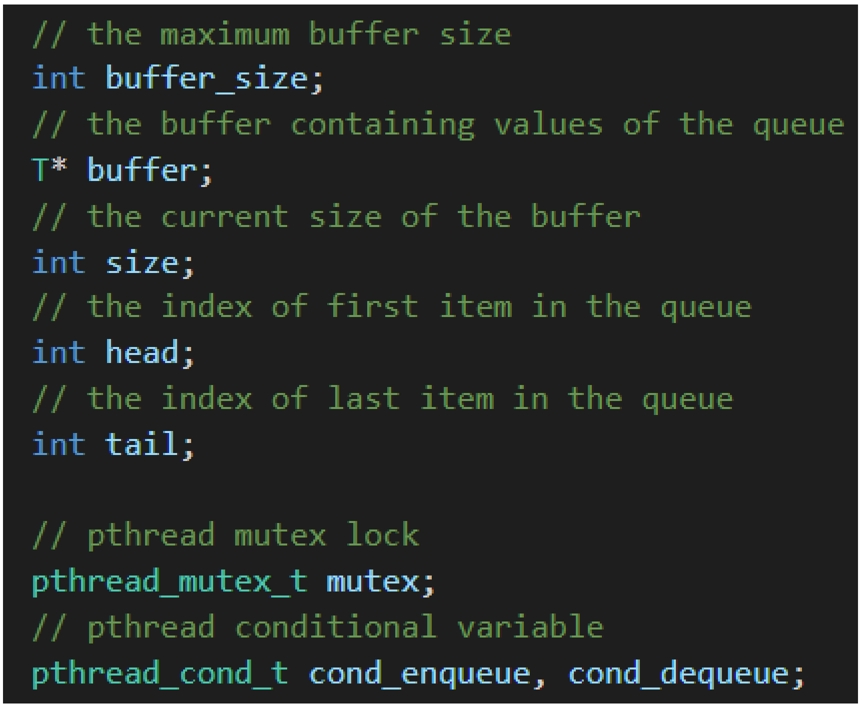
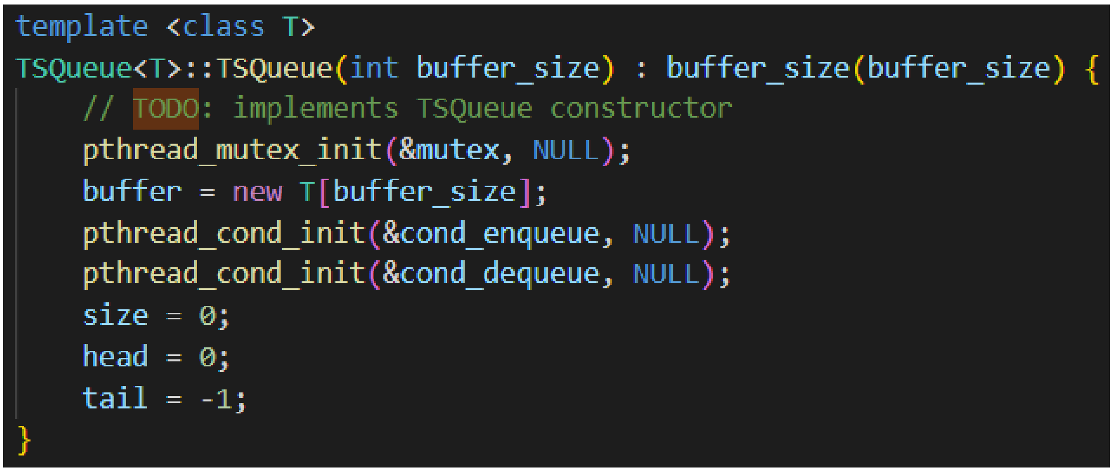
Implementation

首先根據 implementation 的第一點 TSQueue，我們找到了第一個 TODO 坐落在 TSQueue 的 constructor 裡。因為傳入 constructor 的參數只有 buffer_size，所以我們再依據 class 裡已經定義好的 private 變數，將未初始化的進行初始化。

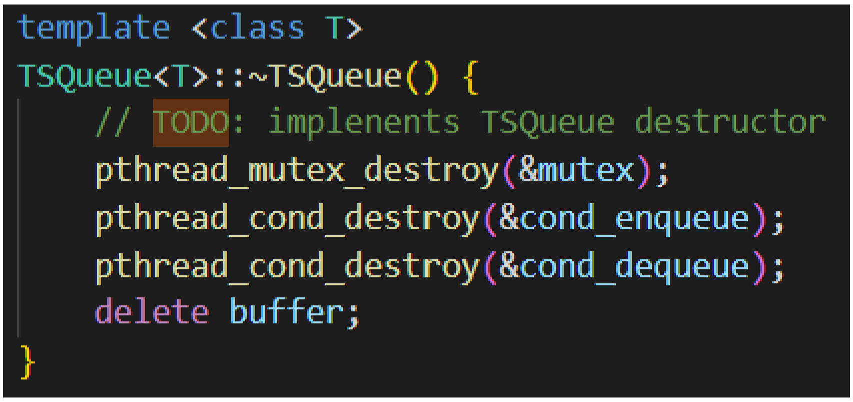


```
// the maximum buffer size
int buffer_size;
// the buffer containing values of the queue
T* buffer;
// the current size of the buffer
int size;
// the index of first item in the queue
int head;
// the index of last item in the queue
int tail;

// pthread mutex lock
pthread_mutex_t mutex;
// pthread conditional variable
pthread_cond_t cond_enqueue, cond_dequeue;
```



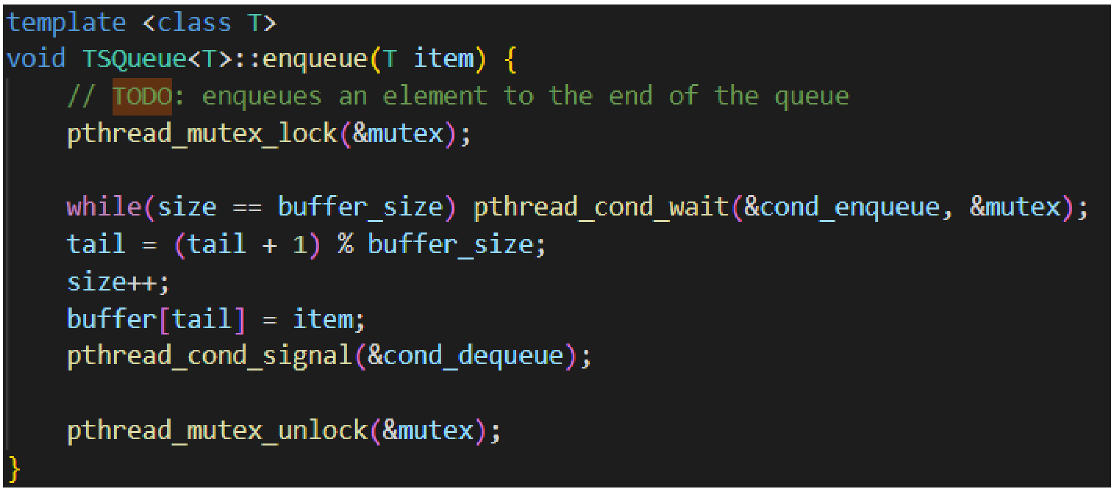
```
template <class T>
TSQueue<T>::TSQueue(int buffer_size) : buffer_size(buffer_size) {
    // TODO: implements TSQueue constructor
    pthread_mutex_init(&mutex, NULL);
    buffer = new T[buffer_size];
    pthread_cond_init(&cond_enqueue, NULL);
    pthread_cond_init(&cond_dequeue, NULL);
    size = 0;
    head = 0;
    tail = -1;
}
```

在此處值得注意的地方是，tail 我們初始化為-1，因為此時 queue 裡還沒有任何的 item 存在，所以以 index -1 代表目前無法存取 dequeue 位置，我們在操作 tail 這個 index 的時候會先將其+1 所以並不會有超出 boundary 的情形。而 mutex 和 condition variable（以下簡稱 CV）的初始化都是依照講義上的範例實作出來的。



```
template <class T>
TSQueue<T>::~TSQueue() {
    // TODO: implenents TSQueue destructor
    pthread_mutex_destroy(&mutex);
    pthread_cond_destroy(&cond_enqueue);
    pthread_cond_destroy(&cond_dequeue);
    delete buffer;
}
```

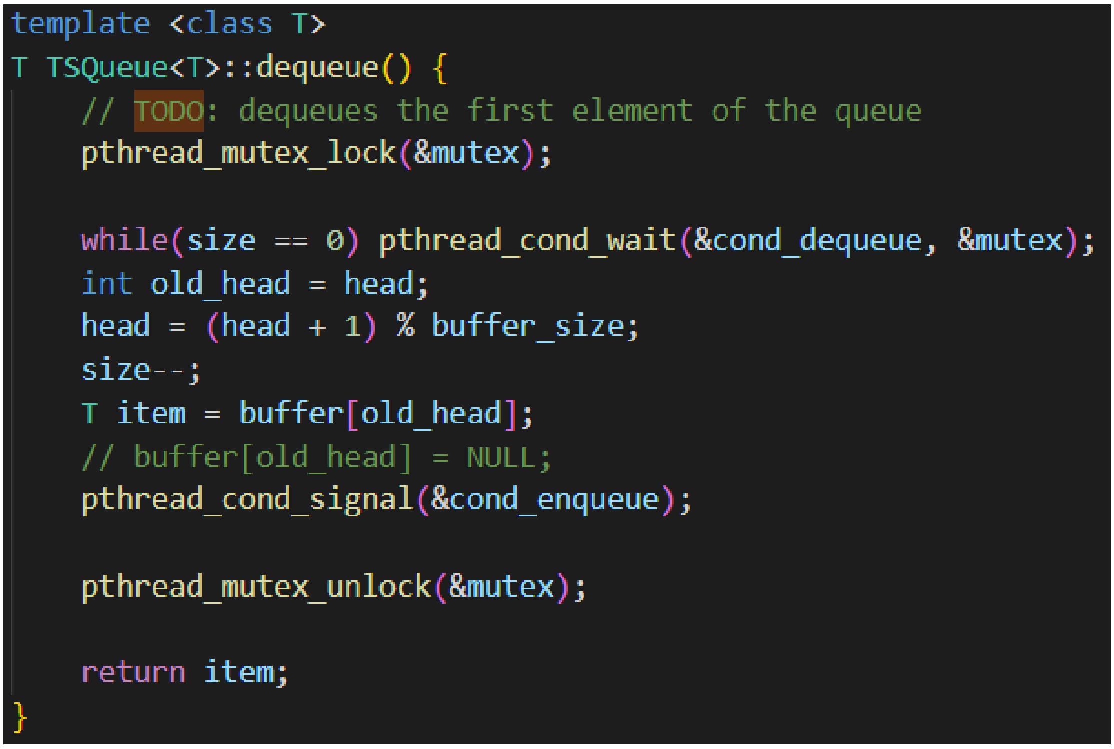
接下來便是 destructor 的實作，這一部分很簡單，只是將 mutex 和 CV 呼叫各自的 destroy function，以及把 buffer 給 delete 掉而已。



```
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);
    tail = (tail + 1) % buffer_size;
    size++;
    buffer[tail] = item;
    pthread_cond_signal(&cond_dequeue);

    pthread_mutex_unlock(&mutex);
}
```

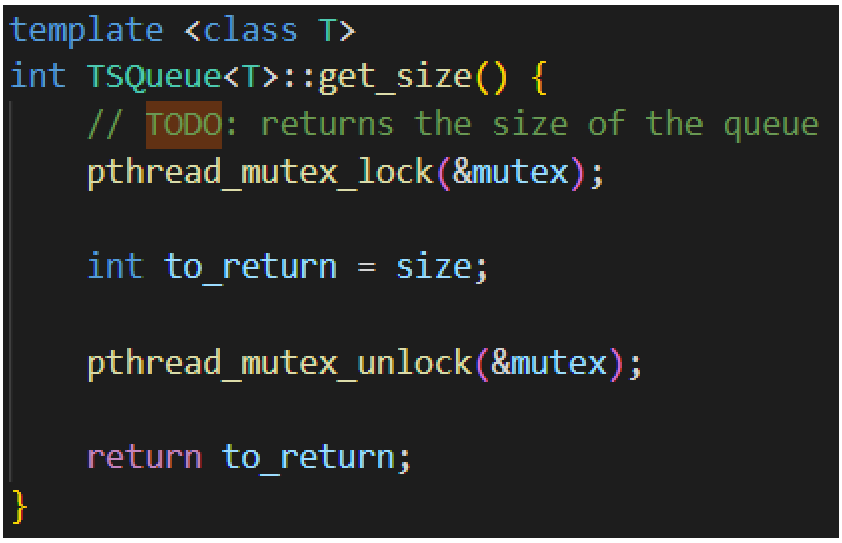
再往下便是這次作業的重頭戲——TSQueue 的 enqueue 和 dequeue。我們需要操作 mutex 和 CV 將 TSQueue 這個 shared variable 給保護好。在閱讀完講義後，我們知道 CV 也為 shared variable，所以在操作 CV 時也需要用 mutex 去包住，即 critical section。進入 critical section 後，先判斷目前 queue 的 size 是否等於 buffer_size，若為 True 則代表現在 queue 是滿的，無法再 enqueue 一個新的 item 進去，所以此時我們呼叫 cond_wait，將該 thread block 並將 mutex 釋放掉，好讓其他人可以拿到 mutex 進入 critical section，而這種使 thread 進入 sleep 的方法正是 spec 要求的 non-busyWaiting 的做法。

若 while 迴圈的條件不滿足，即 queue 現在可以再塞 item 進去，或是 return from cond_wait，則會繼續執行下面的 statement，我們便先操作 queue 的 tail 讓其前進一位，此處因為我們使用的是 circular queue 的實作方法，所以在將 tail+1 後需要去 mod buffer_size 確保沒有超出 boundary。而後因為要新增一個 item 進來，所以我們將 buffer 目前的 size+1，最後把 item 放入 queue 裡，並呼叫 cond_signal 和 release mutex 結束 enqueue



```
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);
    int old_head = head;
    head = (head + 1) % buffer_size;
    size--;
    T item = buffer[old_head];
    // buffer[old_head] = NULL;
    pthread_cond_signal(&cond_enqueue);

    pthread_mutex_unlock(&mutex);

    return item;
}
```

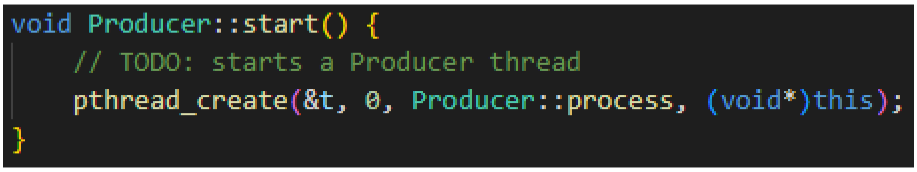
再來是 dequeue，相同的，先利用 mutex 包出一個 critical section，隨後在裡面檢查目前 queue 的 size 是否為 0，若為 True 則代表目前沒有 item 在 queue 裡，無法 dequeue，則我們呼叫 cond_wait 使 thread 被 block 住；若為 False 或 return from cond_wait，則我們繼續以下 statement，操作 queue 的 index 取得一個 item，並將 size-1，最後呼叫 cond_signal 和 release mutex 並將取得的 item return 便結束 dequeue。



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

    int to_return = size;

    pthread_mutex_unlock(&mutex);

    return to_return;
}
```

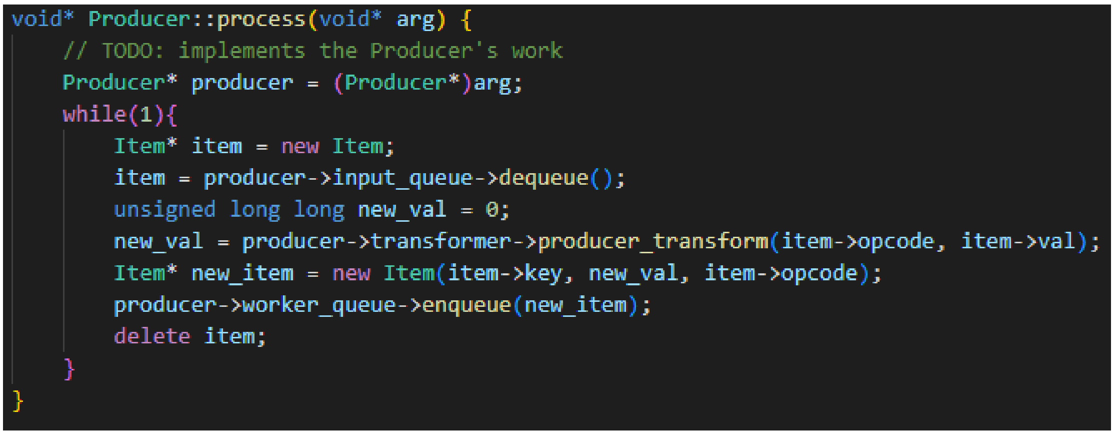
TSQueue 最後一個 function get_size ( )，簡單的將目前 queue 裡面所有的 item 數量回傳，即將 size 給 return，此處因為 TSQueue 本身為 shared variable，在操作任何有關該 class 的一切，我們都要以 mutex 包起來創造出 critical section 以達到 mutual exclusive。

再來便是 producer 與 consumer 的實作。在實作前，我們有先 trace 過 code structure 裡的 Reader，並搭配講義 pthead 的範例，得知如何創建一個 pthread 並讓其開始運作。



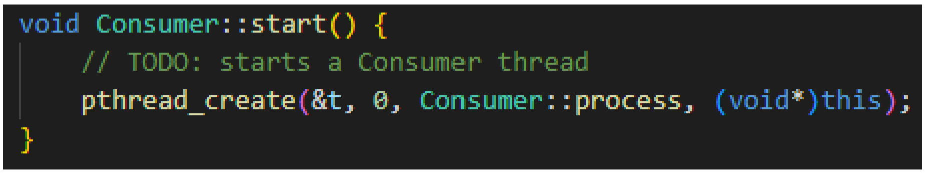
```
void Producer::start() {
    // TODO: starts a Producer thread
    pthread_create(&t, 0, Producer::process, (void*)this);
}
```

在 producer 的 start()，我們呼叫 pthread_create()，將已經在 thread.hpp 宣告好的 pthread_t type 的 t 傳入，並一同傳入 thread 的 start routine，也就是 producer 自己的 process，這樣便可以成功創建出一個 pthread。



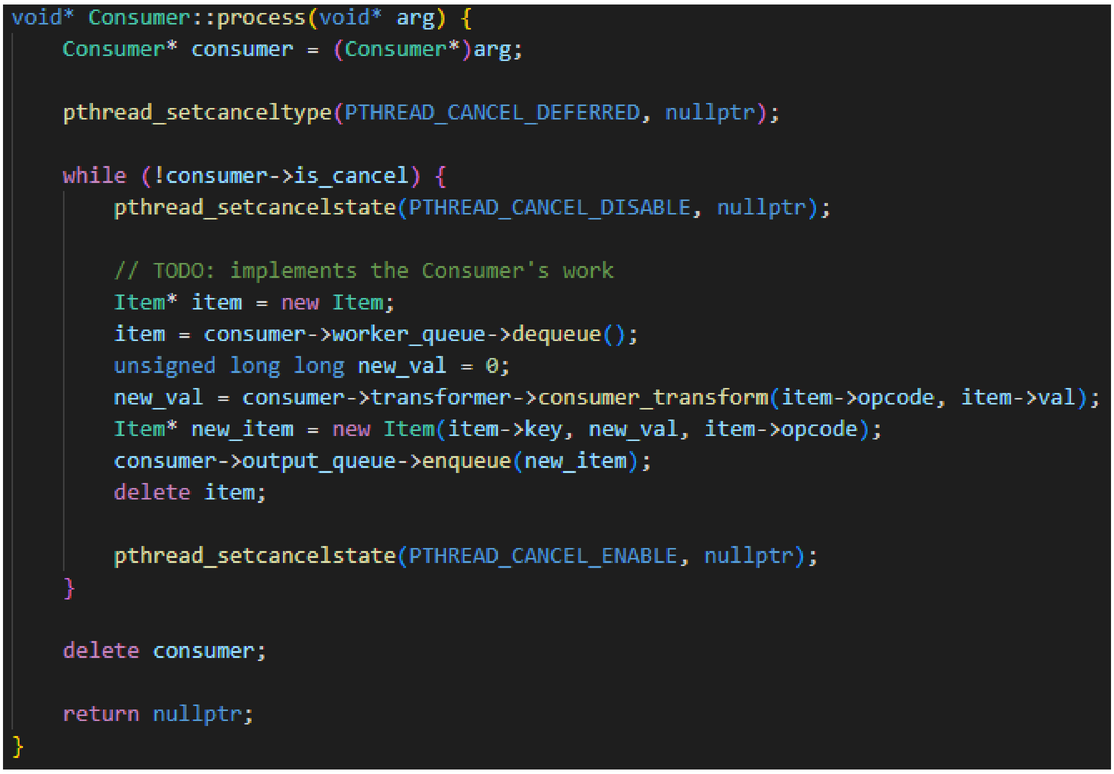
```
void* Producer::process(void* arg) {
    // TODO: implements the Producer's work
    Producer* producer = (Producer*)arg;
    while(1){
        Item* item = new Item;
        item = producer->input_queue->dequeue();
        unsigned long long new_val = 0;
        new_val = producer->transformer->producer_transform(item->opcode, item->val);
        Item* new_item = new Item(item->key, new_val, item->opcode);
        producer->worker_queue->enqueue(new_item);
        delete item;
    }
}
```

隨後的 process()是 thread 開始運作後的第一個 function，我們首先創建一個 producer 的 instance 接住 process 傳進來的 arg，其中包含 input_queue，worker_queue，transformer。隨後是一個無窮迴圈，讓 thread 可以一直運行。在迴圈內我們要做的事情在 spec 裡已經給出步驟了，先從 input_queue 中取得一個 item，所以呼叫 producer->input_queue->dequeue()，再呼叫 transformer 裡給 producer 轉換 value 使用的 function producer_transform 將新的 value 根據傳入的 opcode 和舊 value 轉換出來，最後再將這個新 value 和原本的 key 和 opcode 創建出一個新 item，將其 enqueue 到 worker_queue 裡，刪除舊 item 便結束。



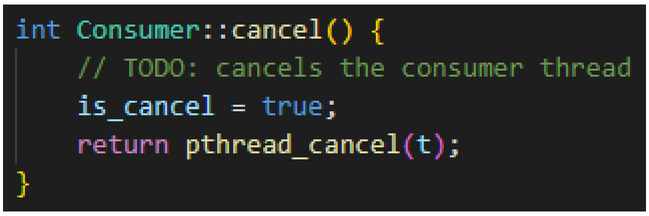
```
void Consumer::start() {
    // TODO: starts a Consumer thread
    pthread_create(&t, 0, Consumer::process, (void*)this);
}
```

在 consumer 的 start()裡，所做的事情與 producer 相同，皆是先創建出一個 pthread，只是在這裡要傳入的 start routine 要為 consumer 自己的 process function。



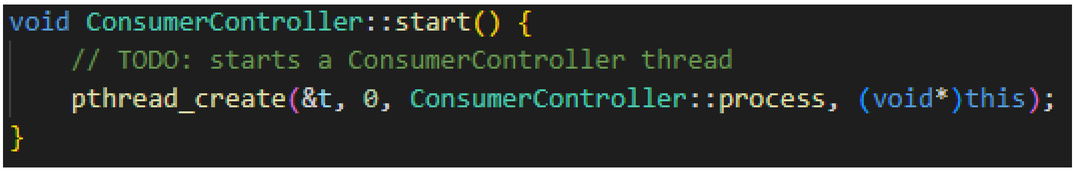
```
void* Consumer::process(void* arg) {
    Consumer* consumer = (Consumer*)arg;

    pthread_setcanceltype(PTHREAD_CANCEL_DEFERRED, nullptr);

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

        // TODO: implements the Consumer's work
        Item* item = new Item;
        item = consumer->worker_queue->dequeue();
        unsigned long long new_val = 0;
        new_val = consumer->transformer->consumer_transform(item->opcode, item->val);
        Item* new_item = new Item(item->key, new_val, item->opcode);
        consumer->output_queue->enqueue(new_item);
        delete item;

        pthread_setcancelstate(PTHREAD_CANCEL_ENABLE, nullptr);
    }

    delete consumer;

    return nullptr;
}
```

Consumer 的 process，一樣根據 spec 給出的步驟，先從 worker_queue 呼叫 dequeue 取得 item，並利用 consumer_transform 依據 opcode 和舊 value 轉換出新的 value，並再利用該新 value 和原本的 key 和 opcode 創建出一個新的 item，最後將新的 item enqueue 到 output_queue 裡並 delete 掉舊的 item 便結束



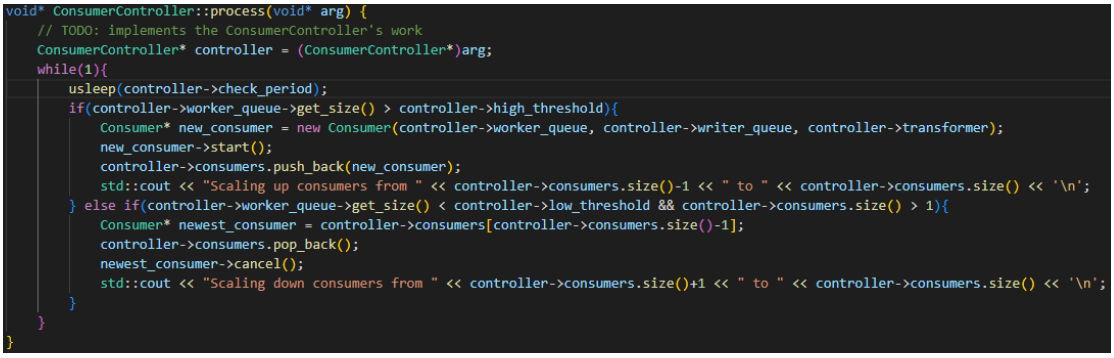
```
int Consumer::cancel() {
    // TODO: cancels the consumer thread
    is_cancel = true;
    return pthread_cancel(t);
}
```

在 consumer 的 class 裡有比 producer 多一個 function 即為 cancel()，稍微回憶了一下 process 的實作，在迴圈的判斷條件上面，使用的並不是與 producer 相同的無窮迴圈的寫法，而是去判斷 consumer->is_cancel 是否為 True，若為 true 的話則不會進入迴圈，便會直接 delete consumer。所以在 cancel()裡，我們將 is_cancel 設為 true，並在 return 時同時呼叫 pthread_cancel 將 consumer thread 給 cancel 掉。



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

ConsumerController 的 start()與前面相同，只是改成傳入 ConsumerController 的 process。

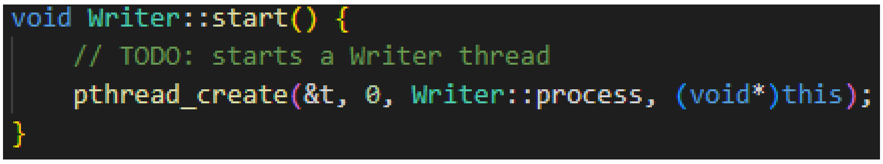


```cpp
void* ConsumerController::process(void* arg) {
    // TODO: implements the ConsumerController's work
    ConsumerController* controller = (ConsumerController*)arg;
    while(1){
        usleep(controller->check_period);
        if(controller->worker_queue->get_size() > controller->high_threshold){
            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(controller->worker_queue->get_size() < controller->low_threshold && controller->consumers.size() > 1){
            Consumer* newest_consumer = controller->consumers[controller->consumers.size()-1];
            controller->consumers.pop_back();
            newest_consumer->cancel();
            std::cout << "Scaling down consumers from " << controller->consumers.size()+1 << " to " << controller->consumers.size() << '\n';
        }
    }
}
```

在 ConsumerController 的 process 裡，因為 program 開始後便會一直存在，所以我們同樣使用無窮迴圈的寫法，在迴圈內，我們首先利用 usleep 去掌控 check_period，因為在 main.cpp 裡 check_period 的定義是 micro seconds，且 usleep 接收的單位也是 microsecond，所以就直接將 check_period 丟進 usleep 就能達到讓 thread sleep 的效果，該 thread 便會在這段 check_period 的時間內都是被 block 住，等到時間結束才會再回來執行下面 statement，符合 period 的實作。在可以執行後，我們需要根據此時 worker_queue 的容量狀況決定是否要對 consumer 進行增減。

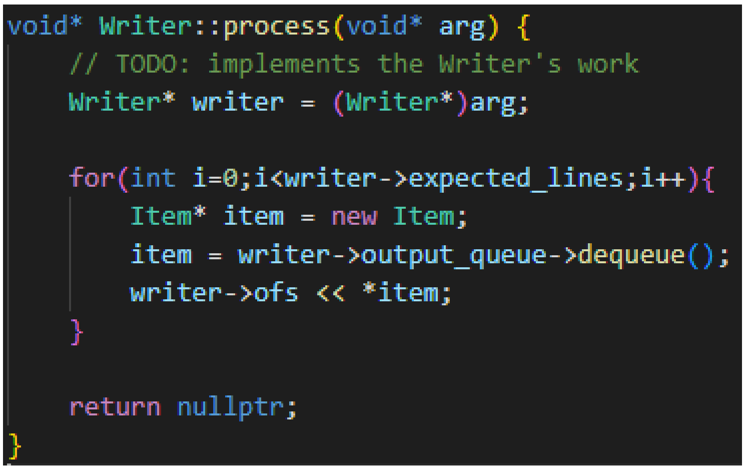
若 worker_queue 的 size 大於 high_threshold，我們需要增加一個 consumer，則將一個 consumer 的 instance 透過傳入 worker_queue，writer_queue 和 transformer 建立出來，並呼叫其 start()讓 thread 開始運作，最後將該 consumer push 到 ConsumerController 管理的一個 consumers 的 vector 裡，並用 std::out print 出訊息。

而若此時 worker_queue 的 size 小於 low_threshold，並且 consumers 裡有兩個(含)以上的 consumer，我們才需要減少一個 consumer，因為如果只有剩下一個 consumer，將其 cancel 掉後便沒有人可以再去做事，也不符合 spec 裡的 at least one 的要求。在符合條件後，我們從 consumers 裡取出最後一個 consumer，並將 vector 裡刪掉該 consumer 的導向，並呼叫該 consumer 的 cancel()將其 delete 掉，最後 print 出改變的訊息。



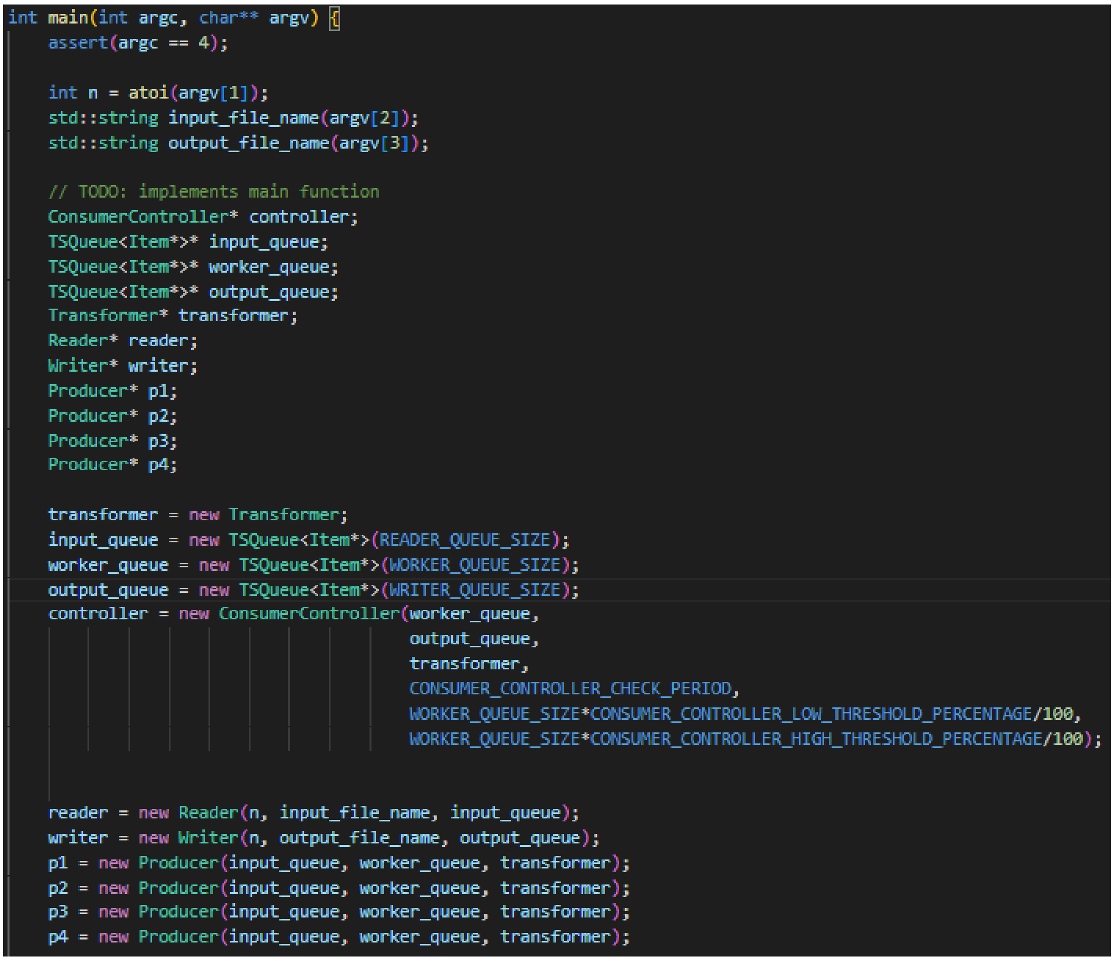
```cpp
void Writer::start() {
    // TODO: starts a Writer thread
    pthread_create(&t, 0, Writer::process, (void*)this);
}
```

在 Writer 的 start 裡傳入自己的 process 以創建 pthread。



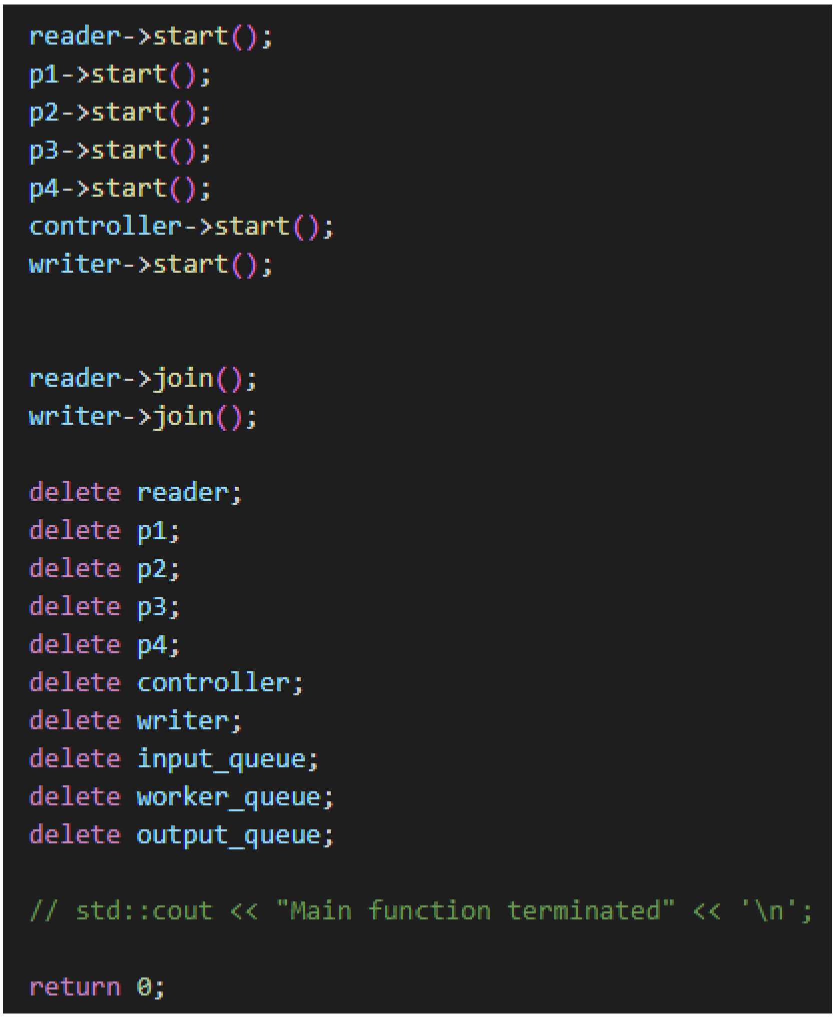
```cpp
void* Writer::process(void* arg) {
    // TODO: implements the Writer's work
    Writer* writer = (Writer*)arg;

    for(int i=0;i<writer->expected_lines;i++){
        Item* item = new Item;
        item = writer->output_queue->dequeue();
        writer->ofs << *item;
    }

    return nullptr;
}
```

在 Writer 自己的 process 裡，做的事情與 Reader 很像只是相反，根據 expected_lines 得到有多少個 item 要輸出後，利用 for loop 從 output_queue 中取出 item，並利用 operator<< 將其寫入 writer 當中。



```cpp
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
    ConsumerController* controller;
    TSQueue<Item*>* input_queue;
    TSQueue<Item*>* worker_queue;
    TSQueue<Item*>* output_queue;
    Transformer* transformer;
    Reader* reader;
    Writer* writer;
    Producer* p1;
    Producer* p2;
    Producer* p3;
    Producer* p4;

    transformer = new Transformer;
    input_queue = new TSQueue<Item*>(READER_QUEUE_SIZE);
    worker_queue = new TSQueue<Item*>(WORKER_QUEUE_SIZE);
    output_queue = new TSQueue<Item*>(WRITER_QUEUE_SIZE);
    controller = new ConsumerController(worker_queue,
                                        output_queue,
                                        transformer,
                                        CONSUMER_CONTROLLER_CHECK_PERIOD,
                                        WORKER_QUEUE_SIZE*CONSUMER_CONTROLLER_LOW_THRESHOLD_PERCENTAGE/100,
                                        WORKER_QUEUE_SIZE*CONSUMER_CONTROLLER_HIGH_THRESHOLD_PERCENTAGE/100);


    reader = new Reader(n, input_file_name, input_queue);
    writer = new Writer(n, output_file_name, output_queue);
    p1 = new Producer(input_queue, worker_queue, transformer);
    p2 = new Producer(input_queue, worker_queue, transformer);
    p3 = new Producer(input_queue, worker_queue, transformer);
    p4 = new Producer(input_queue, worker_queue, transformer);
```

在 main function 裡，首先將我們所需要的任何東西都做 initial，因為 spec 的要求 producer 需要四個，所以我們重複創建出 p1~p4 四個 producer。

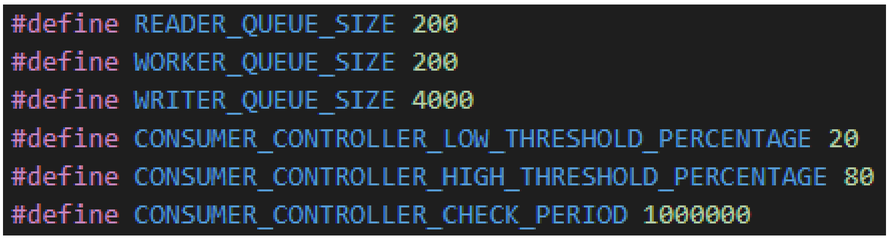


```
reader->start();
p1->start();
p2->start();
p3->start();
p4->start();
controller->start();
writer->start();


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

delete reader;
delete p1;
delete p2;
delete p3;
delete p4;
delete controller;
delete writer;
delete input_queue;
delete worker_queue;
delete output_queue;

// std::cout << "Main function terminated" << '\n';

return 0;
```

在 initial 完之後，便讓 reader、四個 producer、controller 和 writer 開始運作，call 個別的 start()。而因為此時會是一個同步執行的 program，判斷 program 是否結束會依據 reader 是否已經讀完，writer 是否已經寫完作為判斷，所以我們 call reader 的 join 和 writer 的 join 代表我們需要等到該兩個 thread 都全部執行完畢才可以繼續往下執行 statement，而該兩者執行完後，也代表整個 program 要結束了，所以我們便將剛剛 create 出來的 resource 全部 delete 掉，便結束整個 program。

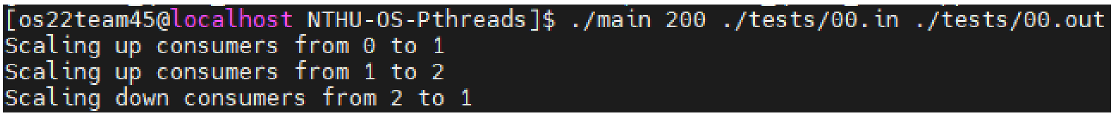
Experiment

在每一項實驗中，對照組都是一樣的，所以我們先將對照組的結果寫在最前面，這樣後面就不用一直重複:



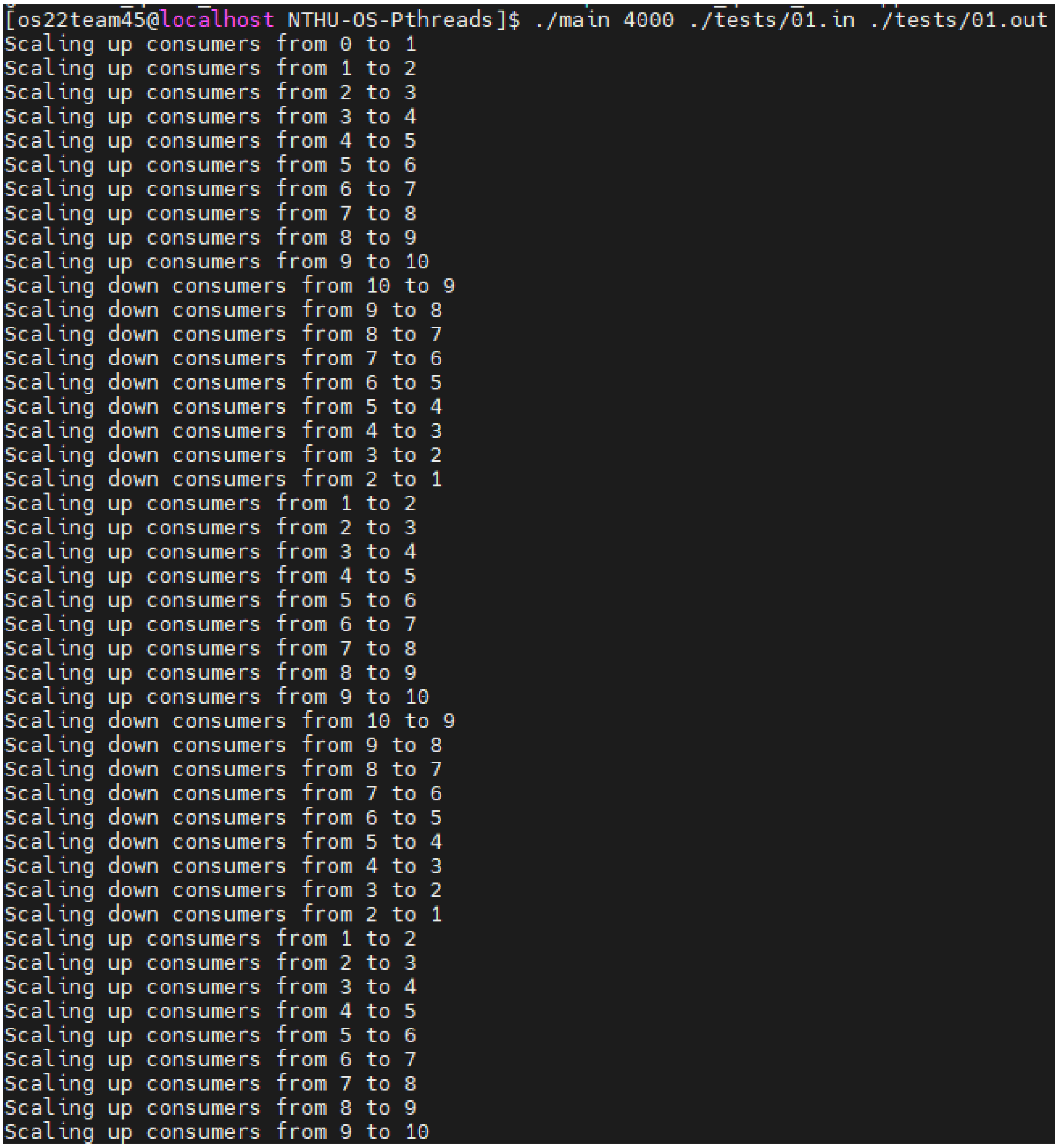
```
#define READER_QUEUE_SIZE 200
#define WORKER_QUEUE_SIZE 200
#define WRITER_QUEUE_SIZE 4000
#define CONSUMER_CONTROLLER_LOW_THRESHOLD_PERCENTAGE 20
#define CONSUMER_CONTROLLER_HIGH_THRESHOLD_PERCENTAGE 80
#define CONSUMER_CONTROLLER_CHECK_PERIOD 1000000
```

`./main 200 ./tests/00.in ./tests/00.out`的對照組 : scale up 到 2 個 consumers 就結束，總共 2 個 scale up, 1 個 scale down。



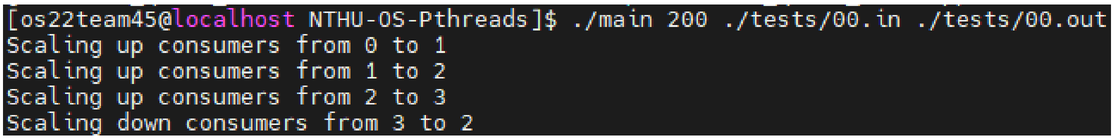
```
[os22team45@localhost NTHU-OS-Pthreads]$ ./main 200 ./tests/00.in ./tests/00.out
Scaling up consumers from 0 to 1
Scaling up consumers from 1 to 2
Scaling down consumers from 2 to 1
```

`./main 4000 ./tests/01.in ./tests/01.out` 的對照組: 最高會 scale up 到 10 個 consumers，並且總共有 28 個 scale up, 18 個 scale down。



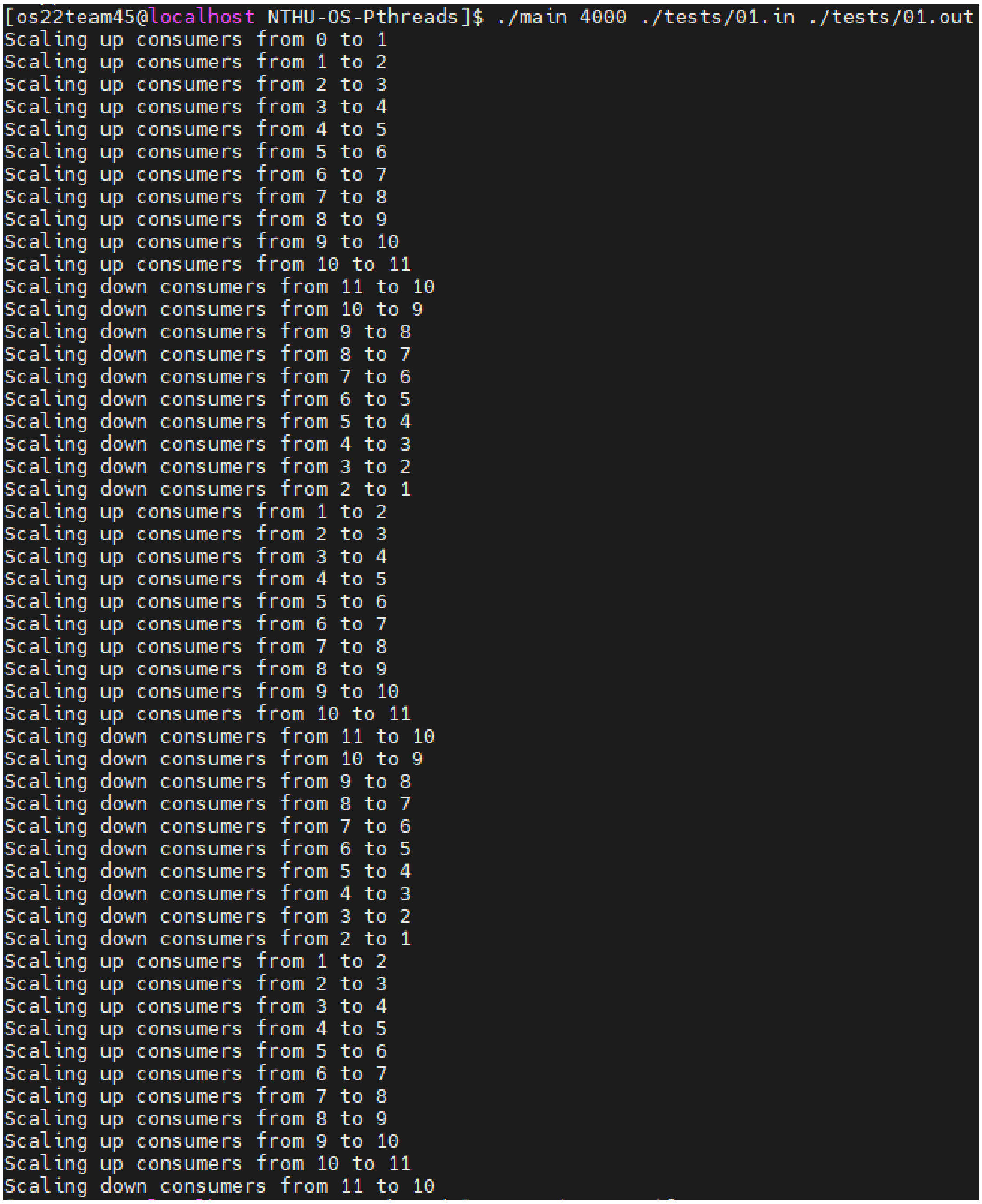
```
[os22team45@localhost NTHU-OS-Pthreads]$ ./main 4000 ./tests/01.in ./tests/01.out
Scaling up consumers from 0 to 1
Scaling up consumers from 1 to 2
Scaling up consumers from 2 to 3
Scaling up consumers from 3 to 4
Scaling up consumers from 4 to 5
Scaling up consumers from 5 to 6
Scaling up consumers 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 down consumers from 10 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
Scaling down consumers from 5 to 4
Scaling down consumers from 4 to 3
Scaling down consumers from 3 to 2
Scaling down consumers from 2 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
Scaling down consumers from 7 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
Scaling down consumers from 2 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
```

1. **Different values of CONSUMER_CONTROLLER_CHECK_PERIOD**.
   對照組:CONSUMER_CONTROLLER_CHECK_PERIOD = 1000000
   實驗駔: CONSUMER_CONTROLLER_CHECK_PERIOD = 500000
   在`./main 200 ./tests/00.in ./tests/00.out`的實驗組中，最高 scale up 到 3 個 consumer，總共有 3 個 scale up, 1 個 scale down。



```
[os22team45@localhost NTHU-OS-Pthreads]$ ./main 200 ./tests/00.in ./tests/00.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
```

在`./main 4000 ./tests/01.in ./tests/01.out`的實驗組中，最高 scale up 到 11 個 consumer，總共有 31 個 scale up, 21 個 scale down。



```
[os22team45@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
Scaling down consumers from 8 to 7
Scaling down consumers from 7 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
Scaling down consumers from 2 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
Scaling down consumers from 8 to 7
Scaling down consumers from 7 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
Scaling down consumers from 2 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
```

由此可知，當檢查的間距變短後，在固定時間內會檢查的次數就變多了，因此就有可能會生產更多 consumers。

2. **Different values of CONSUMER_CONTROLLER_LOW_THRESHOLD_PERCENTAGE and CONSUMER_CONTROLLER_HIGH_THRESHOLD_PERCENTAGE.**

對照組:

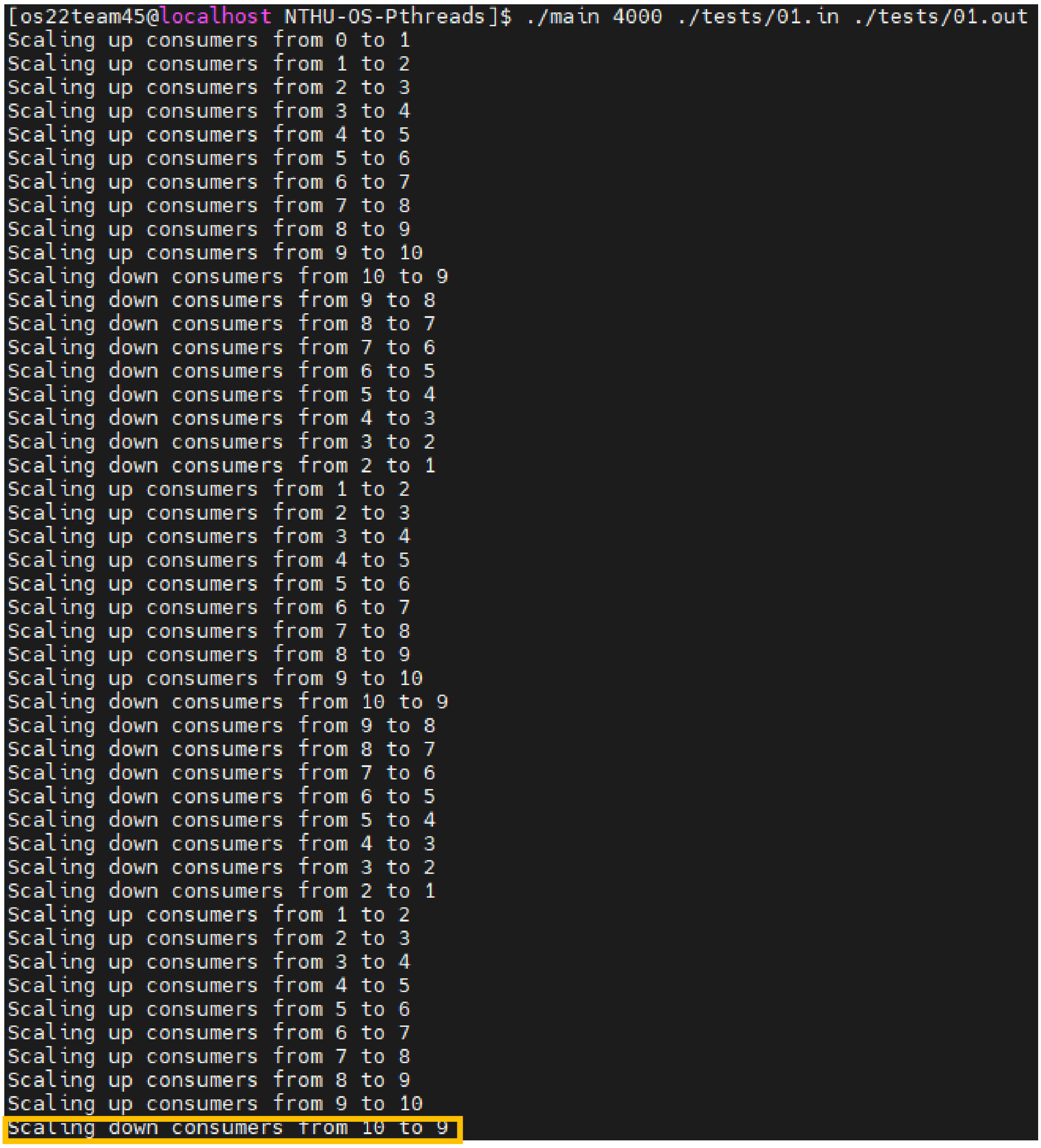
CONSUMER_CONTROLLER_LOW_THRESHOLD_PERCENTAGE = 20

實驗組:

CONSUMER_CONTROLLER_LOW_THRESHOLD_PERCENTAGE = 50

在`./main 200 ./tests/00.in ./tests/00.out`的實驗組中，跟對照組完全一樣。

在`./main 4000 ./tests/01.in ./tests/01.out`的實驗組中，最高 scale up 到 10 個 consumer，總共有 28 個 scale up, 19 個 scale down。



```
[os22team45@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 down consumers from 10 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
Scaling down consumers from 5 to 4
Scaling down consumers from 4 to 3
Scaling down consumers from 3 to 2
Scaling down consumers from 2 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
Scaling down consumers from 7 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
Scaling down consumers from 2 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
```

由此可以知道，當我們調高 low_threshold，代表 worker_queue 中 item 的數量更有可能低於 threshold，導致最後會刪除更多的 consumer。

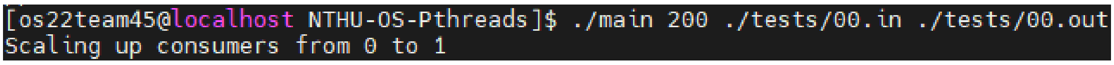
對照組:

CONSUMER_CONTROLLER_HIGH_THRESHOLD_PERCENTAGE = 80

實驗組:

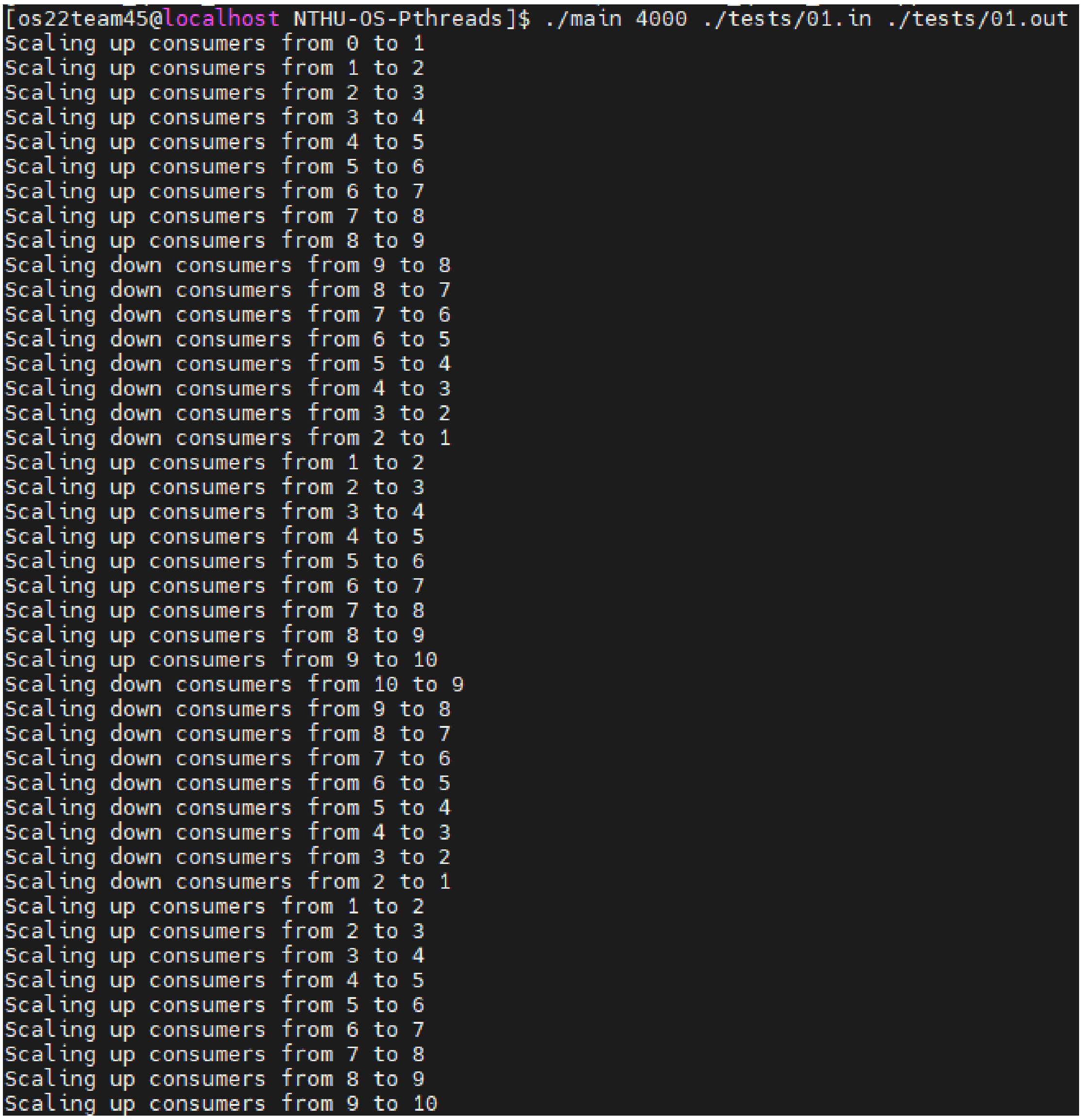
CONSUMER_CONTROLLER_HIGH_THRESHOLD_PERCENTAGE = 90

在`./main 200 ./tests/00.in ./tests/00.out`的實驗組中，最高 scale up 到 1 個 consumer，總共有 1 個 scale up, 0 個 scale down。



```
[os22team45@localhost NTHU-OS-Pthreads]$ ./main 200 ./tests/00.in ./tests/00.out
Scaling up consumers from 0 to 1
```

在`./main 4000 ./tests/01.in ./tests/01.out`的實驗組中，最高 scale up 到 10 個 consumer，總共有 27 個 scale up, 17 個 scale down。



```
[os22team45@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
Scaling down consumers from 5 to 4
Scaling down consumers from 4 to 3
Scaling down consumers from 3 to 2
Scaling down consumers from 2 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
Scaling down consumers from 7 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
Scaling down consumers from 2 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
```

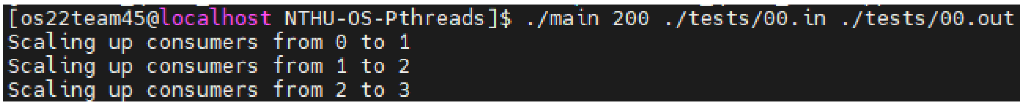
由此可以知道，當我們調高 high_threshold，代表 worker_queue 中 item 的數量更不可能高於 threshold，導致會有較少的 consumer 被生產。

3. Different values of WORKER_QUEUE_SIZE.

對照組: WORKER_QUEUE_SIZE = 200

實驗組: WORKER_QUEUE_SIZE = 150

在`./main 200 ./tests/00.in ./tests/00.out`的實驗組中，最高 scale up 到 3 個 consumer，總共有 3 個 scale up, 0 個 scale down。



```
[os22team45@localhost NTHU-OS-Pthreads]$ ./main 200 ./tests/00.in ./tests/00.out
Scaling up consumers from 0 to 1
Scaling up consumers from 1 to 2
Scaling up consumers from 2 to 3
```

在`./main 4000 ./tests/01.in ./tests/01.out`的實驗組中，跟對照組一樣。

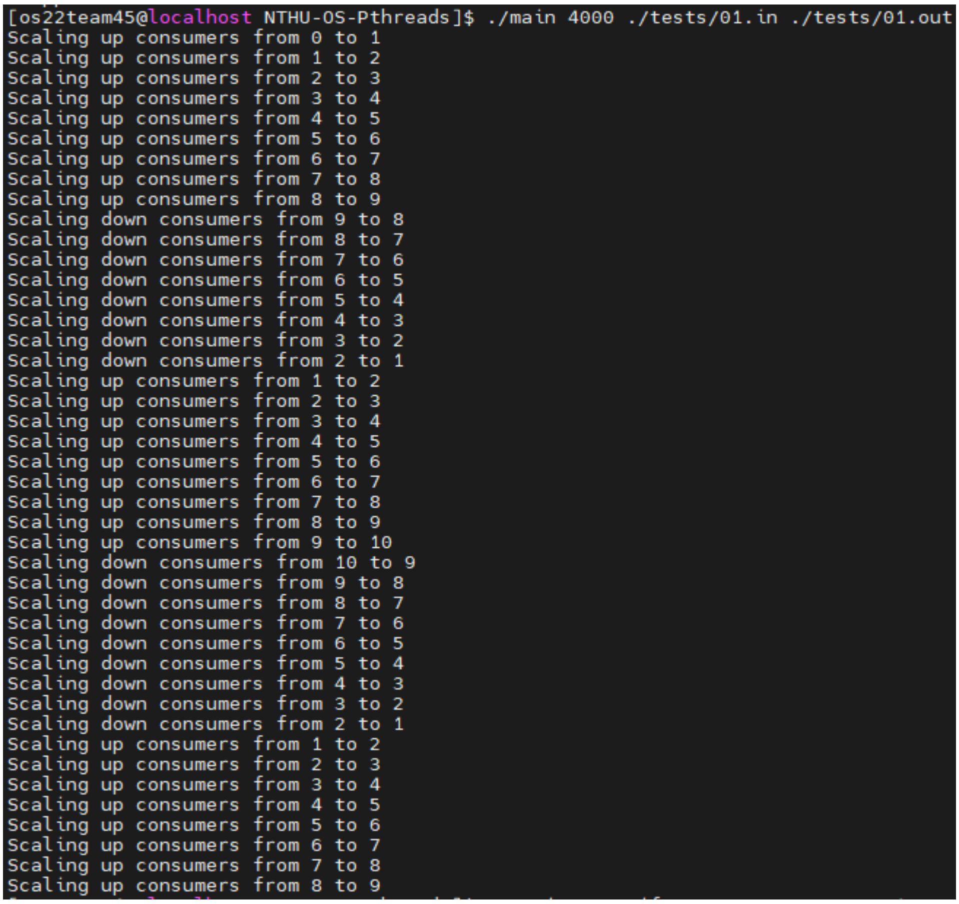
這裡可以看出 worker_queue_size 變小導致上下限都縮小，因此 scale up 變得更容易達成。

對照組: WORKER_QUEUE_SIZE = 200

實驗組: WORKER_QUEUE_SIZE = 300

在`./main 200 ./tests/00.in ./tests/00.out`的實驗組中，完全沒有任何動作，推測是因為 WORKER_QUEUE_SIZE 太大導致 worker_queue 中的 item 數量沒辦法超過 high_threshold，所以沒有 consumer 被產生。

在`./main 4000 ./tests/01.in ./tests/01.out`的實驗組中，最高 scale up 到 10 個 consumer，總共有 26 個 scale up, 17 個 scale down。



```
[os22team45@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
Scaling down consumers from 5 to 4
Scaling down consumers from 4 to 3
Scaling down consumers from 3 to 2
Scaling down consumers from 2 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
Scaling down consumers from 7 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
Scaling down consumers from 2 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
```

4. What happens if WRITER_QUEUE_SIZE is very small?

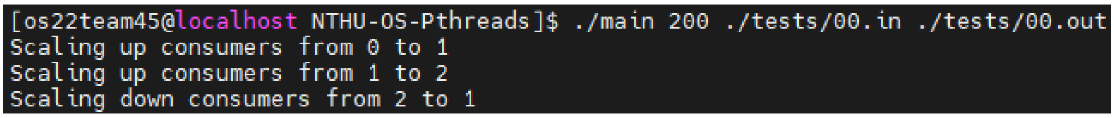
對照組: WRITER_QUEUE_SIZE = 4000
實驗組: WRITER_QUEUE_SIZE = 10
在`./main 200 ./tests/00.in ./tests/00.out`的實驗組中，跟對照組一樣
在`./main 4000 ./tests/01.in ./tests/01.out`的實驗組中，跟對照組一樣

5. What happens if READER_QUEUE_SIZE is very small?
對照組: READER_QUEUE_SIZE = 200
實驗組: READER_QUEUE_SIZE = 10
在`./main 200 ./tests/00.in ./tests/00.out`的實驗組中，跟對照組一樣
在`./main 4000 ./tests/01.in ./tests/01.out`的實驗組中，跟對照組一樣

由 4 跟 5 可以知道，writer_queue 跟 reader_queue 的大小上限並不會影響他們的行為，如果 queue 滿了，其他要把 item 丟進 queue 的 thread 便會等待到 queue 有多的空間在繼續執行，所以 queue 的上限只會影響到 thread 執行的效率而已。

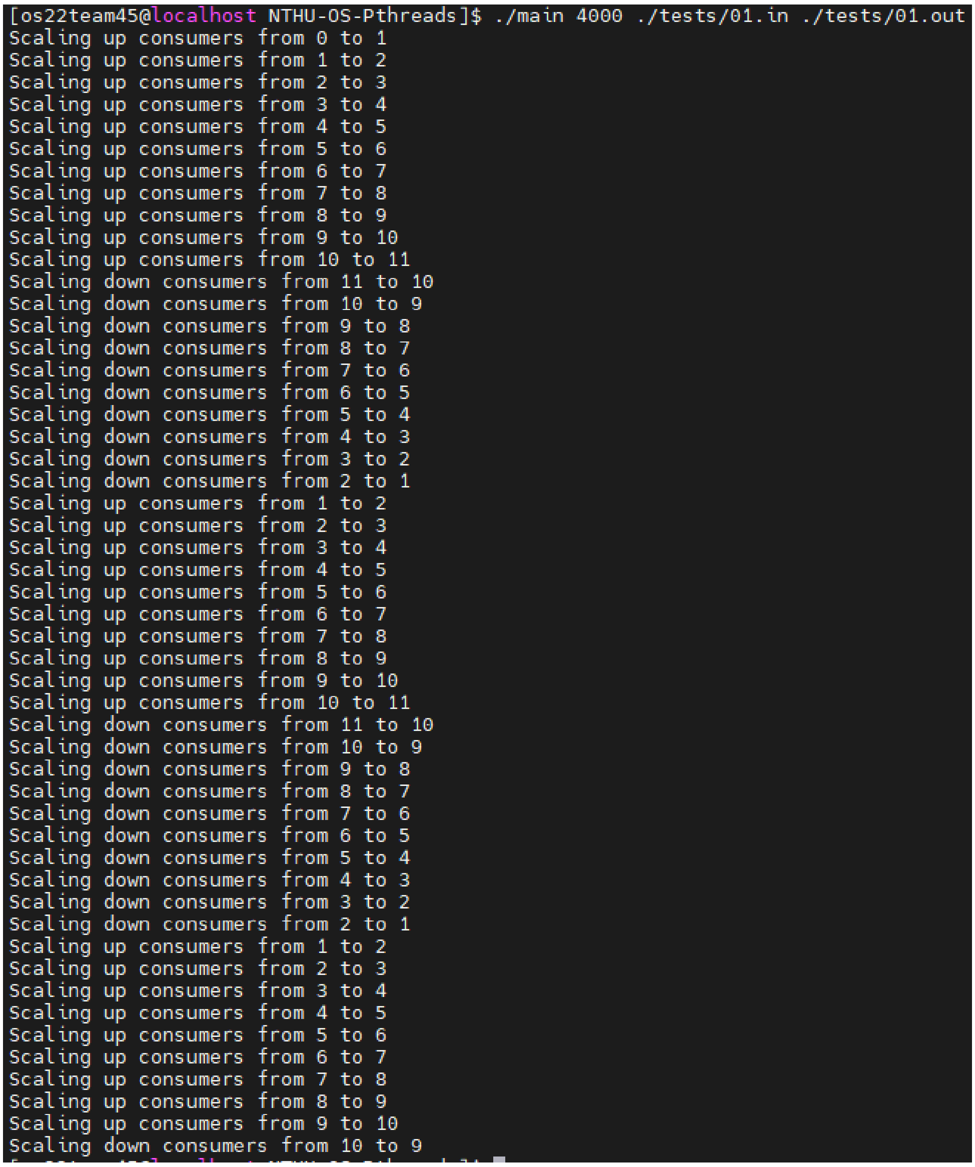
Additional experiment:

6. 對照組：
WORKER_QUEUE_SIZE = 200
CONSUMER_CONTROLLER_LOW_THRESHOLD_PERCENTAGE=20
CONSUMER_CONTROLLER_HIGH_THRESHOLD_PERCENTAGE=80
實驗組：
WORKER_QUEUE_SIZE = 400
CONSUMER_CONTROLLER_LOW_THRESHOLD_PERCENTAGE=10
CONSUMER_CONTROLLER_HIGH_THRESHOLD_PERCENTAGE=40
在`./main 200 ./tests/00.in ./tests/00.out`的實驗組中，跟對照組一樣



```
[os22team45@localhost NTHU-OS-Pthreads]$ ./main 200 ./tests/00.in ./tests/00.out
Scaling up consumers from 0 to 1
Scaling up consumers from 1 to 2
Scaling down consumers from 2 to 1
```

在`./main 4000 ./tests/01.in ./tests/01.out`的實驗組中，一開始會多 scale up 一個 consumer，最高 scale up 到 11 個 consumer，而在最後會多 scale down 一個 consumer。

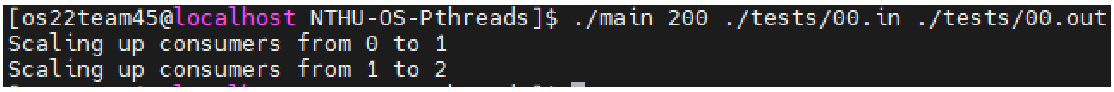


```
[os22team45@localhost NTHU-OS-Pthreads]$ ./main 4000 ./tests/01.in ./tests/01.out
Scaling up consumers from 0 to 1
Scaling up consumers from 1 to 2
Scaling up consumers from 2 to 3
Scaling up consumers from 3 to 4
Scaling up consumers from 4 to 5
Scaling up consumers from 5 to 6
Scaling up consumers 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
Scaling down consumers from 8 to 7
Scaling down consumers from 7 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
Scaling down consumers from 2 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
Scaling down consumers from 8 to 7
Scaling down consumers from 7 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
Scaling down consumers from 2 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
```

由 consumer controller 要增減 consumer 的公式計算，對照組 worker queue size 為 200，low=200*20/100=40，high=200*80/100=160，而實驗組 worker queue size 為 400，low=400*10/100=40，high=400*40/100=160，計算的值會是一模一樣，所以理應在做增減 consumer 時的情況要是一樣，但我們發現執行出來的結果卻與對照組不相同，因此推測可能是因為 worker queue size 變大後，原本執行過程中會使得 worker queue 塞滿的情況在此時不會出現，使得可以一直塞入 worker queue 讓其維持較高的使用率以達到多一個 consumer。

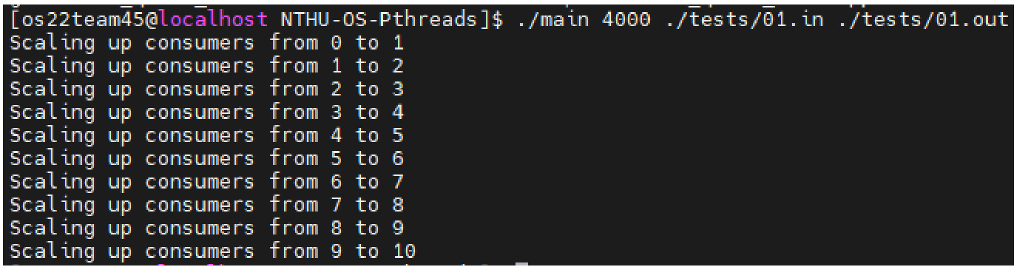
7. **對照組：**
   CONSUMER_CONTROLLER_LOW_THRESHOLD_PERCENTAGE=20
   CONSUMER_CONTROLLER_HIGH_THRESHOLD_PERCENTAGE = 80
   實驗組：
   CONSUMER_CONTROLLER_LOW_THRESHOLD_PERCENTAGE=0

CONSUMER_CONTROLLER_HIGH_THRESHOLD_PERCENTAGE = 80



```
[os22team45@localhost NTHU-OS-Pthreads]$ ./main 200 ./tests/00.in ./tests/00.out
Scaling up consumers from 0 to 1
Scaling up consumers from 1 to 2
```

在`./main 200 ./tests/00.in ./tests/00.out`的實驗組中，如上圖所示，一開始與對照組一樣先 scale up 到 2 個 consumer，但之後因為 low threshold 為 0，要等到 worker queue 都沒有東西才會刪除 consumer，所以在執行過程中沒有出現這種情形，就會一直保持 2 個 consumer 直到程式結束。



```
[os22team45@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
```

在`./main 4000 ./tests/01.in ./tests/01.out`的實驗組中，如上圖所示，與`./main 200 ./tests/00.in ./tests/00.out`相同，只有 scale up 沒有 scale down。

8. **對照組：**

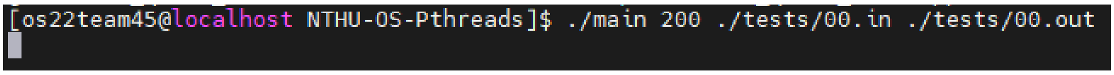
CONSUMER_CONTROLLER_LOW_THRESHOLD_PERCENTAGE=20

CONSUMER_CONTROLLER_HIGH_THRESHOLD_PERCENTAGE = 80

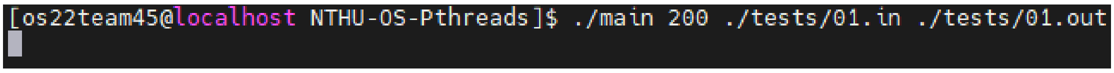
實驗組：

CONSUMER_CONTROLLER_LOW_THRESHOLD_PERCENTAGE=20

CONSUMER_CONTROLLER_HIGH_THRESHOLD_PERCENTAGE=100



```
[os22team45@localhost NTHU-OS-Pthreads]$ ./main 200 ./tests/00.in ./tests/00.out
```



```
[os22team45@localhost NTHU-OS-Pthreads]$ ./main 200 ./tests/01.in ./tests/01.out
```

當 high threshold 設成 100 時，不論是`./main 200 ./tests/00.in ./tests/00.out`還是`./main 4000 ./tests/01.in ./tests/01.out`都不會有 consumer 產生，因為 consumer controller 要大於 threshold 才會產生 consumer，而不會有大於 queue size 的情況所以沒有 consumer 產生造成 program stuck。

Difficulty：

本次的作業整體難度上比起 scheduling 和 file system 來說是簡單不少，不會有那種無從下手的迷茫，也不會在眾多 file 裡不知道自己改了什麼或是還要改什麼，但也不是說此次作業就沒有難度，在 program 上需要考慮到多個 thread 同時存取 share data 時，要保證 data consistentcy。特別是在實作 TSQueue 時，我們一開始的 enqueue 和 dequeue 只有用 condition variable，但是忘記 condition variable 本身也是一個 shared variable，導致我們的結果錯誤，並且在 debug 時都沒想到是這裡會出錯，在其他 file 上面東改西改浪費了不少時間，直到我們在複習 pthread 部分的講義時，才想起來 condition variable 需要搭配 mutex，這才解開困擾許久的問題。

心得:

這次的 pthread 有比之前的 assignment 簡單一些，由於很多函式都是已經定義好的，所以這次只要搞懂 mutex 跟 condition variable 需要被呼叫的時機就好。
經過了這次的作業，我們也更了解有關 thread programming 的細節，把上課時學到的東西可以實際應用。這也是這學期的最後一個作業，很感謝我的隊友跟我一起努力完成每次的作業，整個學期下來互相督促及鼓勵。同時也很感謝助教跟老師都很有耐心地回答我們的問題，也幫助我們更順利的完成作業!