
worker_queue	producer (4)	consumer (flexible)
output_queue	consumer (flexible)	writer (1)

首先觀察 **TSQueue** 的結構，可以看到它是由一個 buffer 組成，並維護四個參數：

- buffer\_size (buffer 內最多可以有幾個 element)
- size (此刻的 element 個數)
- head (queue 的第一個 element 此刻的 index)
- tail (queue 的最後一個 element 此刻的 index)

這個部分需要實作的是 TSQueue 的 constructor、destructor，以及最重要的 enqueue、dequeue，另外還有取用 #element 使用的 get\_size()，共五個 function：

- **constructor/destructor**：constructor 主要需要初始化的部分是由傳入的 buffer\_size 指定 buffer 的大小，同時根據上表所見，每個 buffer 都會涉及不同 thread 的 access（寫入或取用其中的內容），因此需要 mutex 來作為 critical section 的保護；並且因為 buffer size 並不能無限擴張，因此還需要使用 condition variable 進行 buffer size 的維護（hint by the private variables of **TSQueue**），所以在這邊也要先進行 mutex 和 condition variable 的初始化。而 destructor 就很單純的移除我們一開始創建的 buffer 即可。

```

template <class T>
TSQueue<T>::TSQueue(int buffer_size) : buffer_size(buffer_size) {
    // TODO: implements TSQueue constructor
    size = head = tail = 0;
    buffer = new T[buffer_size];
    pthread_mutex_init(&mutex, nullptr);
    pthread_cond_init(&cond_enqueue, nullptr);
    pthread_cond_init(&cond_dequeue, nullptr);
}

template <class T>
TSQueue<T>::~~TSQueue() {
    // TODO: implements TSQueue destructor
    delete[] buffer;
}

```

- **enqueue** : enqueue 表示在 buffer 內放入一個 element，由於這個 buffer 由寫入方和取用方共享，整個 buffer 內的 element 數量也會因此被改變，因此我們首先會需要把「放入」這個動作用 mutex 鎖住，保證在有 thread 要放入/取用東西時，沒有其他 thread 可以接觸到這個 buffer。同時，我們也要考慮到 buffer 並不能無限的被放入 element，因此在進入 critical section 後第一件事情是先檢查 buffer 是否已經滿員 (size == buffer\_size)。假如已經滿員，表示現在要卡住任何試圖往內寫入的 thread，這邊使用的就是 pthread\_cond\_wait()；假設此時 buffer 尚有空位，才允許在 buffer 的最後放入新的 element。在此，由於 buffer 是一個 circular array 的形式，因此用 (tail+1)%buffer\_size 表示新的 tail index，同時，也更新 size。待這些 TSQueue 內的參數都 update 完畢，我們先使用

pthread\_cond\_signal(&cond\_dequeue) 來喚醒正在等待取用 buffer 內容的 thread (if any)，最後才打開 mutex，允許其他 thread 來 access 這個 buffer。

```

template <class T>
void TSQueue<T>::enqueue(T item) {
    // TODO: enqueues an element to the end of the queue
    pthread_mutex_lock(&mutex);
    while(size == buffer_size){
        pthread_cond_wait(&cond_enqueue, &mutex); //
    }
}

```

```

queue 已滿不能再新增東西
    }
    buffer[tail] = item;
    tail = (tail+1)%buffer_size; // circular index
    size++;
    pthread_cond_signal(&cond_dequeue); // 現在有東西可以
    以去取用！
    pthread_mutex_unlock(&mutex);
}

```

- dequeue：和 enqueue 是類似的邏輯，差別只是要移除並回傳 buffer 的第一個 element。因此在 access buffer 前先進行上鎖（mutex lock），接著先用 while 來維護當 buffer 為空（size = 0），即使進到了這個 critical section 也必須 hold 在迴圈中，直到被任何 enqueue 的 thread signal 為止。接著是對第一個 element 的取用、維護 head index 以及 size，最後回傳第一個 element。

```

template <class T>
T TSQueue<T>::dequeue() {
    // TODO: 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--;
    pthread_cond_signal(&cond_enqueue);
    pthread_mutex_unlock(&mutex);
    return item;
}

```

- get\_size()：直接回傳 size 即可

```

template <class T>
int TSQueue<T>::get_size() {
    return size;
}

```

▼ `producer.hpp`：仿照 reader 的邏輯撰寫 start 和 process function，這是表示每一個 producer thread 需要執行的內容

- start：開始一個 thread 作為 producer（這邊的 &t 是在 `thread.hpp` 中已經定義好的 protected variable: thread）

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

- process：因為 producer 的工作是從 input queue 中取出一個 item，並經過 transformer 轉換後，再把轉換結果放入 worker queue，所以可以使用上面定義的 TSQueue 進行此操作：

```
void* Producer::process(void* arg) {  
    // TODO: implements the Producer's work  
    while(true){  
        Producer* producer = (Producer*)arg;  
        Item* item = producer->input_queue->dequeue()  
();  
        /* call transformer */  
        item->val = producer->transformer->producer_transform(item->opcode, item->val);  
        producer->worker_queue->enqueue(item);  
    }  
  
    return NULL;  
}
```

呼叫 transformer 的方法可以參考 `transformer.cpp`：可以觀察到 Producer\_transformer 的 input 參數為：item 的 opcode、item 的 value，因此我們只要從 input queue 中取出 item 後再將此二值當作參數輸入 function，即可用傳入的 opcode 和 value 來決定 item 的新 value。

▼ `consumer.hpp`：即所謂的 worker，必須從 worker queue 中取出被 producer 傳入內容，再用自己的 transformer function 進行再度轉換以後寫入 output queue。

- start：開始一個 thread 作為 consumer，邏輯同 reader, producer

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

- process：在這邊助教的 code 已經幫我們寫好 **cancel 的保護機制**，所以我們其實只要很單純的 implement 三個步驟：(1) 從 worker queue 取出 item、(2) 針對取出的 item 用 consumer\_transformer 進行轉換、(3) 將轉換後的 item 再放入 writer queue 即可。

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

    pthread_setcanceltype(PTHREAD_CANCEL_DEFERRED, nullptr);

    while (!consumer->is_cancel) {
        pthread_setcancelstate(PTHREAD_CANCEL_DISABLE, nullptr);

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

        pthread_setcancelstate(PTHREAD_CANCEL_ENABLE, nullptr);
    }

    delete consumer;
    return nullptr;
}
```

- pthread\_cancel：

- `pthread_setcanceltype(PTHREAD_CANCEL_DEFERRED, nullptr)`：cancel type 有兩種，一種是 asynchronous（立即執行 cancel thread），另一種是我們使用的 deferred（運行到下一個 cancelation point 才執行 cancel thread）。
- `pthread_setcancelstate(PTHREAD_CANCEL_DISABLE, nullptr)`、`pthread_setcancelstate(PTHREAD_CANCEL_ENABLE, nullptr)`：表示在此階段都 thread 都不可以被 controller cancel 掉，因為我們要確保 item 在被 dequeue、轉換（transform）到最後 enqueue 的過程中，因為執行的 thread 突然的被 cancel 而讓 item 沒有被成功轉換或成功放到 output queue 的狀況。
- cancel：當 worker queue 的 buffer size 超過或低於我們設定的 threshold 時，會需要動態調整 consumer 的數量，consumer controller 需要透過這個 function 知道需要結束哪一個 consumer thread。

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

▼ `consumer_controller.hpp`：即用來動態調整 worker size 的 process，需要根據傳入的 threshold、period 定期進行 worker 數量調整

- 微調：由於我們需要 `max_worker_size` 才能算出高低 threshold 的值，因此在 controller 的 constructor 加上一個新的輸入參數：`max_worker_size`（即在 `main.cpp` 定義的 `#define WORKER_QUEUE_SIZE 200`）

```
// constructor
ConsumerController(
    TSQueue<Item*>* worker_queue,
    TSQueue<Item*>* writer_queue,
    Transformer* transformer,
    int check_period,
    int low_threshold,
    int high_threshold,
    int max_worker_size
);
```

- start：開始一個 thread 作為 consumer，邏輯同 reader, producer, consumer

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

- process：由於 consumer (worker) 數量需要經過動態調整，換句話說，我們需要創建一個額外的 process 依照設定的 check period 進行監控和調整。controller 需要做的事情如下：
  1. 先取得此時的 worker size，並計算出目前 worker queue 的佔滿率 (size/max\_worker\_size)
  2. 利用 controller 在建構時定義好的 high/low threshold 取得 20%、80% 的臨界值 (ratio)
  3. **情況一**：如果當前 worker queue 佔滿率 > high\_ratio (80%)，表示我們需要更多 consumer，所以在這邊新增一個 consumer，並啟動後丟入 controller 的 consumer vector 中；**情況二**：如果當前 worker queue 佔滿率 < low\_ratio (20%)，表示我們不需要這麼多 consumer，因此可以從 vector 當中取出最後一個進行 cancel (詳見 consumer 的 cancel)
  4. 使用 vector 的功能 size() 印出 scale up 和 scale down 所需要的 worker queue 內數量資訊
  5. 根據助教在留言區的提示，使用 `usleep(check_period)`，來設定週期性的檢查 period

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

    while(true){

        double s = (double)controller->worker_queue->get_size()/
            (double)controller->max_worker_size;
        double high_ratio = (double)(controller->h
```

```

    igh_threshold)/100;
    double low_ratio = (double)(controller->low
w_threshold)/100;

    if(s > high_ratio){
        /* create new worker:consumer */
        Consumer* new_consumer = new Consumer
(controller->worker_queue, controller->writer_queue,
controller->transformer);
        new_consumer->start();
        controller->consumers.push_back(new_consumer);

        std::cout << "Scaling up consumers from " << controller->consumers.size()-1 << " to " <<
controller->consumers.size() << "\n";
    }

    else if(s < low_ratio && controller->consumers.size() > 1){
        /* delete worker */
        controller->consumers.back()->cancel
(); // 不能馬上移除，要等他工作完，所以先宣告不可以再派工作
給這個 thread
        controller->consumers.pop_back();
        std::cout << "Scaling down consumers from " << controller->consumers.size()+1 << " to "
<< controller->consumers.size() << "\n";
    }
    usleep(controller->check_period);
}
}

```

#### ▼ [writer.hpp](#)

- start：開始一個 thread 作為 writer，邏輯同 reader, producer, consumer

```

void Writer::start() {
    // TODO: starts a Writer thread
}

```



```
pthread_create(&t, 0, Writer::process, (void*)this);
}
```

- process：由一個 writer 將 worker 轉換後的內容寫入最後的 .out file，邏輯與 reader 非常相似，只是改成寫入 ofs 中，直到取完所有內容後結束。

```
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;
}
```

#### ▼ main.cpp

- 定義我們想要的各項參數（原始設定，這裡有新增一個參數定義是 PRODUCER\_CNT，下方宣告 producer 會使用到）

```
#define PRODUCER_CNT 4

#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
```

- main function 啟動整個 input → 轉換 → output 的過程：
  1. 首先，根據給定的 size 參數創建 input、worker、output queue（使用 TSQueue 結構）
  2. 建構 transformer

3. 建構化四個 producer (by PRODUCER\_CNT) , 並讓這些 thread 開始工作 (start)
4. 建構 reader、writer、consumer controller, 並讓這些 thread 開始工作 (start)
5. 待所有工作結束後, 使用 join() 來結束 writer, reader thread
6. 最後記得把所有 new 出來的資源 delete 掉

```
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(READER_QUEUE_SIZE);
    TSQueue<Item*> worker_queue(WORKER_QUEUE_SIZE);
    TSQueue<Item*> output_queue(WRITER_QUEUE_SIZE);
    Transformer trans;

    std::vector<Producer> P_array;
    for(int pid = 0; pid < PRODUCER_CNT; pid++){
        Producer p(&input_queue, &worker_queue, &trans);
        P_array.push_back(p);
    }

    Reader r(n, input_file_name, &input_queue);
    Writer w(n, output_file_name, &output_queue);
    ConsumerController consumerController(&worker_queue, &output_queue, &trans,
        CONSUMER_CONTROLLER_CHECK_PERIOD, CONSUMER_CONTROLLER_LOW_THRESHOLD_PERCENTAGE,
        CONSUMER_CONTROLLER_HIGH_THRESHOLD_PERCENTAGE, WORKER_QUEUE_SIZE);

    r.start();
```

```

        w.start();
        for(int pid = 0; pid < PRODUCER_CNT; pid++){P_array[pid].start();}
        consumerController.start();

        r.join();
        w.join();
        return 0;
    }

```

## Part2 - Experiment



執行時間計算方式：

因為 pthread 的概念是想要以平行化的方式加速（transform 的部分），因此以下實驗也會觀察不同 setting 下的執行時間是否確實受到影響，因此在 code 中加入計時部分。

```

struct timespec begin, end;
double timediff;
clock_gettime(CLOCK_MONOTONIC, &begin);

/* main code ... */

clock_gettime(CLOCK_MONOTONIC, &end);
timediff = end.tv_sec - begin.tv_sec + (end.tv_nsec - begin.tv_nsec) / 1000000000.0;
printf("執行時間: %f秒\n", timediff);

```

### 1. Different values of `CONSUMER_CONTROLLER_CHECK_PERIOD`

以 test.00 當做測資，測試 500000、1000000、2000000、4000000 的差異，可以發現如果 check period 時間越長，進行 scaling up/down 的次數就會越少，同時，因為可能 scale up 的機會減少了，運行時間也會較長，以下為不同時間參數的最高 worker 數量以及總執行時間：

check period	max #threads	time (秒)
25,0000	4	4.26
50,0000	3	5.39
100,0000	2	7.23
200,0000	1	10.26
800,0000	1	16.26

- 舉例：CONSUMER\_CONTROLLER\_CHECK\_PERIOD = 500000，最多有 3 個 worker

```
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
```

## 2. Different values of CONSUMER\_CONTROLLER\_LOW\_THRESHOLD\_PERCENTAGE and

CONSUMER\_CONTROLLER\_HIGH\_THRESHOLD\_PERCENTAGE

- low threshold percentage：以 **check period = 50,0000、test00.in** 做測試 **(high fixed at 80%)**，可以發現調低 low threshold 時降低了 scale down 的可能性，因此較低的 low threshold 會較快完成

low threshold	scale up 次數	scale down 次數	time (秒)
5	3	0	5.25
10	3	1	5.39
20 (original)	3	1	5.38
40	3	2	6.20

- high threshold percentage：以 **check period = 50,0000、test00.in** 做測試 **(low fixed at 20%)**，當我們調低 high threshold，代表此時 consumer 被 scale up 的可能性增加，而調高 high threshold 代表減少 scale up 可能性，因此可以發現 high threshold 越高，所需要的執行時間越久

high threshold	scale up 次數	scale down 次數	time (秒)
50	4	1	4.43
60	4	1	4.42
70	3	1	4.88
80 (original)	3	1	5.38

high threshold	scale up 次數	scale down 次數	time (秒)
90	2	1	6.74

3. Different values of `WORKER_QUEUE_SIZE` : 固定 threshold 為 20, 80, check period 為 50,000 : 可以看到當我們調小 worker queue, 在 thread 處理速度相同的情況下, 很容易就因為達到 high threshold 的限制而進行 scale up, 進而減少執行時間; 反之, 越大的 worker queue 要觸發 scale up 的機會就非常小, 所以可以看到當 worker queue = 400 以上時, 基本上都只有原本的那個 consumer thread 在工作, 因此時間會更久

worker queue size	scale up 次數	scale down 次數	time (秒)
50	5	0	3.16
100	4	0	3.81
200 (original)	3	1	5.38
400	0	0	8.29
800	0	0	8.30

4. What happens if `WRITER_QUEUE_SIZE` is very small?

- 以 tests/00 (小資料) 結果來看, writer queue 的大小不似乎不影響執行時間以及後續的 scale up 和 scale down 的情形。

writer queue size	scale up 次數	scale down 次數	time (秒)
1	3	1	5.39
5	3	1	5.39
10	3	1	5.38
50	3	1	5.39
200	3	1	5.39
500	3	1	5.38
1000	3	1	5.39
2000	3	1	5.39
4000 (original)	3	1	5.38

- 改以 tests/01 測試 (大資料) 時, 可以發現當我們測試 writer queue size = 1, 執行時間為 70.94 秒, 比原本的 62.16 秒還要慢, 推測可能是因為只有一個 writer 進行 dequeue, 但可能有多個 consumer 進行 enqueue, 當 writer queue size 減少時, 這些想要 enqueue 的 consumer 可能因為空間已滿而必須進行 condition wait, 導致執行時間更長。

## 5. What happens if `READER_QUEUE_SIZE` is very small?

- 以 tests/00 結果來看，reader queue 的大小似乎不影響執行時間以及後續的 scale up 和 scale down 的情形，使用 tests/01 之結果也相同。猜測是因為本來 reader thread 就只有一個，而且幾乎是一把 item 放入 producer 就會把 item 取走，所以不會有因為把 queue size 調小而導致塞車而速度變慢的問題。

reader queue size	scale up 次數	scale down 次數	time (秒)
1	3	1	5.39
10	3	1	5.39
25	3	1	5.39
50	3	1	5.39
100	3	1	5.38
200 (original)	3	1	5.38

## Difficulties


這份作業在實作上沒有遇到特別大的困難，主要是感謝 reader.hpp 提供了整個架構的模板，關於怎麼 start 和 process 的設定方式都是參考最一開始的 reader.hpp。但在 allocate 和回收資源（比如說哪些需要 delete、哪些需要 join()）時遇到了比較多障礙，像是我一開始就是因為對 controller 做了 join() 導致程式一直無法正確結束，因此卡關了一陣子。整體而言這份作業讓我們體驗了 pthread 的運用與相關程式的撰寫，實際運用了 mutex 和 condition variable 後，對於同步化機制也有了更具體的了解。

## Reference

- <https://feng-qi.github.io/2017/05/08/Why-do-pthreads-condition-variable-functions-require-a-mutex/>
- thread 計時的寫法

### pthread中如何计时

使用pthread编写的多线程程序，若是用clock\_t结构体和clock()函数计时，多线程程序的运行时间会偏大，如123456#include <time.h>;clock\_t begin, end;begin = clock();// pthread多线程代码end = clock();printf(&quot;%f秒\n&quot;;, (double)(t1 - t0) / C

 <https://yiyang186.github.io/2016/04/06/pthread中如何计时/>