Operating System
Pthreads Report

組別：25

陳凱揚：Implement、Experiment、Report

簡佩如：Implement、Experiment、Report

1. Implement

(1) main.cpp

首先在 main 裡面（圖 1-1），我先建立了所有需要的物件，包括 3 個 queue、transformer、reader、writer、4 個 producer、consumerController 等，並傳入對應參數至建構元，其中 controller 會根據 worker_queue 的儲存數量控制 consumer 的數量。接著將這些 thread 都呼叫 start()開始運行，最後需要呼叫 reader 和 writer 的 join()，等待這 2 個 thread 運行完畢，再結束程式。此外，這邊因為 thread 都使用 local 變數，在最後 main 結束後，釋放 local 變數後，便同時會釋放運行中的 thread，但如果這些 thread 是由 new 所開啟的，則一定要將所有東西 delete 掉，否則其他 thread 仍會持續進行。

```
24      // TODO: implements main function
25      TSQueue<Item*> reader_queue(READER_QUEUE_SIZE);
26      TSQueue<Item*> worker_queue(WORKER_QUEUE_SIZE);
27      TSQueue<Item*> writer_queue(WRITER_QUEUE_SIZE);
28      Transformer transformer;
29
30      Reader reader(n, input_file_name, &reader_queue);
31      Writer writer(n, output_file_name, &writer_queue);
32      Producer producer1(&reader_queue, &worker_queue, &transformer);
33      Producer producer2(&reader_queue, &worker_queue, &transformer);
34      Producer producer3(&reader_queue, &worker_queue, &transformer);
35      Producer producer4(&reader_queue, &worker_queue, &transformer);
36      ConsumerController consumerController(
37          &worker_queue, &writer_queue, &transformer,
38          CONSUMER_CONTROLLER_CHECK_PERIOD,
39          CONSUMER_CONTROLLER_LOW_THRESHOLD_PERCENTAGE,
40          CONSUMER_CONTROLLER_HIGH_THRESHOLD_PERCENTAGE);
41
42      reader.start();
43      writer.start();
44      producer1.start();
45      producer2.start();
46      producer3.start();
47      producer4.start();
48      consumerController.start();
49
50      reader.join();
51      writer.join();
```

▲ 圖 1-1

(2) writer.hpp

在 Wirter 中，需要完成 2 個函式（圖 1-2）。在 start()中只要呼叫 pthread_create()，參數放入 Write::process 和 this，開始運行這個 thread 即可；在 process()中與 Reader 的 process()類似，需要不斷從 output_queue 中呼叫 dequeue()取出 1 個 item，並寫進 ofs 中，直到所有 item 寫完就結束。

```
41  void Writer::start() {
42      // TODO: starts a Writer thread
43      pthread_create(&t, 0, Writer::process, (void*)this);
44  }
45
46  void* Writer::process(void* arg) {
47      // TODO: implements the Writer's work
48      Writer *writer = (Writer*)arg;
49
50      while(writer->expected_lines--){
51          Item *item = writer->output_queue->dequeue();
52          writer->ofs << *item;
53          // std::cout << "writer: " << writer->expected_l
54      }
55
56      return nullptr;
57  }
```

▲ 圖 1-2

(3) producer.hpp

在 Producer 中（圖 1-3），start()呼叫 pthread_create()開始運行 thread；而 process()會有一個無限迴圈，不斷從 input_queue 取出 1 個 item，並使用 producer_transform()，傳入 item 的 opcode 和 val 來決定新的 val 值，最後再把 item 放進 worker_queue 裡。

```
35  void Producer::start() {
36      // TODO: starts a Producer thread
37      pthread_create(&t, 0, Producer::process, (void*)this);
38  }
39
40  void* Producer::process(void* arg) {
41      // TODO: implements the Producer's work
42      Producer *producer = (Producer*)arg;
43
44      while(true){
45          Item *item = producer->input_queue->dequeue();
46          item->val = producer->transformer->producer_transform(item->opcode, item->val);
47          producer->worker_queue->enqueue(item);
48      }
49
50      return nullptr;
51  }
```

▲ 圖 1-3

(4) consumer.hpp

Consumer 與 Producer 大致相同（圖 1-4），但多了 cancel()函式，用來給 ConsumerController 呼叫來結束這個 thread，cancel()會將 is_cancel 設為 true，即可跳出 process()的迴圈，結束此 thread。此外，為了防止 thread 在搬運 item 的過程中被直接結束，會先 set canceltype 為 DEFERRED，也就是同步的意思，並在搬運前 set calcelstate 為 DISABLE，直到搬運結束再設回 ENABLE。

```
41  void Consumer::start() {
42      // TODO: starts a Consumer thread
43      pthread_create(&t, 0, Consumer::process, (void*)this);
44  }
45
46  int Consumer::cancel() {
47      // TODO: cancels the consumer thread
48      return is_cancel = true;
49  }
50
51  void* Consumer::process(void* arg) {
52      Consumer* consumer = (Consumer*)arg;
53
54      pthread_setcanceltype(PTHREAD_CANCEL_DEFERRED, nullptr);
55
56      while (!consumer->is_cancel) {
57          pthread_setcancelstate(PTHREAD_CANCEL_DISABLE, nullptr);
58
59          // TODO: implements the Consumer's work
60          Item *item = consumer->worker_queue->dequeue();
61          item->val = consumer->transformer->consumer_transform(item->opcode, item->val);
62          consumer->output_queue->enqueue(item);
63
64          pthread_setcancelstate(PTHREAD_CANCEL_ENABLE, nullptr);
65      }
66
67      delete consumer;
68
69      return nullptr;
70  }
```

▲ 圖 1-4

(5) consumer_controller.hpp

在 ConsumerController 中（圖 1-5），會有一個無限迴圈，週期的不斷檢查 worker_queue 的大小，當大小超過 high_threshold 時，建立一個新的 Consumer 且呼叫 start()開始運行，並放置到 vector consumers 的最後面，同時輸出前後變化；當大小小於 low_threshold 且目前 consumers 的大小大於 1 時（避免 Consumer 數量減為 0），呼叫 cancel()將最近建立的 consumer 結束，並將 consumers pop_back()，同時輸出前後變化。最後會使用 usleep()，並傳入 check_period，代表等待幾微秒後再繼續向下執行，來達到週期性檢查的效果。

```
69  void ConsumerController::start() {
70      // TODO: starts a ConsumerController thread
71      pthread_create(&t, 0, ConsumerController::process, this);
72  }
73
74  void* ConsumerController::process(void* arg) {
75      // TODO: implements the ConsumerController's work
76      ConsumerController *controller = (ConsumerController*)arg;
77
78      while(true){
79          // std::cout << "check" << "\n";
80          int size = controller->worker_queue->get_size();
81          if(size > controller->high_threshold){
82              Consumer* consumer = new Consumer(controller->worker_queue, controller->wri
83              consumer->start();
84              controller->consumers.push_back(consumer);
85              std::cout << "Scaling up consumers from " << controller->consumers.size()-1
86          }
87          else if(size < controller->low_threshold && controller->consumers.size() > 1){
88              controller->consumers.back()->cancel();
89              controller->consumers.pop_back();
90              std::cout << "Scaling down consumers from " << controller->consumers.size()
91          }
92          usleep(controller->check_period);
93      }
94
95      return nullptr;
```

▲ 圖 1-5

(6) ts_queue.hpp

在 TSQueue 的建構元中（圖 1-6），會將 size、head、tail 都設為 0，並 acllocate 大小為 buffer_size 的陣列給 buffer，接著把 mutex、cond_enqueue、cond_dequeue 初始化。而在解構元中，只需 delete 掉分配給 buffer 的空間。

```
51  template <class T>
52  TSQueue<T>::TSQueue(int buffer_size) : buffer_size(buffer_size) {
53      // TODO: implements TSQueue constructor
54      size = head = tail = 0;
55      buffer = new T[buffer_size];
56      pthread_mutex_init(&mutex, nullptr);
57      pthread_cond_init(&cond_enqueue, nullptr);
58      pthread_cond_init(&cond_dequeue, nullptr);
59  }
60
61  template <class T>
62  TSQueue<T>::~TSQueue() {
63      // TODO: implenents TSQueue destructor
64      delete[] buffer;
65  }
```

▲ 圖 1-6

在 enqueue()和 dequeue()中（圖 1-7），這是這次作業最重要的部分，因為會同時有多個 thread 來存取相同的記憶體，而產生 synchronization 的問題，必須使用 mutex、condition variable 來達成 mutual exclusion。而 get_size()只需直接回傳 size，因為 size 的值在 enqueue 和 dequeue 中都會做更新。

我的實作方法為先按照一般的方式，在 enqueue()中，將 item 放入 buffer[tail]裡，並更新 tail 和 size 的值；在 dequeue()中，將 buffer[head]取出，並更新 head 和 size 的值。接著在 enqueue()中，buffer 滿了時，就呼叫 cond_wait()，參數為 cond_enqueue，將目前這個 thread 放入等待中，並在完成 enqueue 後呼叫 cond_signal()，參數為 cond_dequeue，告訴等待 dequeue 的 thread 可以開始 dequeue，若沒有 thread 在等待中，則沒有影響，最後使用 mutex 的 lock 和 unlock 把整個函式包起來。而在 dequeue()中與 enqueue()相似，首先在 buffer 為空時，呼叫 cond_wait()進入 cond_dequeue 中等待，並在完成 dequeue 後呼叫 cond_signal()，告訴等待 enqueue 的 thread 可以開始 enqueue，最後同樣將整個函式以 mutex 的 lock 和 unlock 包起來。

```
68  template <class T>
69  void TSQueue<T>::enqueue(T item) {
70      // TODO: enqueues an element to the end of the queue
71      pthread_mutex_lock(&mutex);
72      while(size == buffer_size){
73          pthread_cond_wait(&cond_enqueue, &mutex);
74      }
75      buffer[tail] = item;
76      tail = (tail+1)%buffer_size;
77      size++;
78      pthread_cond_signal(&cond_dequeue);
79      pthread_mutex_unlock(&mutex);
80  }
81
82  template <class T>
83  T TSQueue<T>::dequeue() {
84      // TODO: dequeues the first element of the queue
85      pthread_mutex_lock(&mutex);
86      while(size == 0){
87          pthread_cond_wait(&cond_dequeue, &mutex);
88      }
89      T item = buffer[head];
90      head = (head+1)%buffer_size;
91      size--;
92      pthread_cond_signal(&cond_enqueue);
93      pthread_mutex_unlock(&mutex);
94      return item;
95  }
96
97  template <class T>
98  int TSQueue<T>::get_size() {
99      // TODO: returns the size of the queue
100     return size;
101 }
```

▲ 圖 1-7

2. Experiment

(1) Different values of CONSUMER_CONTROLLER_CHECK_PERIOD

原本 CONSUMER_CONTROLLER_CHECK_PERIOD 為 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 down consumers from 10 to 9
```

由 1000000 開始向上乘兩倍去觀察變化，2000000、4000000、… 以此類推，發現 scaling up 的次數會減半，直到剩下一次。

```
CONSUMER_CONTROLLER_CHECK_PERIOD2000000
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
```

```
CONSUMER_CONTROLLER_CHECK_PERIOD4000000
Scaling up consumers from 0 to 1
Scaling up consumers from 1 to 2
Scaling up consumers from 2 to 3
```

```
CONSUMER_CONTROLLER_CHECK_PERIOD8000000
Scaling up consumers from 0 to 1
Scaling up consumers from 1 to 2
```

```
CONSUMER_CONTROLLER_CHECK_PERIOD16000000
Scaling up consumers from 0 to 1
```

由 1000000 開始向下除兩倍觀察變化，一開始會在 scaling up 5~6 跟 6~7 之間跑。

```
CONSUMER_CONTROLLER_CHECK_PERIOD500000
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
```

```
CONSUMER_CONTROLLER_CHECK_PERIOD250000
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
```

```
CONSUMER_CONTROLLER_CHECK_PERIOD125000
Scaling up consumers from 0 to 1
Scaling up consumers from 1 to 2
Scaling up consumers from 2 to 3
Scaling up consumers from 3 to 4
Scaling up consumers from 4 to 5
Scaling up consumers from 5 to 6
Scaling up consumers from 6 to 7
Scaling down consumers from 7 to 6
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
```

```
CONSUMER_CONTROLLER_CHECK_PERIOD62500
Scaling up consumers from 0 to 1
Scaling up consumers from 1 to 2
Scaling up consumers from 2 to 3
Scaling up consumers from 3 to 4
Scaling up consumers from 4 to 5
Scaling up consumers from 5 to 6
Scaling up consumers from 6 to 7
Scaling down consumers from 7 to 6
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
```

```
CONSUMER_CONTROLLER_CHECK_PERIOD31250
Scaling up consumers from 0 to 1
Scaling up consumers from 1 to 2
Scaling up consumers from 2 to 3
Scaling up consumers from 3 to 4
Scaling up consumers from 4 to 5
Scaling up consumers from 5 to 6
Scaling up consumers from 6 to 7
Scaling down consumers from 7 to 6
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
```

```
CONSUMER_CONTROLLER_CHECK_PERIOD15625
Scaling up consumers from 0 to 1
Scaling up consumers from 1 to 2
Scaling up consumers from 2 to 3
Scaling up consumers from 3 to 4
Scaling up consumers from 4 to 5
Scaling up consumers from 5 to 6
Scaling up consumers from 6 to 7
Scaling down consumers from 7 to 6
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
```

```
CONSUMER_CONTROLLER_CHECK_PERIOD7811
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
```

```
CONSUMER_CONTROLLER_CHECK_PERIOD7812
Scaling up consumers from 0 to 1
Scaling up consumers from 1 to 2
Scaling up consumers from 2 to 3
Scaling up consumers from 3 to 4
Scaling up consumers from 4 to 5
Scaling up consumers from 5 to 6
Scaling up consumers from 6 to 7
Scaling down consumers from 7 to 6
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
```

(2) Different values of CONSUMER_CONTROLLER_LOW_THRESHOLD_PERCENTAGE and CONSUMER_CONTROLLER_HIGH_THRESHOLD_PERCENTAGE

固定 CONSUMER_CONTROLLER_LOW_THRESHOLD_PERCENTAGE，則 scaling 的數量會隨著 CONSUMER_CONTROLLER_HIGH_THRESHOLD_PERCENTAGE 上升而下降。

```
CONSUMER_CONTROLLER_LOW_THRESHOLD_PERCENTAGE20
CONSUMER_CONTROLLER_HIGH_THRESHOLD_PERCENTAGE90
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
```

```
CONSUMER_CONTROLLER_LOW_THRESHOLD_PERCENTAGE20
CONSUMER_CONTROLLER_HIGH_THRESHOLD_PERCENTAGE100
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
```

```
CONSUMER_CONTROLLER_LOW_THRESHOLD_PERCENTAGE20
CONSUMER_CONTROLLER_HIGH_THRESHOLD_PERCENTAGE110
Scaling up consumers from 0 to 1
Scaling up consumers from 1 to 2
Scaling up consumers from 2 to 3
Scaling up consumers from 3 to 4
Scaling up consumers from 4 to 5
Scaling up consumers from 5 to 6
Scaling up consumers from 6 to 7
Scaling up consumers from 7 to 8
Scaling down consumers from 8 to 7
```

```
CONSUMER_CONTROLLER_LOW_THRESHOLD_PERCENTAGE20
CONSUMER_CONTROLLER_HIGH_THRESHOLD_PERCENTAGE120
Scaling up consumers from 0 to 1
Scaling up consumers from 1 to 2
Scaling up consumers from 2 to 3
Scaling up consumers from 3 to 4
Scaling up consumers from 4 to 5
Scaling up consumers from 5 to 6
Scaling up consumers from 6 to 7
Scaling down consumers from 7 to 6
```

固定 CONSUMER_CONTROLLER_HIGH_THRESHOLD_PERCENTAGE，

當 CONSUMER_CONTROLLER_LOW_THRESHOLD_PERCENTAGE 小於 17 就不會 scaling down 了。

```
CONSUMER_CONTROLLER_LOW_THRESHOLD_PERCENTAGE17
CONSUMER_CONTROLLER_HIGH_THRESHOLD_PERCENTAGE80
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
```

```
CONSUMER_CONTROLLER_LOW_THRESHOLD_PERCENTAGE16
CONSUMER_CONTROLLER_HIGH_THRESHOLD_PERCENTAGE80
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
```

固定 CONSUMER_CONTROLLER_HIGH_THRESHOLD_PERCENTAGE，Scaling down 的次數會隨著 CONSUMER_CONTROLLER_LOW_THRESHOLD_PERCENTAGE 增加而增加。

```
CONSUMER_CONTROLLER_LOW_THRESHOLD_PERCENTAGE30
CONSUMER_CONTROLLER_HIGH_THRESHOLD_PERCENTAGE80
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
```

```
CONSUMER_CONTROLLER_LOW_THRESHOLD_PERCENTAGE60
CONSUMER_CONTROLLER_HIGH_THRESHOLD_PERCENTAGE80
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
```

```
CONSUMER_CONTROLLER_LOW_THRESHOLD_PERCENTAGE80
CONSUMER_CONTROLLER_HIGH_THRESHOLD_PERCENTAGE80
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
```

(3) Different values of WORKER_QUEUE_SIZE

實驗方法是 200*2^n, n = 2,3,4,…，會發現 scaling up 的次數基本上是固定不動的，而 scaling down 的次數會在 1~2 次之間跳，主要是以 1 次為主。

```
WORKER_QUEUE_SIZE1600
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
```

```
WORKER_QUEUE_SIZE12800
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
```

基本上 WORKER_QUEUE_SIZE 小於 80 就跑不動了，所以不能太小。

```
WORKER_QUEUE_SIZE100
Scaling up consumers from 0 to 1
Scaling up consumers from 1 to 2
Scaling up consumers from 2 to 3
Scaling up consumers from 3 to 4
Scaling up consumers from 4 to 5
Scaling up consumers from 5 to 6
Scaling up consumers from 6 to 7
Scaling up consumers from 7 to 8
Scaling down consumers from 8 to 7
```

```
WORKER_QUEUE_SIZE90
Scaling up consumers from 0 to 1
Scaling up consumers from 1 to 2
Scaling up consumers from 2 to 3
Scaling up consumers from 3 to 4
Scaling up consumers from 4 to 5
Scaling up consumers from 5 to 6
Scaling up consumers from 6 to 7
Scaling down consumers from 7 to 6
```

(4) What happens if WRITER_QUEUE_SIZE is very small?

test00 只有在 size 等於 1 的時候會多一次 scaling down。因為 test00 效果不明顯，因此使用 test01 去測試，整體來說是 scaling up 跟 scaling down 的次數會減少。

```
  WRITER_QUEUE_SIZE 1
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
```

(5) What happens if READER_QUEUE_SIZE is very small?

在 test00 中，不管 READER_QUEUE_SIZE 是 4000 還是 1，scaling 的次數都是一樣的。在 test01 中，READER_QUEUE_SIZE 是 4000 或 1 也沒有顯著影響，只有 4000 的 scaling 次數稍微多一點。

```
  READER_QUEUE_SIZE 1
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
```

3. Difficulties and Feedback

這次作業中由於都有 TODO 的引導，也有很多相似且寫好的程式，讓整個 implementation 簡單不少，唯一比較遇到問題的只有週期性的檢查 consumer 的地方，一開始使用 clock()來得到時間檢查，但一直都比預期的快，後來改採 usleep()就順利解決問題了。此外，這也是第一次實作使用 Pthread 的函式庫，增加了不少對 synchronization 解決方法的熟悉度。