OS Pthreads Programming Assignment Report

Team Members: 張智孝、吳泊諭

Contributions

張智孝:Code tracing, `producer.hpp` , `consumer.hpp` , `consumer_controller.hpp`
吳泊諭:Code tracing `tsq_queue.hpp` , `writer.hpp` , `main.hpp`

Implementation

TSQueue

The queue itself (without the thread-safe part) is maintained as a circular buffer, where the `head` is exclusive, and the `tail` is inclusive - the last item in the queue is always `buffer[tail]` , and the first item in the queue is always `(head + 1) % buffer_size` . Note that the modulo is necessary for cyclic indexing on the queue.
When enqueueing an item, `tail` is incremented cyclicly (`tail = (tail + 1) % buffer_size`), and then the item is placed in `buffer[tail]` . Similarly, when dequeueing, `head` is incremented cyclicly (`head = (head + 1) % buffer_size`), and `buffer[head]` is returned. The size of the queue is maintained as `size` , incremented and decremented for each `enqueue` and `dequeue` operation.
To make the queue thread-safe, we use the same mutex for both `enqueue` and `dequeue` to ensure atomicity over the queue.
Two condition variables are used so that `enqueue` can wait for a `dequeue` operation when the queue is full, and `dequeue` can wait for an `enqueue` operation when the

queue is empty, all without busy waiting. (We still have to wrap `pthread_cond_wait` with the corresponding `while` loop because there may be spurious wakeups)
Of course, this means we have to use `pthread_cond_signal` to wake up `enqueue` calls waiting for `dequeue` 's condition variable, and vice versa.
In the destructor, we also have to acquire the queue's mutex first with `pthread_mutex_lock` , so that we can wake up every thread waiting on the two condition variables with `pthread_cond_broadcast` *without* them looping again and start waiting on the condition variables again (they will be blocked by the mutex lock). Then we can safely destroy the condition variables with `pthread_cond_destroy` , and finally, destroy the queue's mutex, before being able to release the underlying buffer's memory.

ConsumerController

The `ConsumerController` maintains the number of consumer by monitoring worker queue's size by calling `worker_queue.get_size` .
To peroidly check the worker queue, `usleep(check_period)` is used. After every period, `ConsumerController` check the size of worker queue, trying to increase(decrease) consumer if the worker queue's size is greater(less) then `high_threshold` (low_threshold).
When the worker queue's size is greater than `high_threashold` , we will construct a new consumer(denoted as newConsumer) by calling `Consumer(worker_queue, writer_queue)` . After the construction of a consumer, we append newConsumer into consumer vector by calling `consumerController->consumers->push_back()` . Then we enable the newConsumer to work by calling `newConsumer->start`

When the worker queue's size is lower than `low_threshold` and the number of consumer is greater than 1, we get the first element of consumer(denoted as newestConsumer) vector by calling `consumerController->consumers->pop_back()`. After the pop operation, we disable newestConsumer by calling `newestConsumer->cancel`.
There are two things needed to be noted. First, the consumer vector is used as a stack, so no matter `consumerController->consumers.push_back()` or `consumerController->consumers.pop_back()` handles the newest consumer. Thus, the first several consumer may be enabled for a long time. Second, `consumerController->consumers.push_back()` and `consumerController->consumers.pop_back()` is not exclusive. That is, these two lines are triggered indepently(when low_threshold is greater than high_threshold, both lines will be triggered)

Consumer

The consumer repeatly consumes out the element in the worker queue if such consumer is not canceled. If such comsumer is canceled, just stop consumering and delete the consumer. Otherwise, the consumer attempts to dequeue the worker queue and store it in `item` by calling `Item *item = consumer->worker_queue->dequeue()`.
Since worker queue is implemented by TSQ and only returns item when there is elemnt in worker queue, we can always expect that the consumer can always get something rather than a `NULL` pointer. Thus, there is no need to re-check if the return value of `consumer->worker_queue->dequeue()` is `NULL`.
After that, we utilze the `item` to perform transformation. Then enqueue the transformed `item` into `output_queue` by calling `consumer->output_queue->enqueue(item)`

Producer

The producer repeatly consumes out the element in the input queue if such producer is not canceled. If not, the producer attempts to dequeue the input queue and store it in `item` by calling `Item *item = prodcuer->input_queue->dequeue()` . If such producer is canceled, let the caller handle the deletion of the producer.
Since input queue is implemented by TSQ and only returns item when there is elemnt in input queue, we can always expect that the producer can always get something rather than a `NULL` pointer.
After that, we utilze the `item` to perform transformation. Then enqueue the transformed `item` into `worker_queue` by calling `producer->worker_queue->enqueue(item)`

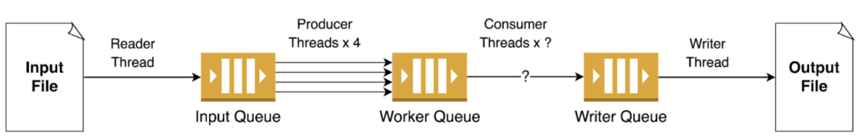
Writer

The writer repeatly dequeues the output queu and writer it into writer outputstream by calling `*writer->output_queue->dequeue()` until the number of expected linese to writer is 0. Considering that the return value of `*writer->output_queue->dequeue()` is a pointer to `Item` .
We have to dereference it such that it makes operator `<<` , defined in `item.hpp` , function properly.
Besides, writer's destructor `Writer::~Writer` execute ofs close by calling `ofs.close()`

Main

Main connectes each component to make them function together. It is composed of two parts: Initialize, Work, Terminate.

- Initialize:

  As shown in the above image, it takes several classes: Reader Thread(Reader), Writer Thread(Writer), Producer, Consumer, ConsumerController, Input Queue, Worker Queue, Writer Queue. We have to construct them one by one, passing each class requiring parameter. Two things needs some attention: First, according to the spec, there are 4x Producer and 0x Consumer initially. Thus, we don't need to construct Consumer at the begin. Second, ConsumerController's fifth and sixth parameter are both integer, implying the `low_threshold` and `high_threshold` respectively. Considering that `CONSUMER_CONTROLLER_LOW_THRESHOLD_PERCENTAGE` and `CONSUMER_CONTROLLER_HIGH_THRESHOLD_PERCENTAG` are also integers, divide by 100 is necessary for them after those threshold percentage multiply by the `WORKER_QUEUE_SIZE`

- Work:
  After initialization, each class call `start()` to start working. But at this point, reader and writer are not yet communicated to each other. To make reader and writer connected, we have to call join. That is calling `reader->join()` and `writer->join()`.
- Terminate:
  After the finishing those jobs, delete all the class used in `main.hpp`.

Experiments

To amplify the experiment result, we take bigger test bench(01). And we run the follow command to calculate the total runtime.

```
time ./main 4000 ./tests/01.in ./tests/01.out
```

1. Different values of `CONSUMER_CONTROLLER_CHECK_PERIOD`.

CHECK_PERIOD	1	10	100	1000	10000	1(
TIME	53.32	57.68	53.25	53.36	52.83	5
MAXCONSUMER	10	10	9	9	12	1

2. Different values of `CONSUMER_CONTROLLER_LOW_THRESHOLD_PERCENTAGE` and `CONSUMER_CONTROLLER_HIGH_THRESHOLD_PERCENTAGE`.

HIGH_PER\LOW_PER	0	25	75	100
0	52.67/11	54.74/11	58.20/10	60.92/8
25	52.66/11	55.43/11	57.77/11	60.81/10
75	58.74/10	58.78/10	58.99/10	60.75/9
100	RE	RE	RE	RE

Becasue the slide tolds us to surpass the high threshold, we can't add extra consumer into an empty queue when high threshold is 100. This makes the process RE.

3. Different values of `WORKER_QUEUE_SIZE`.

WORKER_QUEUE_SIZE	0	2	20	200	2000
TIME	RE	55.31	61.57	59.7	75.17
MAXCONSUMER	RE	9	9	10	11

Different from 100 percent of high threshold, the process will stuck if the worker queue's size is 0 for we can't emplace input instruction into the worker queue.

4. What happens if `WRITER_QUEUE_SIZE` is very small?

WRITER_QUEUE_SIZE	0	4	40	400	4000
TIME	INF	58.79	59.7	58.79	59.7
MAXCONSUMER	INF	10	10	10	10

When the `WRITER_QUEUE_SIZE` is 0, the program will not terminate and the total consumer will keep increasing for the `Item` in worker queue can't transfer to writer queue.

Those accumulated `Item` make the `ConsumerController` generate lots of consumer but fail to tackle the problem because the `WRITER_QUEUE_SIZE` is 0.

5. What happens if `READER_QUEUE_SIZE` is very small?

READER_QUEUE_SIZE	0	2	20	200	2000
TIME	RE	56.43	61.54	59.70	74.95
MAXCONSUMER	RE	9	9	10	11

When the `READER_QUEUE_SIZE` is very small, the program stuck for no `Item` can be placed into `INPUT_QUEUE` .

Difficulties

Printing out debug message fails during concurrent access.

Feedback