

## **CS342301: Operating System**

### **MP5: Pthread**

#### **Team member & contribution:**

##### **I. 江佩霖 111062118**

- i.** Trace code
- ii.** Implement function
- iii.** 測試, Debug

##### **II. 陳庭竣 111020025**

- i.** Trace code
- ii.** Implement function
- iii.** 測試, Debug

## I. Implement function explain

### 1. TODO list

#### ➤ Consumer\_controller.hpp:

```
void ConsumerController::start();
// TODO: starts a ConsumerController thread
void *ConsumerController::process(void *arg);
// TODO: implements the ConsumerController's work
```

#### ➤ Consumer.hpp

```
void Consumer::start();
// TODO: starts a Consumer thread
int Consumer::cancel();
// TODO: cancels the consumer thread
void *Consumer::process(void *arg);
// TODO: implements the Consumer's work
```

#### ➤ Main.cpp

```
// TODO: implements main function
```

#### ➤ Producer.hpp

```
void Producer::start()
// TODO: starts a Producer thread
void *Producer::process(void *arg)
// TODO: implements the Producer's work
```

#### ➤ Ts\_queue.hpp

```
template <class T>
TSQueue<T>::TSQueue(int buffer_size) : buffer_size(buffer_size)
// TODO: implements TSQueue constructor
template <class T>
TSQueue<T>::~~TSQueue()
// TODO: implenents TSQueue destructor
template <class T>
void TSQueue<T>::enqueue(T item)
// TODO: enqueues an element to the end of the queue
template <class T>
T TSQueue<T>::dequeue()
// TODO: dequeues the first element of the queue
template <class T>
int TSQueue<T>::get_size()
// TODO: returns the size of the queue
```

#### ➤ Writer.hpp

```
void Writer::start()
// TODO: starts a Writer thread
```

```
void *Writer::process(void *arg)
// TODO: implements the Writer's work
```

## 2. Implement

### i. TS\_Queue.hpp

說明:

- 實作內容同註解
- 有額外實作一個 function: get\_buffer\_size()供 consumer\_controller 使用(return buffer\_size)

```
51 template <class T>
52 TSQueue<T>::TSQueue(int buffer_size) : buffer_size(buffer_size)
53 {
54     // TODO: implements TSQueue constructor
55     // initialize mutex
56     pthread_mutex_init(&mutex, NULL);
57
58     pthread_mutex_lock(&mutex); // For protect init: enter critical section
59     /*****critical section*****/
60     // initialize members
61     buffer = new T[buffer_size];
62     size = 0;
63     head = 0;
64     tail = -1;
65     // initialize conditional variables
66     pthread_cond_init(&cond_enqueue, NULL);
67     pthread_cond_init(&cond_dequeue, NULL);
68     /*****critical section*****/
69     pthread_mutex_unlock(&mutex); // leave critical section
70 }
```

```
73 template <class T>
74 TSQueue<T>::~TSQueue()
75 {
76     // TODO: implements TSQueue destructor
77     pthread_mutex_lock(&mutex); // For protect destory: enter critical section
78     /*****critical section*****/
79     // free members
80     delete buffer;
81
82     // destroy condition variables
83     pthread_cond_destroy(&cond_enqueue);
84     pthread_cond_destroy(&cond_dequeue);
85     /*****critical section*****/
86     pthread_mutex_unlock(&mutex);
87
88     // delete the mutex
89     pthread_mutex_destroy(&mutex);
90 }
```

```

92 template <class T>
93 void TSQueue<T>::enqueue(T item)
94 {
95     // TODO: enqueues an element to the end of the queue
96     pthread_mutex_lock(&mutex); // To protect queue: enter critical section
97     /*****critical section*****/
98     /* check for the free place in queue:
99     | | let cond_enqueue be the condition variable to lock TSQueue::enqueue */
100     while (size == buffer_size)
101     {
102         // if no => let it wait
103         pthread_cond_wait(&cond_enqueue, &mutex);
104     }
105
106     // if there still has place for consumer => put it into queue
107     tail = (tail + 1) % buffer_size;
108     buffer[tail] = item;
109     size++;
110
111     /* After we done a enqueue, queue have at least one element: notify dequeue*/
112     pthread_cond_signal(&cond_dequeue);
113     /*****critical section*****/
114
115     pthread_mutex_unlock(&mutex); // leave critical section
116 }

```

```

118 template <class T>
119 T TSQueue<T>::dequeue()
120 {
121     // TODO: dequeues the first element of the queue
122     pthread_mutex_lock(&mutex); // To protect queue: enter critical section
123     /*****critical section*****/
124     /* check if the queue already has item:
125     | | let cond_enqueue be the condition variable to lock TSQueue::dequeue */
126     while (size == 0)
127     {
128         // if no => let it wait
129         pthread_cond_wait(&cond_dequeue, &mutex);
130     }
131
132     // if there exist at least one item in the queue=> do dequeue
133     T val = buffer[head];
134     head = (head + 1) % buffer_size;
135     size--;
136
137     /* After we done a dequeue, queue have at least one empty place: notify enqueue*/
138     pthread_cond_signal(&cond_enqueue);
139     /*****critical section*****/
140     pthread_mutex_unlock(&mutex); // leave critical section
141     return val;
142 }

```

```

144 template <class T>
145 int TSQueue<T>::get_size()
146 {
147     // TODO: returns the size of the queue
148     // just return the val, no need to get into critical section
149     return size;
150 }
151
152 template <class T>
153 int TSQueue<T>::get_buffer_size()
154 {
155     // just return the val, no need to get into critical section
156     return buffer_size;
157 }

```

## ii. Writer.hpp

說明:

- 實作內容同註解
- Writer::process()結束後會回傳空指標

```

45 void Writer::start()
46 {
47     // TODO: starts a Writer thread
48     // Create a new thread that runs the Writer::process method
49     pthread_create(&this->t, 0, Writer::process, (void *)this);
50 }
51
52 // Static method: implements the logic executed by the writer thread
53 void *Writer::process(void *arg)
54 {
55     // TODO: implements the Writer's work
56     // Cast the argument to a Writer object
57     Writer *writer = (Writer *)arg;
58
59     // Loop until the expected number of lines is written
60     while (writer->expected_lines--)
61     {
62         // Dequeue an item from the output queue (blocking operation if queue is empty)
63         Item *item = writer->output_queue->dequeue();
64         // Write the item's content to the output file using the ofstream object
65         writer->ofs << *item;
66     }
67
68     // Exit the thread
69     return nullptr;
70 }

```

## iii. Producer.hpp

說明:

- 實作內容同註解

```

38 void Producer::start()
39 {
40     // TODO: starts a Producer thread
41     // Creates a new thread and starts executing the process method
42     pthread_create(&this->t, 0, Producer::process, (void *)this);
43 }
44
45 void *Producer::process(void *arg)
46 {
47     // TODO: implements the Producer's work
48     // Casts the argument to a Producer object
49     Producer *producer = (Producer *)arg;
50
51     while (1) // Infinite loop that runs the producer thread
52     {
53         // Check if there are items in the input queue
54         if (producer->input_queue->get_size() > 0)
55         {
56             // Dequeues an item from the input queue for processing (need to be deleted)
57             Item *transform_item = producer->input_queue->dequeue();
58             // Uses the transformer to process the item
59             unsigned long long int val = producer->transformer->producer_transform(transform_item->opcode, transform_item->val);
60             Item *new_item = new Item(transform_item->key, val, transform_item->opcode); // new Item
61             // Enqueues the new item into the worker queue for further processing
62             producer->worker_queue->enqueue(new_item);
63             //! important for heap memory management
64             // Deletes the original item as it's no longer needed
65             delete transform_item;
66         }
67     }
68     // Returns null when the thread finishes
69     return nullptr;
70 }

```

#### iv. Consumer.hpp

説明:

➤ 実作内容同註解

```

44 void Consumer::start()
45 {
46     // TODO: starts a Consumer thread
47     // Creates a new thread and starts executing the process method
48     pthread_create(&this->t, 0, Consumer::process, this);
49 }
50
51 int Consumer::cancel()
52 {
53     // TODO: cancels the consumer thread
54     // Sets the cancellation flag to true: notify "static Consumer::proceee" to end the infinity loop and delete Consumer
55     is_cancel = true;
56     return pthread_cancel(this->t);
57 }
58
59 // very same as static Producer::process
60 void *Consumer::process(void *arg)
61 {
62     Consumer *consumer = (Consumer *)arg;
63     pthread_setcanceltype(PTHREAD_CANCEL_DEFERRED, nullptr);
64     while (!consumer->is_cancel)
65     {
66         pthread_setcancelstate(PTHREAD_CANCEL_DISABLE, nullptr);
67
68         // TODO: implements the Consumer's work
69         // take the item form worker_queue
70         if (consumer->worker_queue->get_size() > 0)
71         {
72             // the same as producer::process, dequeue an item form queue
73             Item *transform_item = consumer->worker_queue->dequeue();
74             // new item: use "Transformer::consumer_transform" transfer data
75             unsigned Long Long int val = consumer->transformer->consumer_transform(transform_item->opcode, transform_item->val);
76             Item *new_item = new Item(transform_item->key, val, transform_item->opcode);
77             // put the new item into "output_queue"
78             consumer->output_queue->enqueue(new_item);
79
80             // delete the original item
81             delete transform_item;
82         }
83         pthread_setcancelstate(PTHREAD_CANCEL_ENABLE, nullptr);
84     }
85     delete consumer;
86     return nullptr;
87 }

```

#### v. Consumer\_controller.hpp

説明:

➤ 実作内容同註解

```

69 void ConsumerController::start()
70 {
71     // TODO: starts a ConsumerController thread
72     // Creates a new thread that runs the process method
73     pthread_create(&this->t, 0, ConsumerController::process, this);
74 }

```

```

76 void *ConsumerController::process(void *arg)
77 {
78     // TODO: implements the ConsumerController's work
79     // Casts the argument to a ConsumerController object
80     ConsumerController *controller = (ConsumerController *)arg;
81     while (1) // Infinite loop that keeps checking the worker queue size and scaling consumers up/down
82     {
83         // Calculates the proportion of items in the worker queue relative to its buffer size
84         double worker_proportion = (double)controller->worker_queue->get_size() / controller->worker_queue->get_buffer_size();
85         // If the worker queue is more than the high threshold
86         if (worker_proportion > (double)controller->high_threshold / 100)
87         {
88             // Creates a new consumer to handle more items and starts it
89             Consumer *new_consumer = new Consumer(controller->worker_queue, controller->writer_queue, controller->transformer);
90             new_consumer->start();
91
92             // Adds the new consumer to the consumers vector
93             controller->consumers.push_back(new_consumer);
94             std::cout << "Scaling up consumers from " << controller->consumers.size() - 1 << " to " << controller->consumers.size() << "\n";
95         }
96         // If the worker queue is less than the low threshold and there are more than 1 consumer (scale down)
97         else if (worker_proportion < (double)controller->low_threshold / 100 && controller->consumers.size() > 1)
98         {
99             // Removes and cancels the last consumer from the vector
100             Consumer *delete_consumer = controller->consumers[controller->consumers.size() - 1];
101             delete_consumer->cancel();
102             controller->consumers.pop_back();
103             std::cout << "Scaling up consumers from " << controller->consumers.size() + 1 << " to " << controller->consumers.size() << "\n";
104         }
105         // Pauses for the specified check period before checking again
106         usleep(controller->check_period);
107     }
108     // Returns nullptr when the thread finishes
109     return nullptr;
110 }

```

## vi. Main.cpp

説明:

➤ 実作内容同註解

```

17 int main(int argc, char **argv)
18 {
19     assert(argc == 4);
20
21     int n = atoi(argv[1]);
22     std::string input_file_name(argv[2]);
23     std::string output_file_name(argv[3]);
24
25     // TODO: implements main function
26     TQueue<Item *> *input_queue = new TQueue<Item *>(READER_QUEUE_SIZE);
27     TQueue<Item *> *woker_queue = new TQueue<Item *>(WORKER_QUEUE_SIZE);
28     TQueue<Item *> *output_queue = new TQueue<Item *>(WRITER_QUEUE_SIZE);
29
30     // Start the threads for reading, writing, producing, and controlling consumers
31     Transformer *transformer = new Transformer();
32     Reader *reader = new Reader(n, input_file_name, input_queue);
33     Writer *writer = new Writer(n, output_file_name, output_queue);
34     Producer *p1 = new Producer(input_queue, woker_queue, transformer);
35     Producer *p2 = new Producer(input_queue, woker_queue, transformer);
36     Producer *p3 = new Producer(input_queue, woker_queue, transformer);
37     Producer *p4 = new Producer(input_queue, woker_queue, transformer);
38     ConsumerController *controller = new ConsumerController(
39         woker_queue, output_queue, transformer,
40         CONSUMER_CONTROLLER_CHECK_PERIOD,
41         CONSUMER_CONTROLLER_LOW_THRESHOLD_PERCENTAGE,
42         CONSUMER_CONTROLLER_HIGH_THRESHOLD_PERCENTAGE);
43
44     // Start all the threads
45     reader->start();
46     writer->start();
47     controller->start();
48     p1->start();
49     p2->start();
50     p3->start();
51     p4->start();

```

```
53 // Wait for the reader and writer threads to finish (join them to the main thread)
54 reader->join();
55 writer->join();
56
57 // Once reading and writing are complete, clean up dynamically allocated memory
58 delete p1;
59 delete p2;
60 delete p3;
61 delete p4;
62 delete controller;
63 delete reader;
64 delete writer;
65 delete transformer;
66 delete input_queue;
67 delete woker_queue;
68 delete output_queue;
69
70 // End the program
71 return 0;
72 }
```



## II. Experiments / Result

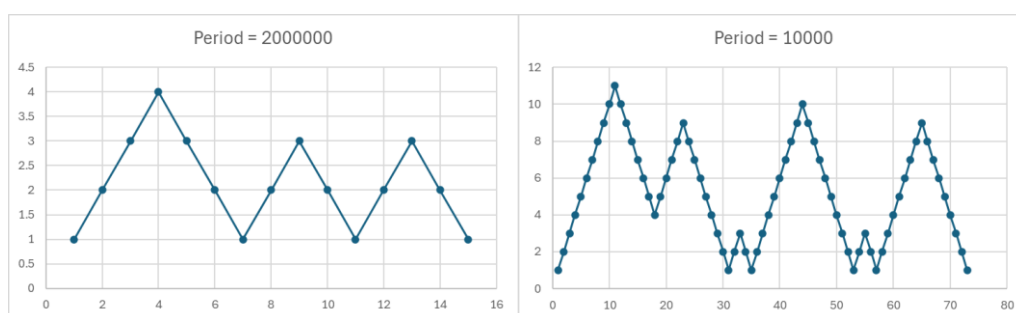
- 為了測試，在 main.cpp 中引入標頭檔 `#include <chrono>`，並在 start process 和呼叫首行刪除程序之間計算起始-結束時間(此程式碼並無列在實作部分，僅在 main 做更動)。
- 主要使用 test case 01 測試。
- 一次測試僅更動題目要求的變數，其餘變數均為預設值不做更動(表格會列出當前更動項目)。
- 所有測試均計算三次(含)以上取平均值。
- 由於測試過程中偶爾會有極端值出現 (Ex. 200000ms/ 5000ms)，尚且無法斷定是否為連線或是其他問題，經過多次測試或是等待一段時間後再測試來確認當下是否為極端值出現，若非此狀況則會記錄於測試資料中。
- 灰色底的資料是原始資料以及其 time cost。
- 詳細測試資料存在 NTHU-OS-PTHREADS/TESTDATA 中(test\_data\_1-5.md)，亦附在 report 最後。

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

- 測試資料

CONSUMER_CONTROLLER_CHECK_PERIOD	Time Cost(ms)
10000	55762
100000	55846
1000000 (origin value)	79286
1500000	85480
2000000	166103

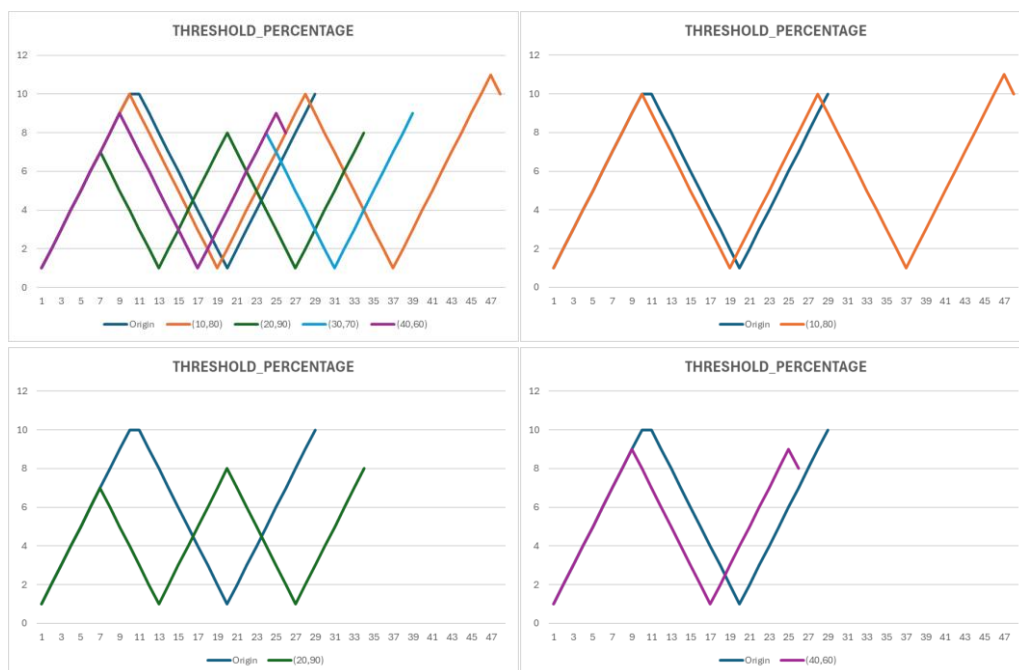
- 總耗時與 check period 大致上呈現正相關，推測是檢查週期變常導致來不及分配適當數量的 consumer 導致。
- 另外，從測試資料可觀察到，check period 越大，其改變 consumer 的次數、幅度也有所差異(下圖是取[period,test] = [2000000, 3], [10000, 2]做圖)



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

- 測試資料

CONSUMER_CONTROLLER_LOW_THRESHOLD_PERCENTAGE	CONSUMER_CONTROLLER_HIGH_THRESHOLD_PERCENTAGE	Time Cost(ms)
10	80	74469
20	80	79286
20	90	79746
30	70	62531
40	60	65042



- low value 降低，震幅變大；high value 升高，震幅變低。
- 比對 high-low 的差值，縮小後震幅也會降低。
- 另外，time cost 似乎和這兩個 value 的關係不大(目前所有測試資料都在 50000-70000 左右浮動)；額外多測試的幾筆資料顯示，high value 和 time cost 似乎有正相關，推測由於 high value 升高，controller 越不容易創建新的 consumer，故執行時間延長。

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

- 測試資料

WORKER_QUEUE_SIZE	Time Cost(ms)
50	117598
100	132217
200	79286
300	103612
400	94485



- 可以發現 WORKER\_QUEUE\_SIZE 和 Time Usage 有點負相關，WORKER\_QUEUE\_SIZE 越大，

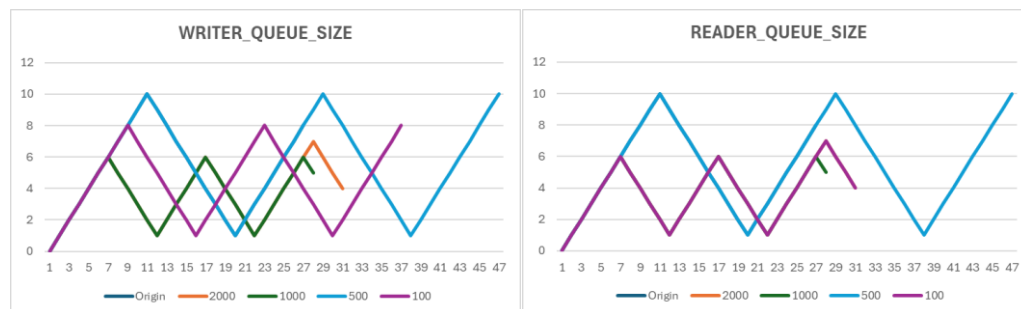
Time Usage 越小(但下降情況不明顯，很可能是因為 controller 是根據百分比調節 consumer 數量)

- 折線圖在 WORKER\_QUEUE\_SIZE 較小時，會有較多震動情況，推測是因為 queue 容量變小導致更新狀況較頻繁。

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

- 測試資料

WRITER_QUEUE_SIZE	Time Cost(ms)
4000	79286
2000	94279
1000	106236
500	86644
100	78152

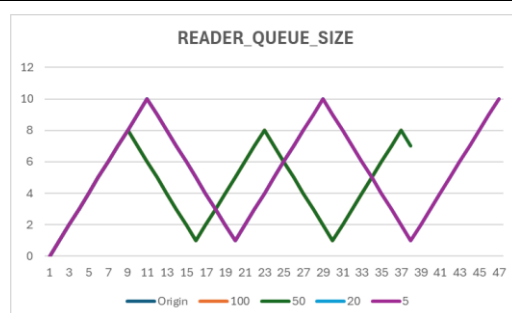


- WRITE\_QUEUE\_SIZE 和 Time Usage 的關係大致並無規律
- 目前結果顯示，WRITER\_QUEUE\_SIZE 對 Time Usage 與 CONSUMER SIZE CHANGE 並沒有太大影響，從製圖結果可得出資料大多重疊且震幅規律相似。

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

- 測試資料

READER_QUEUE_SIZE	Time Cost(ms)
200	79286
100	97325
50	67182
20	65866
5	60987



- 目前結果顯示，READER\_QUEUE\_SIZE 對 Time Usage 與 CONSUMER SIZE CHANGE 並沒有太大影響(有些微正相關但改變不大)，從製圖結果可得出資料大多重疊且震幅規律相似。

### III. Difficult / Feedback

- 這次的作業新增了實驗部分，讓我覺得非常有趣和新奇，挑戰性也更大了。雖然課程材料提供了大致的做法，但在實驗部分的說明上並沒有給出很嚴謹的定義，使得我在進行實驗時感到一些困惑和不確定。這種不確定性讓我有些害怕，擔心自己的理解不夠深入或操作不夠精確。此外，在撰寫報告時，可能未能完全清晰地表達自己的過程和結果。這次作業讓我學到了不少，儘管有些挑戰，但也激發了我更多的學習動力。

### IV. Appendix

- <https://drive.google.com/drive/folders/1F902vWeZO4vdorWPPbHZ8jMRQ-DPD9z6?usp=sharing>