Operating System

Pthread

Team Member(Team 10): 林子謙 (112065517)、陳奕潔 (110000212)

Contributions: Both contributed to implementation and report writing.

Contents

1 Implementation

```cpp
class Thread {
public:
    // to start a new pthread work
    virtual void start() = 0;

    // to wait for the pthread work to complete
    virtual int join();

    // to cancel the pthread work
    virtual int cancel();
protected:
    pthread_t t;
};

int Thread::join() {
    return pthread_join(t, 0);
}

int Thread::cancel() {
    return pthread_cancel(t);
}
```

Figure 1: Thread class

Reader、Producer、Consumer、ConsumerContoroller，和 Writer 都是繼承 Thread，而根據各個 class 的角色，可能會覆寫其 methods。

1.1 TSQueue

```cpp
TSQueue<T>::TSQueue(int buffer_size) : buffer_size(buffer_size) {
    // TODO: implements TSQueue constructor
    pthread_mutex_init(&mutex, 0);
    pthread_cond_init(&cond_enqueue, 0);
    pthread_cond_init(&cond_dequeue, 0);

    buffer = new T[buffer_size];
    size = 0;
    head = tail = 0;
}
```

Figure 2: TSQueue::TSQueue()

TSQueue constructor 會初始化 buffer、lock（mutex）及 condition variables（cond_wnqueue、cond_dequeue）。另外，也把目前 buffer 所佔的資料量（size）、丟新資料的 index（tail），和拿資料的 index（head）設為 0。

TSQueue 的 destructor（Figure 3），則負責刪除 buffer、lock（mutex）及 condition variables（cond_enqueue、cond_dequeue）。

```cpp
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;
}
```

Figure 3: TSQueue::~TSQeue()

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

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

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

Figure 4: TSQueue::enqueue()

由於 TSQueue::enqueue() 會改變 queue 的內容，size 和 tail 都是 shared data，因此需要 mutex 才可修改 queue 的內容（進入 critical section）。

若成功取得 mutex，但目前的 buffer 已經滿了（size == buffer_size），則先把自己放進 waiting queue，並釋放 mutex，透過自己的 condition variable，cond_enqueue，來確認 queue 是否有空間。在此使用 while 是為了確保執行的正確性：舉例而言，若有多個 thread 都在等待，當 thread 0 先搶到 lock 導致 queue 又滿了，thread 1 搶到 lock 時，會再一次檢查 queue 是否滿了，而不會直接塞資源進去。

若還沒滿，就將資源（item）放入 tail 的位置並更新 tail 和 size。接著通知等待此資源的 thread，而因為可能不只一個 threads 在等，故用 pthread_cond_broadcast() 廣而告之。最後釋放 mutex。

TSQueue::dequeue()（Figure 5）與 TSQueue::enqueue() 一樣會改變 queue 的內容，，size 和 head 都是 shared data，因此也需要 mutex 才可修改 queue 的內容（進入 critical section）。

若成功取得 mutex，但目前的 buffer 已經空了（size == 0），則把自己放進 waiting queue，並釋放 mutex，透過自己的 condition variable，cond_dequeue，來確認有資源可用。在此同樣使用 while 確保執行的正

```cpp
template <class T>
T TSQueue<T>::dequeue() {
    // TODO: dequeues the first element of the queue
    pthread_mutex_lock(&mutex);  // start dequeue
    while (size == 0) {
        pthread_cond_wait(&cond_dequeue, &mutex);
    }

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

    pthread_cond_broadcast(&cond_enqueue);
    pthread_mutex_unlock(&mutex);

    return item;
}
```

Figure 5: TSQueue::dequeue()

確性。

若還有資源可用，就將資源從 head 的位置拿出並更新 head 和 size。因為有了空位，接著通知在 enqueue 的 condition variable，叫醒該 thread，而因為可能不只一個 threads 在等，故用 pthread_cond_broadcast() 廣而告之。最後釋放 mutex 並回傳取得的資源。

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

    return stable_val;
}
```

Figure 6: TSQueue::get_size()

在 TSQueue::get_size() 中，為了確保拿到的 size 值是穩定的，所以也用 critical section 包起來。

1.2 Producer

在 Producer::start() 中，會 create 一個 thread（t 為 Producer attribute，代表 threadID），去執行 Producer::process() 這個函式。Producer::process() 的輸入參數為 Producer 自己。

Producer::process() 中，會先用指標把 Producer 自己接住。接著，不斷從 input queue 拿 item，將其 val 利用 Transformer::producer_transform()

```
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
    // takes Item from the Input Queue
    // applies the Item with the Transformer::producer transform function: transformer.producer_transform()
    // puts the result Item into the Worker Queue
    Producer* producer = (Producer*)arg;

    while (1) {
        Item *item = producer->input_queue->dequeue();
        item->val = producer->transformer->producer_transform(item->opcode, item->val);
        producer->worker_queue->enqueue(item);
    }

    return nullptr;
}
```

Figure 7: Producer::start() and Producer::process()

轉換後，丟到 worker queue。Transformer::producer_transform() 會根據 item 的 opcode 找到對應的 spec 來轉換 item 的 val。

1.3 Consumer

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

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

    return pthread_cancel(t);
}
```

Figure 8: Consumer::start() and Consumer::cancel()

Consumer::start() 的實作邏輯與 Producer::start() 相同。

Consumer::cancel() 則更新 attribute is_cancel 的值為 true，再回傳取消 thread 執行成功與否的結果。

```cpp
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
        // Take an Item from the Worker Queue
        // transformer.consumer_transform()
        // Put the Item with new value into the Output Queue
        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;
}
```

Figure 9: Consumer::process()

Consumer::process()（Figure 9）的實作邏輯大致與 Producer::process() 相同，但要檢查是否被取消。

pthread_setcanceltype 的 PTHREAD_CANCEL_DEFERRED 讓 thread 的取消方式是，等到其再次被呼叫才停止。這樣能避免打斷 thread 正在執行的計算而影響最終程式的結果。

pthread_setcancelstate 的 PTHREAD_CANCEL_DISABLE 則讓這個 thread 不能用函式取消。因此，等到 thread 將 worker queue 拿出 item，將其 val 利用 Transformer::comsumer_transform() 轉換，並丟到 output queue 後，才允許系統用函式取消掉這個 thread。

1.4 ConsumerController

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

void* ConsumerController::process(void* arg) {
    // TODO: implements the ConsumerController's work
    // In the beginning, no Consumer thread is created by ConsumerController
    // Check Worker Queue status periodically
    // Worker Queue size > high_threshold -> new Consumer
    // Worker Queue size < low_threshold -> call Consumer->cancel
    ConsumerController *controller = (ConsumerController*)arg;
    while (1) {
        usleep(controller->check_period);

        if (controller->worker_queue->get_size() > controller->high_threshold) {
            Consumer *one_worker = new Consumer(controller->worker_queue, controller->writer_queue, controller->transformer)

            controller->consumers.push_back(one_worker);
            one_worker->start();


            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 *one_worker = controller->consumers.back();

            controller->consumers.pop_back();
            one_worker->cancel();

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

Figure 10: ConsumerController::start() and ConsumerController::process()

ConsumerController::start() 的實作邏輯與 Producer::start() 相同。

ConsumerController::process() 同樣會先用指標把 ConsumerController 自己接住。接著，用 while 及 usleep() 週期性檢查目前的 comsumers（存在 ConsumerController 的 attribute consumers）與 worker queue 的平衡。comsumer 的數量是用 vector 的 method size() 取得，worker queue 裡的數量則是用 get_size() 得到。

如果 worker queue 裡的數量大於 high_threshold，就新增一個 consumer，丟進 ConsumerController 所管理的 consumers 中，並啟動它。最後，印出 scale up 的訊息。

如果 worker queue 裡的數量小於 low_threshold 而且目前有兩個以上的 consumer（Spec 要求最後至少保留一個），就從 ConsumerController 所管理的 consumers 中，拿出一個 consumer 並取消它。最後，印出 scale down 的訊息。

1.5 Writer

```
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--) {
        // Take an Item from the Output Queue
        Item *item = writer->output_queue->dequeue();
        writer->ofs << *item;
    }

    return nullptr;
}
```

Figure 11: Writer::start() and Writer::process()

Writer::start() 的實作邏輯與 Producer::start() 相同。

Writer::process() 的實作邏輯也大致與 Producer::process() 相同。

它同樣會先用指標把 Writer 自己接住。接著，不斷從 output queue 拿 item 直到 expected_lines 為 0 才 return。

在迴圈中，它將拿到的 item 之 key、val，和 opcode 依序寫進 output_file 這個檔案裡（Item class 有重新定義 <<）。

1.6 main.cpp

由於 part 2 要調整不同參數，觀察整體程式的效能，我們另外寫了用於實驗的程式碼，這些程式碼目前已經被註解。

在第 20 及 21 行，是用於記錄程式執行時間的設定。第 24 行的 n 代表要轉換多少行，也就是 reader 跟 writer 的 expected_lines。第 31 至 35 行則 create 要用到的 transformer、reader queue、worker queue、writer queue 及 controller 等指標。

第 37 至 58 行是用於實驗的程式碼。首先會讀進各個參數的值：reader size、worker size、writer size、low threshold、high threshold 以及檢查頻率（check period），再根據這些值去建立 reader queue、worker queue、writer queue 及 controller。其中 controller 根據 comsumer 所負責的 queue，拿到 worker queue、writer queue，並做好 threshold 百分比的轉換。

```cpp
int main(int argc, char** argv) {
    assert(argc == 4);
    // struct timespec start, end;
    // clock_gettime(CLOCK_MONOTONIC, &start);


    int n = atoi(argv[1]);
    std::string input_file_name(argv[2]);
    std::string output_file_name(argv[3]);
    // int doExperiment = atoi(argv[4]);

    // TODO: implements main function
    // Construct
    Transformer *transformer = new Transformer();
    TSQueue<Item*>* reader_queue = NULL;
    TSQueue<Item*>* worker_queue = NULL;
    TSQueue<Item*>* writer_queue = NULL;
    ConsumerController* controller = NULL;
```

Figure 12: Initialization

```cpp
    // if (doExperiment == 1) {
    //  int reader_size = atoi(argv[5]);
    //  int worker_size = atoi(argv[6]);
    //  int writer_size = atoi(argv[7]);
    //  int low = atoi(argv[8]);
    //  int high = atoi(argv[9]);
    //  int check_period = atoi(argv[10]);
    //  std::cout << "reader_size: " << reader_size << std::endl;
    //  std::cout << "worker_size: " << worker_size << std::endl;
    //  std::cout << "writer_size: " << writer_size << std::endl;
    //  std::cout << "low: " << low << std::endl;
    //  std::cout << "high: " << high << std::endl;
    //  std::cout << "check_period: " << check_period << std::endl;

    //  reader_queue = new TSQueue<Item*>(reader_size);
    //  worker_queue = new TSQueue<Item*>(worker_size);
    //  writer_queue = new TSQueue<Item*>(writer_size);

    //  controller = new ConsumerController(worker_queue, writer_queue, transformer,
    //                                      check_period,
    //                                      worker_size * low / 100,
    //                                      worker_size * high / 100);
    // } else {
    //      std::cout << "default setting.\n";
    reader_queue = new TSQueue<Item*>(READER_QUEUE_SIZE);
    worker_queue = new TSQueue<Item*>(WORKER_QUEUE_SIZE);
    writer_queue = new TSQueue<Item*>(WRITER_QUEUE_SIZE);
    controller = new ConsumerController(worker_queue, writer_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* reader = new Reader(n, input_file_name, reader_queue);

    Producer* producer0 = new Producer(reader_queue, worker_queue, transformer);
    Producer* producer1 = new Producer(reader_queue, worker_queue, transformer);
    Producer* producer2 = new Producer(reader_queue, worker_queue, transformer);
    Producer* producer3 = new Producer(reader_queue, worker_queue, transformer);

    Writer* writer = new Writer(n, output_file_name, writer_queue);
```

Figure 13: Creation

第 61 至 67 行則按照原本的設定值去建立 reader queue、worker queue、writer queue 及 controller。其中同樣根據 comsumer 所負責的 queue，拿到 worker queue、writer queue，並做好 threshold 百分比的轉換。

第 72 至 79 行則是根據 reader、prodecer(四個)、writer 所負責的部分，丟入相應的 queue。

```
    // Transfer
    reader->start();

    producer0->start();
    producer1->start();
    producer2->start();
    producer3->start();

    controller->start();

    writer->start();


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

Figure 14: Transferring

接著，啟動每個 thread。啟動後，利用 reader 與 writer 的 `join()` 來確保整個轉換過程都執行完畢，也將結果寫進目標檔案，才讓 main.cpp 繼續執行接下來的程式碼。

```
    delete transformer;
    delete reader_queue;
    delete worker_queue;
    delete writer_queue;
    delete reader;
    delete producer0;
    delete producer1;
    delete producer2;
    delete producer3;
    delete controller;
    delete writer;

    // clock_gettime(CLOCK_MONOTONIC, &end);
    // double elapsed = (end.tv_sec - start.tv_sec) + (end.tv_nsec - start.tv_nsec) / 1e9;
    // std::cout << "execution time: " << elapsed << " seconds\n";
    return 0;
}
```

Figure 15: Deletion

最後刪除所有指標指向的空間。在實驗部分，則計算並印出程式執行的時間。

2 Experiment

在解釋以下實驗前，這裡先付上用測資 00 和測資 01 使用 default 參數 (如 Figure 16所示) 的結果，請參考 Figure 17和18。

```
#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
```

Figure 16: Original Parameters

```
default setting.
Scaling up consumers from 0 to 1
Scaling up consumers from 1 to 2
Scaling down consumers from 2 to 1
execution time: 7.27008 seconds
```

Figure 17: Testcase 00: Default Result

2.1 Different values of CONSUMER CONTROLLER CHECK

首先我們使用測資 00，只改變 consumer cotroller 檢查 worker queue 的時間，Figure 19至 Figure 21分別是 CONSUMER_CONTROLLER_CHECK_PERIOD = 1000000000、1000、100 的測試結果。我們可以發現，檢查週期愈長，完成時間會變長，可以從 Figure 19 line 8 中發現需要花 108.31 seconds 去完成工作。相反地，檢查週期愈短，完成時間會變短，Figure 20 line 18 中可以看到只花了 4.34898 seconds 就完成了。兩者相較於 Figure 17分別有變慢和變快的趨勢。原因在於檢查頻率高的話 worker queue 的內容會時常超過 high threshold，因為 consumer consume work 的速度相較 check period 沒有這麼快，所以 controller 會製造出更多的 consumer 去完成工作。相對地，檢查頻率低時，worker queue 會時常低於 high threshold，因為相較於 check period，consumer consume 的時間較短，因此在有定量消耗才檢查的情況下，controller 就認為沒有必要製造出更多 consumer 來消耗工作。

另外地，我們發現當 CONSUMER_CONTROLLER_CHECK_PERIOD = 100 時，系統沒有達成再更快的完成速度，只有和 CONSUMER_CONTROLLER_CHECK_PERIOD = 1000 時的結果差不多，請參考 Figure 21。我們推測這是由於 producer 處理資料的速度有限，配合 consumer 處理的速度，沒辦法常常讓 worker queue 的工作量常常大於 high threshold = 80。為了驗證假設，我們嘗試在 CONSUMER_CONTROLLER_CHECK_PERIOD = 100 的情形下將 CONSUMER_CONTROLLER_HIGH_THRESHO 降到 50。而成果如 Figure 22所示，成功把時間再降到了 3.72744 seconds。

```
default setting.
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
execution time: 59.7323 seconds
```

Figure 18: Testcase 01: Default Result

```
reader_size: 200
worker_size: 200
writer_size: 4000
low: 20
high: 80
check_period: 100000000
Scaling up consumers from 0 to 1
execution time: 108.31 seconds
```

Figure 19: Experiment 1: Check Time = 1000000000

```
reader_size: 200
worker_size: 200
writer_size: 4000
low: 20
high: 80
check_period: 10000
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
Scaling down consumers from 2 to 1
execution time: 4.34898 seconds
```

Figure 20: Experiment 1: Check Time = 1000

```
reader_size: 200
worker_size: 200
writer_size: 4000
low: 20
high: 80
check_period: 100
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 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
execution time: 4.64042 seconds
```

Figure 21: Experiment 1: Check Time = 100

```
reader_size: 200
worker_size: 200
writer_size: 4000
low: 20
high: 50
check_period: 100
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
Scaling down consumers from 2 to 1
execution time: 3.72744 seconds
```

Figure 22: Experiment 1: Check Time = 100, High Threshold = 50

2.2 Different values of CONSUMER CONTROLLER LOW THRESHOLD

low threshold 會影響到的是 consumer 的數量，因為 worker queue 工作量若低於最低閾值，consumer 會被 cancel。我們分別嘗試將測資 00 的 CONSUMER_CONTROLLER_LOW_THRESHOLD_PERCENTAGE 的值調低和調高，Figure 23和 Figure 24為 CONSUMER_CONTROLLER_LOW_THRESHOLD_PERCENTAGE = 5 和 = 35 的結果。 在其他因素不變的情況下，若最低閾值愈低，代表有更

```
ex02.1.txt
reader_size: 200
worker_size: 200
writer_size: 4000
low: 5
high: 80
check_period: 1000000
Scaling up consumers from 0 to 1
Scaling up consumers from 1 to 2
execution time: 6.64521 seconds
```

Figure 23: Experiment 2: Low Threshold = 5

```
ex02.2.txt
reader_size: 200
worker_size: 200
writer_size: 4000
low: 35
high: 80
check_period: 1000000
Scaling up consumers from 0 to 1
Scaling up consumers from 1 to 2
Scaling down consumers from 2 to 1
execution time: 7.26949 seconds
```

Figure 24: Experiment 2: Low Threshold = 35

多機會讓更多的 consumer 留在場上工作，因此我們能看到在 Figure 23中，相對於原本的配置 (Figure 17)，執行速度變快為 6.64521 seconds，也確實到執行結束前，場上有兩個 consumer 在執行任務。而在 Figure 24中，雖然閾值提高到了 35，但執行速度差不多，且 consumer 增減的結果也和一開始 (Figure 17) 一樣，這代表我們提高的閾值不夠高到讓這些 consumer 可以被 cancel 掉。

2.3 Different values of CONSUMER CONTROLLER HIGH THRESHOLD

High threshold 也會影響到 consumer 的數量，因為 worker queue 工作量若高於最高閾值，consumer 會被 controller 增加。我們一開始假設以上敘述成立，因此我們在其他條件不變的情況下將測資 00 的 `CONSUMER_CONTROLLER_HIGH_THRESHOLD_P` = 50，讓 worker queue 的工作量容易超出最高閾值，以致 controller 會新增更多 consumer。Figure 25是我們的實驗結果，可以看到相較於 default(Figure 17) 條件，consumer 的巔峰數量變為了 3，時間也變快為 5.77078 seconds。但這樣的實驗結果有一個值得注意的地方，就是最後 consumer 的數量沒有下降，而是停留在 3 程式就結束了，我們推測是由於新增了一個 consumer 後，工作快速地在下次 controller 進行 check 之前就被消耗完畢，因此之後程式直接結束，consumer 的數量沒有在執行中被減少。

```
reader_size: 200
worker_size: 200
writer_size: 4000
low: 20
high: 50
check_period: 1000000
Scaling up consumers from 0 to 1
Scaling up consumers from 1 to 2
Scaling up consumers from 2 to 3
execution time: 5.77078 seconds
```

Figure 25: Experiment 3: High Threshold = 50

2.4 Different values of WORKER QUEUE SIZE

Worker queue size 在其他因素不變的情況下也會間接影響到 consumer 的數量，進而影響程式的執行時間。提供的影響在於固定的 task 數量下 worker queue size 會讓工作量比例更容易超出閾值，或更容易低於閾值。Figure 26中我們降低測資 00 的 worker queue size 到剩下 100，可以看到執行結果比原本 Figure 17還要快，consumer 的巔峰數量也增加到了 3。接著我們嘗試再將 worker queue size 增加到 400，等了很久很久都沒有結果，請參考 Figure 27，最後發現是由於測資 00 只有 200 筆資料，在 queue size 為 400，其他參數不變的情況下，工作在如何堆積都無法超過高閾值 80 percent，所以永遠都不會有 consumer 被 controller 創造來工作，程式永遠不會被執行完畢。最後我們決定用測資 01，4000 筆資料對上 4000 的 worker queue size，Figure 28為結果，比起測資 01 原本 default 的執行時間還要長 (Figure 18)，說明了提高 worker queue size 對於 cotroller 來說較不容易達到生成 consumer 的門檻，因而執行時間較長。

```
reader_size: 200
worker_size: 100
writer_size: 4000
low: 20
high: 80
check_period: 1000000
Scaling up consumers from 0 to 1
Scaling up consumers from 1 to 2
Scaling up consumers from 2 to 3
execution time: 4.78903 seconds
```

Figure 26: Experiment 4: Worker Queue Size = 100

```
reader_size: 200
worker_size: 400
writer_size: 4000
low: 20
high: 80
check_period: 1000000
```

Figure 27: Experiment 4: Worker Queue Size = 400

2.5 What happens if WRITER QUEUE SIZE is very small?

我們用測資 00 去測試 writer queue size = 1。這個實驗有一個前提，在知道 writer 只有一個的情況下，writer 的執行時間應該要成為 bottleneck。原本以為測試結果是會讓 writer queue 太塞，consumer 要等待把資料放進 writer queue 中，因此執行時間變得更長，但結果卻和原本的執行時間差不多，如同 Figure 29所示。根據這個情況，我們推測在測資 00 的其他參數是 default 的情況下，consumer 把資料給 writer queue 的速度根本沒有 writer queue 寫進 file 的速度來得快，因此就算 writer queue 變得很小，writer 也還是維持一樣的速度一個一個拿起，甚至有 writer queue 為空的情形發生。consumer 們根本也不需要等待把資料放入 writer queue，因為它們每次工作做完想把資料放進 queue 裡時，writer 早就已經清空 writer queue 在等待新資料了。

2.6 What happens if READER QUEUE SIZE is very small?

在這個讓 reader queue 極端小的實驗，我們也是使用測資 00，除了更動 reader queue size 外，沒有更動其它的參數，結果請參考 Figure 30，執行時間和原本所有參數都是 default 差不多。這裡有一個值得注意的重點，reader 並沒有為了等待 reader queue 有空位可以放資料而讓執行時間變長，代表每一次 reader 要放東西時，queue 早就被 producer 清空，因此無需等

```
reader_size: 200
worker_size: 4000
writer_size: 4000
low: 20
high: 80
check_period: 1000000
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
execution time: 73.4731 seconds
```

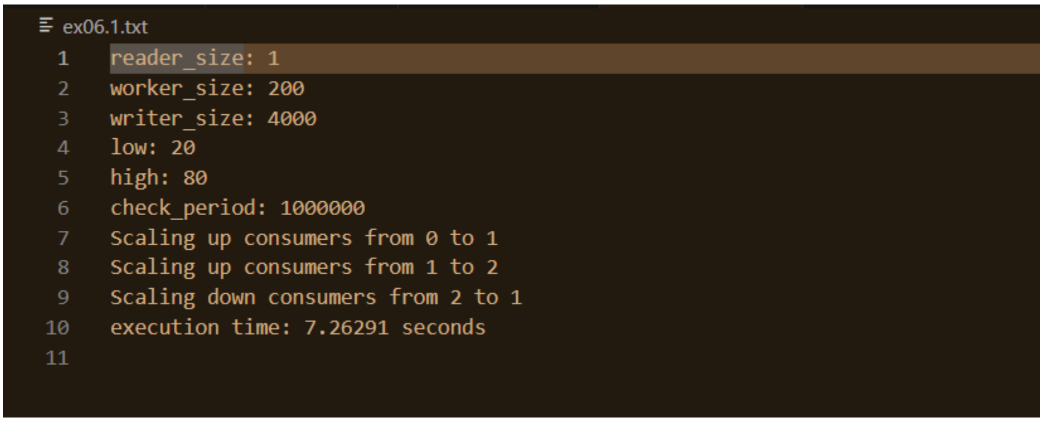
Figure 28: Experiment 4: Worker Queue Size = 4000

ex05.1.txt

```
reader_size: 200
worker_size: 200
writer_size: 1
low: 20
high: 80
check_period: 1000000
Scaling up consumers from 0 to 1
Scaling up consumers from 1 to 2
Scaling down consumers from 2 to 1
execution time: 7.26548 seconds
```

Figure 29: Experiment 5: Writer Queue Size = 1

待。這樣的結果證明 reader 就是一個 bottleneck，不論 producer 多快的把 reader queue 中的工作清空，還是要等待 reader 放新的東西進來才能繼續執行工作。



ex06.1.txt

```
reader_size: 1
worker_size: 200
writer_size: 4000
low: 20
high: 80
check_period: 1000000
Scaling up consumers from 0 to 1
Scaling up consumers from 1 to 2
Scaling down consumers from 2 to 1
execution time: 7.26291 seconds
```

Figure 30: Experiment 6: Reader Queue Size = 1

3 Difficulties

我們遇到的主要困難在於在正確的時機（位置）呼叫 join 並刪除指標所指向的空間。若不是在 main.cpp 呼叫 join 並刪除 reader 及 writer，程式會在中途 segmentation fault，或是遇到最終 writer 無法寫入目標檔案的問題。