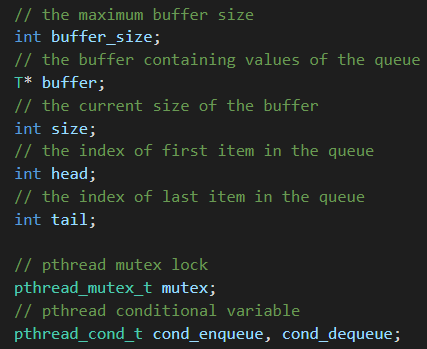
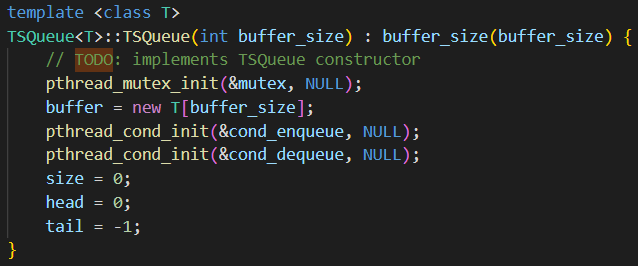
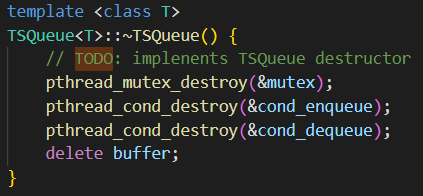
**Implementation**

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。

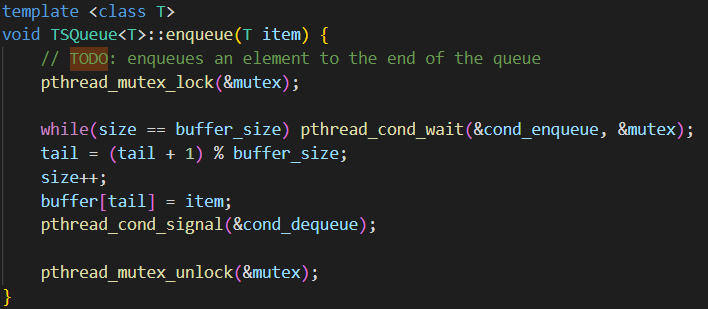
****



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。

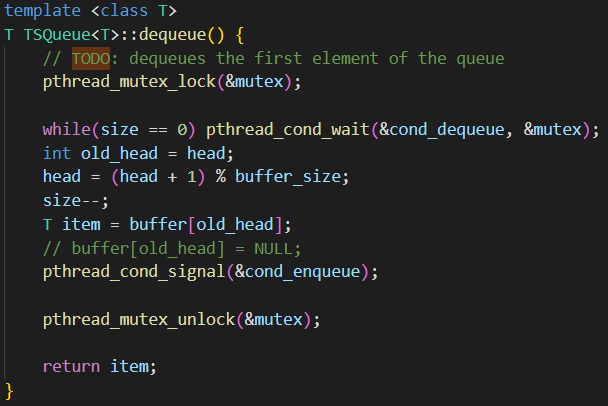


接下來便是destructor的實作，這一部分很簡單，只是將mutex和CV呼叫各自的destroy function，以及把buffer給delete掉而已。

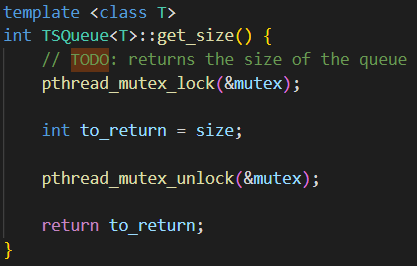


再往下便是這次作業的重頭戲——TSQueue的enqueue和dequeue。我們需要操作mutex和CV將TSQueue這個shared variable給保護好。在閱讀完講義後，我們知道CV也為shared variable，所以在操作CV時也需要用mutex去包住，即critical section。進入critical section後，先判斷目前queue的size是否等於buffer\_size，若為True則代表現在queue是滿的，無法再enqueue一個新的item進去，所以此時我們呼叫cond\_wait，將該thread block並將mutex釋放掉，好讓其他人可以拿到mutex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

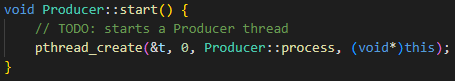


再來是dequeue，相同的，先利用mutex包出一個critical section，隨後在裡面檢查目前queue的size是否為0，若為True則代表目前沒有item在queue裡，無法dequeue，則我們呼叫cond\_wait使thread被block住；若為False或return from cond\_wait，則我們繼續以下statement，操作queue的index取得一個item，並將size-1，最後呼叫cond\_signal和release mutex並將取得的item return便結束dequeue。

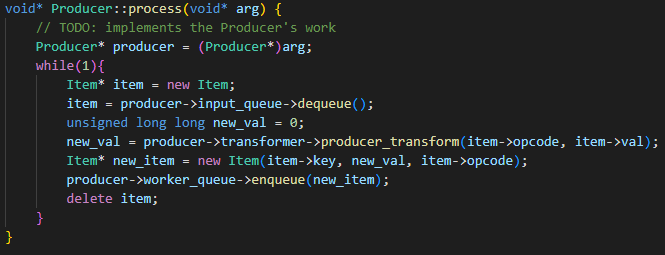


TSQueue最後一個function get\_size（），簡單的將目前queue裡面所有的item數量回傳，即將size給return，此處因為TSQueue本身為shared variable，在操作任何有關該class的一切，我們都要以mutex包起來創造出critical section以達到mutual exclusive。

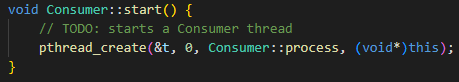
再來便是producer與consumer的實作。在實作前，我們有先trace過code structure裡的Reader，並搭配講義pthead的範例，得知如何創建一個pthread並讓其開始運作。



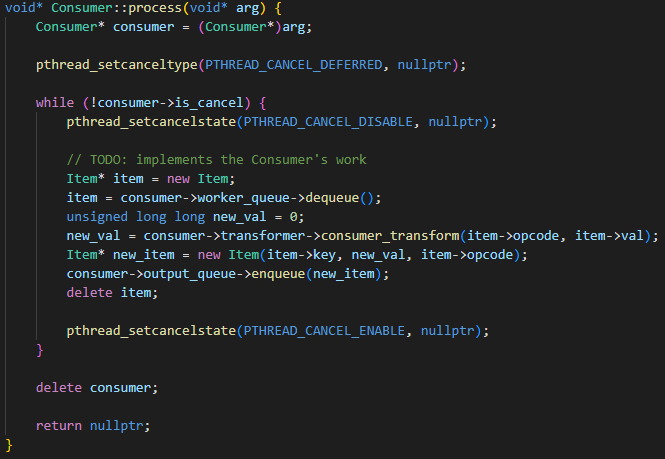
在producer的start()，我們呼叫pthread\_create()，將已經在thread.hpp宣告好的pthread\_t type的t傳入，並一同傳入thread的start routine，也就是producer自己的process，這樣便可以成功創建出一個pthread。



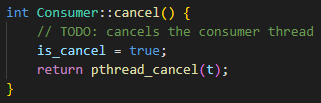
隨後的process()是thread開始運作後的第一個function，我們首先創建一個producer的instance接住process傳進來的arg，其中包含input\_queue，worker\_queue，transformer。隨後是一個無窮迴圈，讓thread可以一直運行。在迴圈內我們要做的事情在spec裡已經給出步驟了，先從input\_queue中取得一個item，所以呼叫producer->input\_queue->dequeue()，再呼叫transformer裡給producer轉換value使用的function producer\_transform將新的value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。



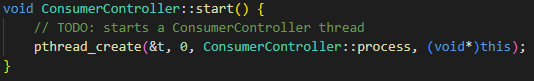
在consumer的start()裡，所做的事情與producer相同，皆是先創建出一個pthread，只是在這裡要傳入的start routine要為consumer自己的process function。



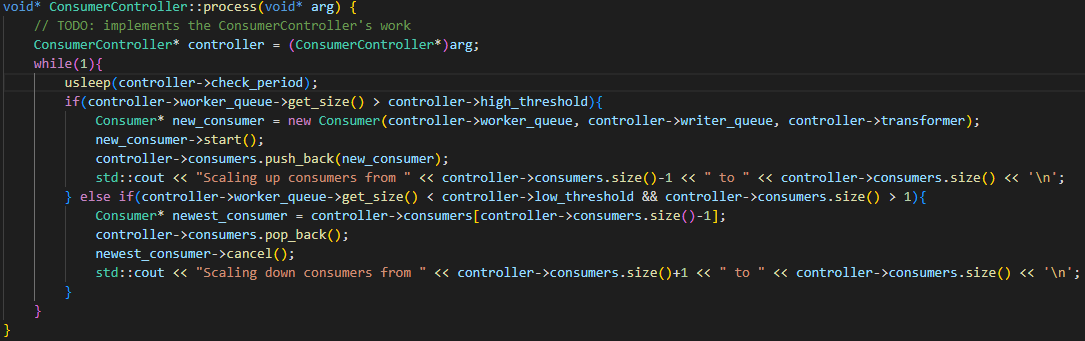
Consumer的process，一樣根據spec給出的步驟，先從worker\_queue呼叫dequeue取得item，並利用consumer\_transform依據opcode和舊value轉換出新的value，並再利用該新value和原本的key和opcode創建出一個新的item，最後將新的item enqueue到output\_queue裡並delete掉舊的item便結束



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。



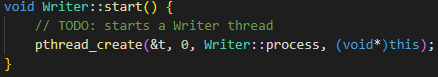
ConsumerController的start()與前面相同，只是改成傳入ConsumerController的process。



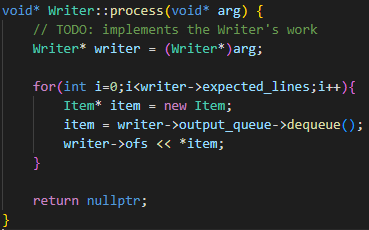
在ConsumerController的process裡，因為program開始後便會一直存在，所以我們同樣使用無窮迴圈的寫法，在迴圈內，我們首先利用usleep去掌控check\_period，因為在main.cpp裡check\_period的定義是micro seconds，且usleep接收的單位也是microsecond，所以就直接將check\_period丟進usleep就能達到讓thread sleep的效果，該thread便會在這段check\_period的時間內都是被block住，等到時間結束才會再回來執行下面statement，符合period的實作。在可以執行後，我們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

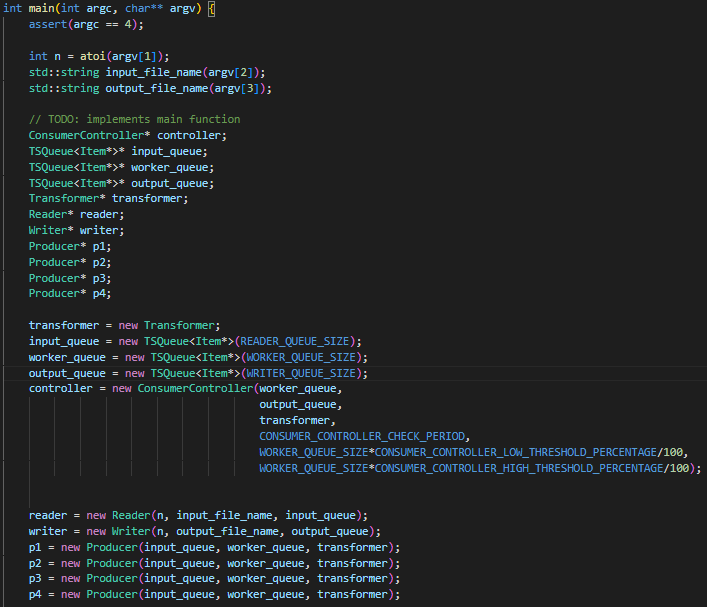
而若此時worker\_queue的size小於low\_threshold，並且consumers裡有兩個(含)以上的consumer，我們才需要減少一個consumer，因為如果只有剩下一個consumer，將其cancel掉後便沒有人可以再去做事，也不符合spec裡的at least one的要求。在符合條件後，我們從consumers裡取出最後一個consumer，並將vector裡刪掉該consumer的導向，並呼叫該consumer的cancel()將其delete掉，最後print出改變的訊息。



在Writer的start裡傳入自己的process以創建pthread。



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。



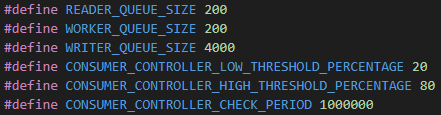
在main function裡，首先將我們所需要的任何東西都做initial，因為spec的要求producer需要四個，所以我們重複創建出p1~p4四個producer。



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**

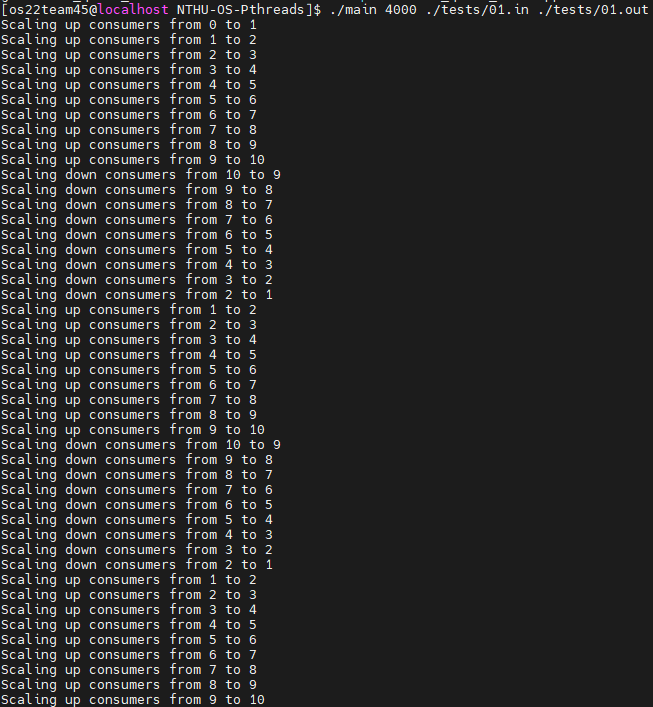
在每一項實驗中，對照組都是一樣的，所以我們先將對照組的結果寫在最前面，這樣後面就不用一直重複:



`./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。



`./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。

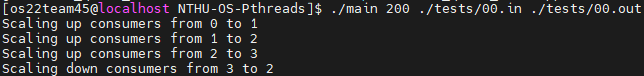


1. **Different values of CONSUMER\_CONTROLLER\_CHECK\_PERIOD**.

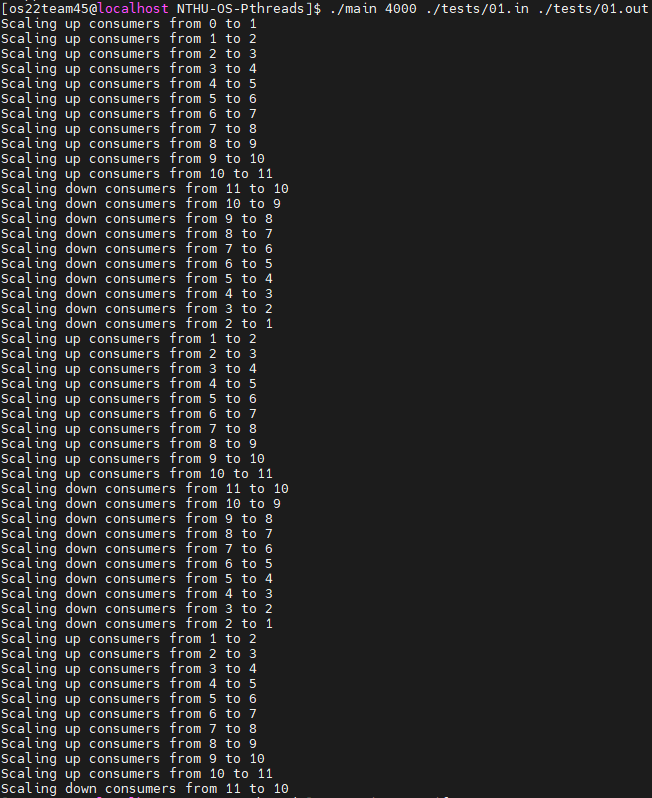
對照組:CONSUMER\_CONTROLLER\_CHECK\_PERIOD = 1000000

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。



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。



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。

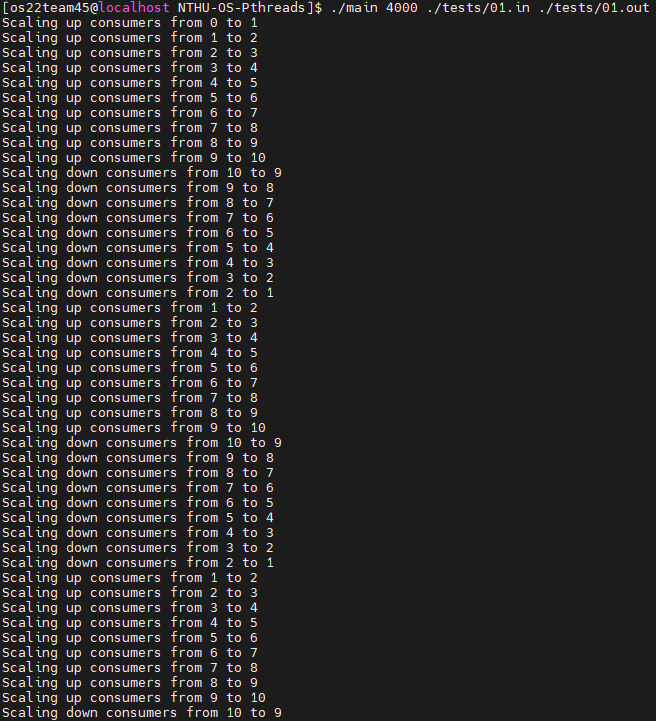
1. **Different values of CONSUMER\_CONTROLLER\_LOW\_THRESHOLD\_PERCENTAGE and CONSUMER\_CONTROLLER\_HIGH\_THRESHOLD\_PERCENTAGE.**

**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。



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。

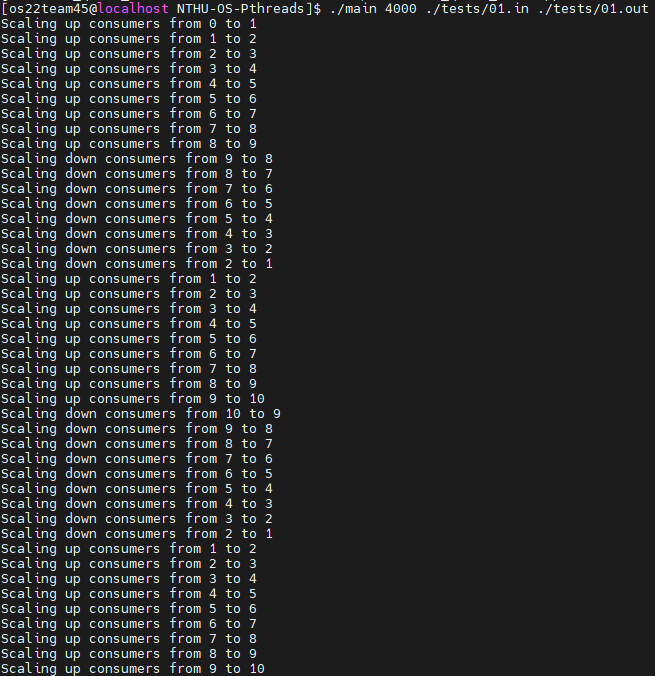
**對照組: CONSUMER\_CONTROLLER\_HIGH\_THRESHOLD\_PERCENTAGE = 80**

**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。



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。



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。

1. 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。



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

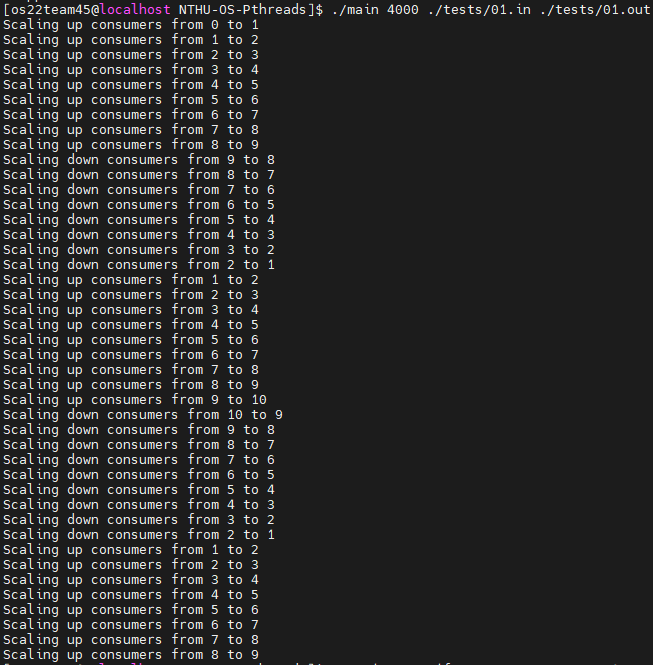
這裡可以看出worker\_queue\_size變小導致上下限都縮小，因此scale up變得更容易達成。

**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。



1. 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

1. 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:**

1. **對照組：**

**WORKER\_QUEUE\_SIZE = 200 CONSUMER\_CONTROLLER\_LOW\_THRESHOLD\_PERCENTAGE=20**

**CONSUMER\_CONTROLLER\_HIGH\_THRESHOLD\_PERCENTAGE=80**

**實驗組：**

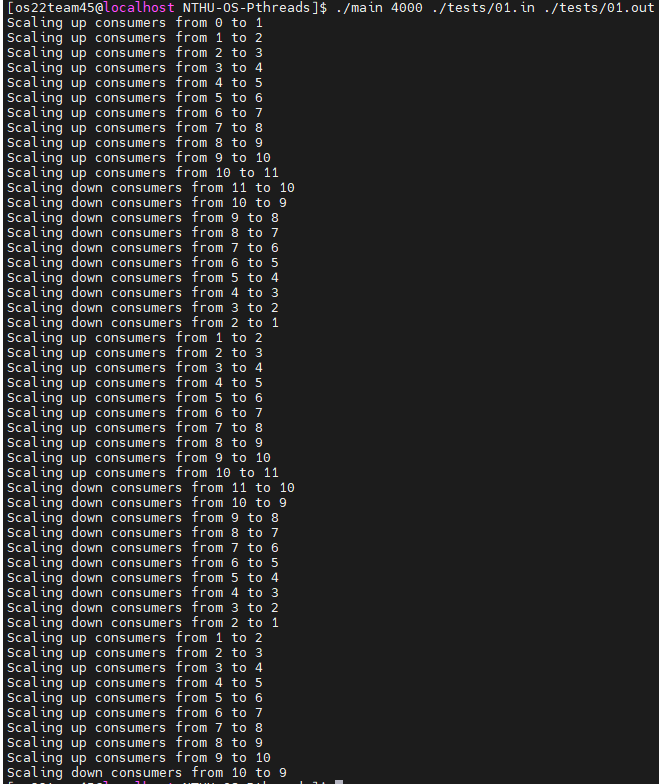
**WORKER\_QUEUE\_SIZE = 400 CONSUMER\_CONTROLLER\_LOW\_THRESHOLD\_PERCENTAGE=10**

**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



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。



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。

1. **對照組：**

**CONSUMER\_CONTROLLER\_LOW\_THRESHOLD\_PERCENTAGE=20**

**CONSUMER\_CONTROLLER\_HIGH\_THRESHOLD\_PERCENTAGE = 80**

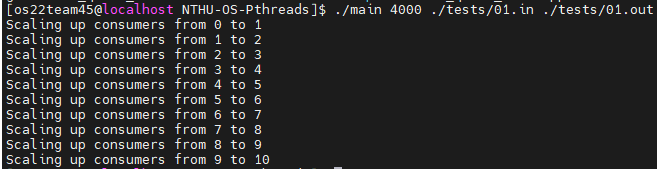
**實驗組：**

**CONSUMER\_CONTROLLER\_LOW\_THRESHOLD\_PERCENTAGE=0**

**CONSUMER\_CONTROLLER\_HIGH\_THRESHOLD\_PERCENTAGE = 80**

****

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。

****

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。

1. **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**





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!