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: implenents 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  }
```

```cpp
template <class T>
void TSQueue<T>::enqueue(T item)
{
  // TODO: enqueues an element to the end of the queue
  pthread_mutex_lock(&mutex); // To protect queue: enter critical section
  /*******************critical section*********************/
  /* check for the free place in queue:
     | let cond_enqueue be the condition variable to lock TSQueue::enqueue */
  while (size == buffer_size)
  {
    // if no => let it wait
    pthread_cond_wait(&cond_enqueue, &mutex);
  }

  // if there still has place for consumer => put it into queue
  tail = (tail + 1) % buffer_size;
  buffer[tail] = item;
  size++;

  /* After we done a enqueue, queue have at least one element: notify dequeue*/
  pthread_cond_signal(&cond_dequeue);
  /*******************critical section*********************/

  pthread_mutex_unlock(&mutex); // leave critical section
}
```

```cpp
template <class T>
T TSQueue<T>::dequeue()
{
  // TODO: dequeues the first element of the queue
  pthread_mutex_lock(&mutex); // To protect queue: enter critical section
  /*******************critical section*********************/
  /* check if the queue already has item:
     | let cond_enqueue be the condition variable to lock TSQueue::dequeue */
  while (size == 0)
  {
    // if no => let it wait
    pthread_cond_wait(&cond_dequeue, &mutex);
  }

  // if there exist at least one item in the queue=> do dequeue
  T val = buffer[head];
  head = (head + 1) % buffer_size;
  size--;

  /* After we done a dequeue, queue have at least one empty place: notify enqueue*/
  pthread_cond_signal(&cond_enqueue);
  /*******************critical section*********************/
  pthread_mutex_unlock(&mutex); // leave critical section
  return val;
}
```

```cpp
template <class T>
int TSQueue<T>::get_size()
{
  // TODO: returns the size of the queue
  // just return the val, no need to get into critical section
  return size;
}

template <class T>
int TSQueue<T>::get_buffer_size()
{
  // just return the val, no need to get into critical section
  return buffer_size;
}
```

ii. Writer.hpp

說明:

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

```cpp
void Writer::start()
{
  // TODO: starts a Writer thread
  // Create a new thread that runs the Writer::process method
  pthread_create(&this->t, 0, Writer::process, (void *)this);
}

// Static method: implements the logic executed by the writer thread
void *Writer::process(void *arg)
{
  // TODO: implements the Writer's work
  // Cast the argument to a Writer object
  Writer *writer = (Writer *)arg;

  // Loop until the expected number of lines is written
  while (writer->expected_lines--)
  {
    // Dequeue an item from the output queue (blocking operation if queue is empty)
    Item *item = writer->output_queue->dequeue();
    // Write the item's content to the output file using the ofstream object
    writer->ofs << *item;
  }

  // Exit the thread
  return nullptr;
}
```

iii. Producer.hpp

說明:

- 實作內容同註解

```cpp
void Producer::start()
{
  // TODO: starts a Producer thread
  // Creates a new thread and starts executing the process method
  pthread_create(&this->t, 0, Producer::process, (void *)this);
}
```

```cpp
void *Producer::process(void *arg)
{
  // TODO: implements the Producer's work
  // Casts the argument to a Producer object
  Producer *producer = (Producer *)arg;

  while (1) // Infinite loop that runs the producer thread
  {
    // Check if there are items in the input queue
    if (producer->input_queue->get_size() > 0)
    {
      // Dequeues an item from the input queue for processing(need to be deleted)
      Item *transform_item = producer->input_queue->dequeue();
      // Uses the transformer to process the item
      unsigned long long int val = producer->transformer->producer_transform(transform_item->opcode, transform_item->val);
      Item *new_item = new Item(transform_item->key, val, transform_item->opcode); // new Item
      // Enqueues the new item into the worker queue for further processing
      producer->worker_queue->enqueue(new_item);
      //! important for heap memory management
      // Deletes the original item as it's no longer needed
      delete transform_item;
    }
  }
  // Returns null when the thread finishes
  return nullptr;
}
```

iv. Concumer.hpp

說明:

- 實作內容同註解

```cpp
void Consumer::start()
{
  // TODO: starts a Consumer thread
  // Creates a new thread and starts executing the process method
  pthread_create(&this->t, 0, Consumer::process, this);
}

int Consumer::cancel()
{
  // TODO: cancels the consumer thread
  // Sets the cancellation flag to true: notify "static Consumer::proceee" to end the infinity loop and delete Consumer
  is_cancel = true;
  return pthread_cancel(this->t);
}
```

```cpp
// very same as static Producer::process
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 the item form worker_queue
    if (consumer->worker_queue->get_size() > 0)
    {
      // the same as producer::process, dequeue an item form queue
      Item *transform_item = consumer->worker_queue->dequeue();
      // new item: use "Transformer::consumer_transform" transfer data
      unsigned long long int val = consumer->transformer->consumer_transform(transform_item->opcode, transform_item->val);
      Item *new_item = new Item(transform_item->key, val, transform_item->opcode);
      // put the new item into "output_queue"
      consumer->output_queue->enqueue(new_item);

      // delete the original item
      delete transform_item;
    }
    pthread_setcancelstate(PTHREAD_CANCEL_ENABLE, nullptr);
  }
  delete consumer;
  return nullptr;
}
```

v. Consumer_controller.hpp

說明:

- 實作內容同註解

```cpp
void ConsumerController::start()
{
  // TODO: starts a ConsumerController thread
  // Creates a new thread that runs the process method
  pthread_create(&this->t, 0, ConsumerController::process, this);
}
```

```cpp
void *ConsumerController::process(void *arg)
{
  // TODO: implements the ConsumerController's work
  // Casts the argument to a ConsumerController object
  ConsumerController *controller = (ConsumerController *)arg;
  while (1) // Infinite loop that keeps checking the worker queue size and scaling consumers up/down
  {
    // Calculates the proportion of items in the worker queue relative to its buffer size
    double worker_proportion = (double)controller->worker_queue->get_size() / controller->worker_queue->get_buffer_size();
    // If the worker queue is more than the high threshold
    if (worker_proportion > (double)controller->high_threshold / 100)
    {
      // Creates a new consumer to handle more items and starts it
      Consumer *new_consumer = new Consumer(controller->worker_queue, controller->writer_queue, controller->transformer);
      new_consumer->start();

      // Adds the new consumer to the consumers vector
      controller->consumers.push_back(new_consumer);
      std::cout << "Scaling up consumers from " << controller->consumers.size() - 1 << " to " << controller->consumers.size() << "\n";
    }
    // If the worker queue is less than the low threshold and there are more than 1 consumer (scale down)
    else if (worker_proportion < (double)controller->low_threshold / 100 && controller->consumers.size() > 1)
    {
      // Removes and cancels the last consumer from the vector
      Consumer *delete_consumer = controller->consumers[controller->consumers.size() - 1];
      delete_consumer->cancel();
      controller->consumers.pop_back();
      std::cout << "Scaling up consumers from " << controller->consumers.size() + 1 << " to " << controller->consumers.size() << "\n";
    }
    // Pauses for the specified check period before checking again
    usleep(controller->check_period);
  }
  // Returns nullptr when the thread finishes
  return nullptr;
}
```

vi. Main.cpp

說明:

- 實作內容同註解

```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
  TSQueue<Item *> *input_queue = new TSQueue<Item *>(READER_QUEUE_SIZE);
  TSQueue<Item *> *woker_queue = new TSQueue<Item *>(WORKER_QUEUE_SIZE);
  TSQueue<Item *> *output_queue = new TSQueue<Item *>(WRITER_QUEUE_SIZE);

  // Start the threads for reading, writing, producing, and controlling consumers
  Transformer *transformer = new Transformer();
  Reader *reader = new Reader(n, input_file_name, input_queue);
  Writer *writer = new Writer(n, output_file_name, output_queue);
  Producer *p1 = new Producer(input_queue, woker_queue, transformer);
  Producer *p2 = new Producer(input_queue, woker_queue, transformer);
  Producer *p3 = new Producer(input_queue, woker_queue, transformer);
  Producer *p4 = new Producer(input_queue, woker_queue, transformer);
  ConsumerController *controller = new ConsumerController(
      woker_queue, output_queue, transformer,
      CONSUMER_CONTROLLER_CHECK_PERIOD,
      CONSUMER_CONTROLLER_LOW_THRESHOLD_PERCENTAGE,
      CONSUMER_CONTROLLER_HIGH_THRESHOLD_PERCENTAGE);

  // Start all the threads
  reader->start();
  writer->start();
  controller->start();
  p1->start();
  p2->start();
  p3->start();
  p4->start();
```

```
    // Wait for the reader and writer threads to finish (join them to the main thread)
    reader->join();
    writer->join();

    // Once reading and writing are complete, clean up dynamically allocated memory
    delete p1;
    delete p2;
    delete p3;
    delete p4;
    delete controller;
    delete reader;
    delete writer;
    delete transformer;
    delete input_queue;
    delete woker_queue;
    delete output_queue;

    // End the program
    return 0;
}
```

II. Experiments / Result

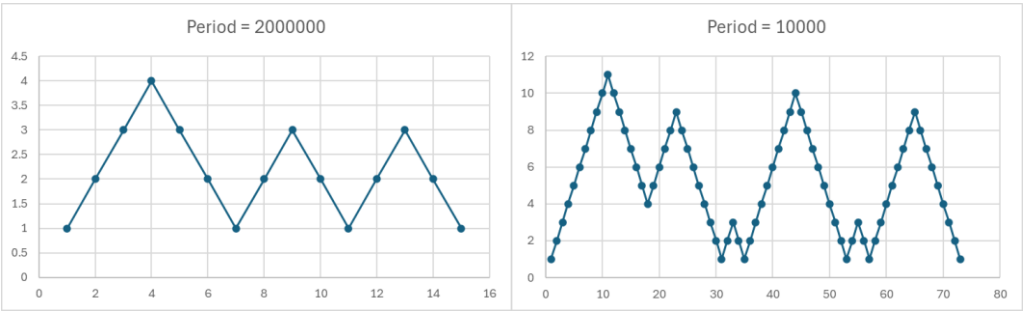
- 為了測試，在 main.cpp 中引入標頭檔`#include <chrono>`，並在 start process 和呼叫首行刪除程序之間計算起始-結束時間(此程式碼並無列在實作部分，僅在 main 做更動)。
- 主要使用 test case 01 測試。
- 一次測試僅更動題目要求的變數，其餘變數均為預設值不做更動(表格會列出當前更動項目)。
- 所有測試均計算三次(含)以上取平均值。
- 由於測試過程中偶爾會有極端值出現 (Ex. 200000ms/ 5000ms)，尚且無法斷定是否為連線或是其他問題，經過多次測試或是等待一段時間後再測試來確認當下是否為極端值出現，若非此狀況則會記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

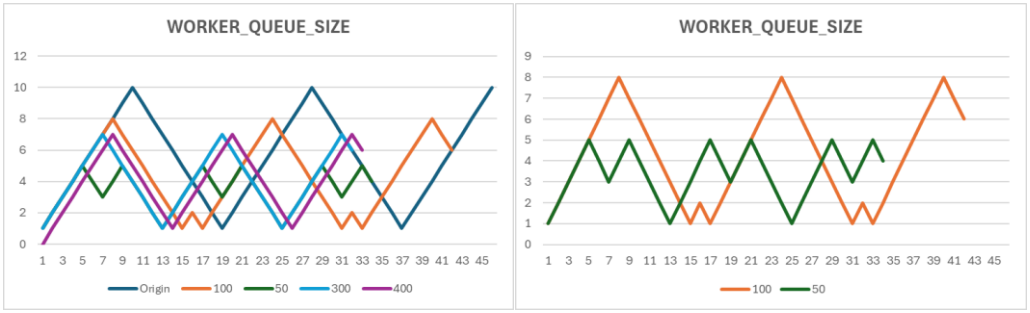
CONSUMER_CONTROLLER_LOW_THRESHOLD_PERCENTAGE	CONSUMER_CONTROLLER_HIGH_THRESHOLD_PERCENTAGE	Time Cost(ms)
10	80	74469
20	80	79286
20	90	79746
30	70	62531
40	60	65042

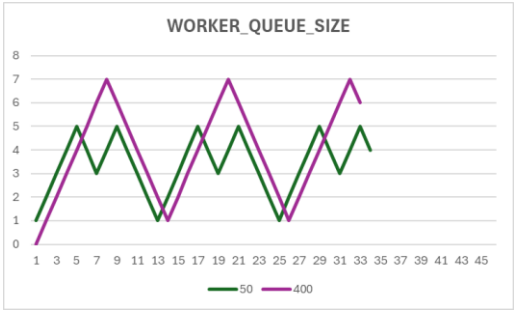
- low value 降低，震幅變大；high value 升高，震幅變低。
- 比對 high-low 的差值，縮小後震幅也會降低。
- 另外，time cost 似乎和這兩個 value 的關係不大(目前所有測試資料都在 50000-70000 左右浮動)；額外多測試的幾筆資料顯示，high value 和 time cost 似乎有正相關，推測由於 high value 升高，controller 越不容易創建新的 consumer，故執行時間延長。

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,

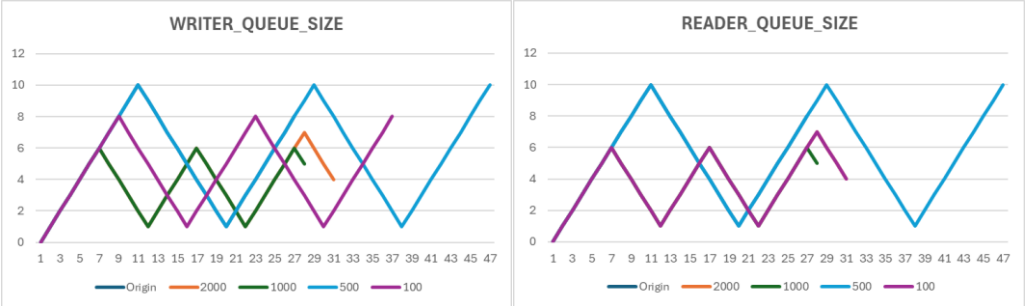
Time Usage 越小(但下降情況不明顯，很可能是因為 controller 是根據百分比調節 consumer 數量)

- 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

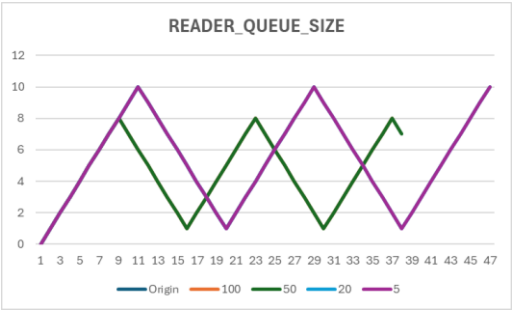


- WRITE__QUEUE_SIZE 和 Time Usage 的關係大致並無規律
- 目前結果顯示，WRITER_QUEUE_SIZE 對 Time Usage 與 CONSUMER SIZE CHANGE 並沒有太大影響，從製圖結果可得看出資料大多重疊且震幅規律相似。

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)，從製圖結果可得看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

code 題

給一段 code，code 做的事情基本上就是，int value[0:9]是一個 global array
用 for(int i=0; i<10; i++)來 create10 個 thread，每個 thread 的 function 是 hello，arg 是 i 的 pointer
*而 hello 大概長下面這樣 hello(&int i) {value[*i] = *i;}，create 完 10 個 thread 之後，會一樣用 for loop 做 thread.join，最後用 for loop 輸出 value[i]*

為什麼不會輸出 0~9

Ans: 因為 create thread 的時候 i 確實是 0~9，但 code 是用指標作為參數，thread 真正執行的時候，指標指到的數值已經變動了

為什麼 condition variable wait 要有 mutex lock 作為參數

條件變數 wait 不能直接確保臨界區域的排他性，其功能主要是釋放 mutex 並等待通知。
Wait 被呼叫時，mutex 會確保只有當前的 thread 可以檢查和修改共享資源，防止有其他 thread 同時檢查條件導致 deadlock。
Wait 會 atomic 解鎖 mutex 並進入等待，其他 thread 才能獲得 mutex 並更新 condition variable。
被喚醒後，mutex 會確保當前 thread 能正確檢查條件並進行操作，不會有其他 thread 同時搶佔資源。

Code 題 S 是一個 condition variable 請問下面的程式會發生什麼事

wait(S)
CRITICAL SECTION
wait(S)

當 thread 執行 wait(s)會釋放持有的 mutex 並進入等待，直到條件變數被其他 thread 喚醒；
被喚醒後 thread 會重新獲得 mutex 並執行臨界區內的程式碼。
再次執行 wait(s)會釋放 mutex 並等待，直到被喚醒。

◆ 很可能會無限期的 wait，因為條件變數的狀態沒有改變。

2. 為什麼 pthread_cond_wait 後其他 thread 仍然能進入 critical section
因為其功能只在將 mutex 釋放，當 pthread_cond_wait 釋放 mutex 時，其他執行緒可以獲得 mutex 並進入臨界區。

➢