CS342301: Operating System
Pthreads Programming

Team no. 46

Member Contribution:

陳敬中: Trace Code, Report, Implementation (50 %)

白宸安: Trace Code, Report, Implementation (50 %)

Report

1. Explain your implementation as requested. (Please tell us as much detail as possible.)

The implementation refers to the process listed in the homework spec.

1. TSQueue
2. Producer
3. Consumer
4. ConsumerController
5. Writer
6. main.cpp

ts_queue.hpp

Constructor and Destructor

```
template <class T>
TSQueue<T>::TSQueue(int buffer_size) : buffer_size(buffer_size) {
    // TODO: implements TSQueue constructor
        buffer = new T[buffer_size];
    pthread_mutex_init(&mutex, NULL);
    pthread_cond_init(&cond_enqueue, NULL);
    pthread_cond_init(&cond_dequeue, NULL);
}

template <class T>
TSQueue<T>::~TSQueue() {
    // TODO: implenents TSQueue destructor
        delete[] buffer;
    pthread_mutex_destroy(&mutex);
    pthread_cond_destroy(&cond_enqueue);
    pthread_cond_destroy(&cond_dequeue);
}
```

First, we wrote the constructor and destructor, and initialized the buffer, mutex, condition variable, respectively.

enqueue

```
template <class T>
void TSQueue<T>::enqueue(T item) {
    // TODO: enqueues an element to the end of the queue
    pthread_mutex_lock(&mutex);

    // Wait if the buffer is full
    while (size == buffer_size) {
```

```
        pthread_cond_wait(&cond_enqueue, &mutex);
    }

    // Enqueue the item
    buffer[tail] = item;
    tail = (tail + 1) % buffer_size;
    size++;

    pthread_cond_signal(&cond_dequeue);

    pthread_mutex_unlock(&mutex);
}
```

Next is the enqueue. You must get the mutex lock before entering. Only one thread can enter the critical section of the enqueue at a time.

If the buffer is full, you have to wait for someone to dequeue it before placing it again.

After placing the item, call the signal of the dequeue so that the process waiting in the dequeue can continue.

Finally release the mutex lock.

dequeue

```
template <class T>
T TSQueue<T>::dequeue() {
        // TODO: dequeues the first element of the queue
        pthread_mutex_lock(&mutex);

    // Wait if the buffer is empty
    while (size == 0) {
        pthread_cond_wait(&cond_dequeue, &mutex);
    }

    // Dequeue the item
    T item = buffer[head];
    head = (head + 1) % buffer_size;
    size--;

    pthread_cond_signal(&cond_enqueue);

    pthread_mutex_unlock(&mutex);

    return item;
}
```

It is very similar to the enqueue method, except that if the buffer is empty, you have to wait, and the signal to call is enqueue signal, so that the process in the enqueue can continue.

get size

```
template <class T>
int TSQueue<T>::get_size() {
    // TODO: returns the size of the queue
        pthread_mutex_lock(&mutex);
    int current_size = size;
    pthread_mutex_unlock(&mutex);

    return current_size;
}
```

Before getting the size, must first get the mutex lock to avoid reading different size values.

producer.hpp

start

```
void Producer::start() {
        // TODO: starts a Producer thread
        pthread_create(&t, 0, Producer::process, (void*)this);
}
```

Use pthread_create to create a producer thread.

void* Producer::process(void* arg)

```
void* Producer::process(void* arg) {
    // TODO: implements the Producer's work
    Producer* producer = (Producer*)arg;

    while (true) {  // process不會結束，因為input_queue可能會一直增加。
      Item* item = producer->input_queue->dequeue();

      if (item == nullptr) {
          break;
      }

      item->val = producer->transformer->producer_transform(item->opcode, item->va

      producer->worker_queue->enqueue(item);
    }

    //delete producer;

    return nullptr;
}
```

The function where the producer thread performs the main work.

First, convert arg into a pointer pointing to producer.

Then continue to dequeue the item from the input queue, perform producer transform on item→val, and finally put it into the worker queue.

consumer.hpp

start

```
void Consumer::start() {
    // TODO: starts a Consumer thread
    pthread_create(&t, 0, Consumer::process, (void*)this);
}
```

Use pthread_create to create consumer thread.

cancel

```
int Consumer::cancel() {
    // TODO: cancels the consumer thread
    is_cancel = true;
  pthread_cancel(t);

  return pthread_join(t, nullptr);
}
```

If cancel is called, the consumer thread is terminated.

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
        Item* item = consumer->worker_queue->dequeue();

        if (item == nullptr) {
            break;
        }

        item->val = consumer->transformer->consumer_transform(item->opcode, item->

        consumer->output_queue->enqueue(item);
```

```
        pthread_setcancelstate(PTHREAD_CANCEL_ENABLE, nullptr);
    }

    //delete consumer;

    return nullptr;
}
```

The consumer thread performs the main work.

First convert arg into a pointer pointing to consumer.

Then, before the consumer thread is canceled, it continuously dequeues the item from the worker queue, performs consumer transform on item→val, and finally puts it into the output queue.

consumer_controller.hpp

start

```
void ConsumerController::start() {
    // TODO: starts a ConsumerController thread
    pthread_create(&t, 0, ConsumerController::process, (void*)this);
}
```

Use pthread_create to create a consumer controller thread.

process

```
void* ConsumerController::process(void* arg) {
    // TODO: implements the ConsumerController's work
    ConsumerController* controller = (ConsumerController*)arg;
    int consumerCount = 0;

    while (true) {
        usleep(controller->check_period);

        int workerQueueSize = controller->worker_queue->get_size();

        if (workerQueueSize > controller->high_threshold ) {
            // Create a new Consumer thread
            Consumer* newConsumer = new Consumer(controller->worker_queue,
                                controller->writer_queue, controller->transformer)
            newConsumer->start();
            controller->consumers.push_back(newConsumer);

                        std::cout << "Scaling up consumers from " << consumerCount
                                << consumerCount+1 << std::endl;
                        consumerCount++;
        }
        else if (workerQueueSize < controller->low_threshold &&
                        controller->consumers.size() > 1) {
```

```
            // Cancel the newest Consumer thread
                        Consumer* newestConsumer = controller->consumers.back();
                        newestConsumer->cancel();
                        controller->consumers.pop_back();
                        std::cout << "Scaling down consumers from " << consumerCou
                             << consumerCount-1 << std::endl;
                        consumerCount--;
        }

    }

    return nullptr;
}
```

The consumer controller thread performs the main work.

First, convert arg into a pointer pointing to consumer controller.

Then make a dynamic judgment. After every check_period micro second (the default is equal to 1 second), check how many items are in the current buffer of the worker queue.

Use the controller→consumers vector to manage consumer threads.

If the item number exceeds the upper limit (default 80%), add a consumer thread.

If the item number is lower than the lower limit (default 20%), one consumer thread will be reduced.

If there is already a consumer thread, at least one consumer must remain executed until the end of the program.

writer.hpp

start

```
void Writer::start() {
    // TODO: starts a Writer thread
    pthread_create(&t, 0, Writer::process, (void*)this);
}
```

Use pthread_create to create a writer thread.

process

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

    while (writer->expected_lines--) {
        Item* item = writer->output_queue->dequeue();
        writer->ofs << item->key << " " << item->val << " " << item->opcode << std
        delete item;
    }
```

```
    return nullptr;
}
```

The function where the writer thread does the main work.

First convert arg into a pointer pointing to writer.

Then it is executed continuously to dequeue the item from the output queue, write the item information into the output file, and finally delete the item.

main.cpp

```
int main(int argc, char** argv) {
        assert(argc == 4);

        int n = atoi(argv[1]);          // opcode數量
        std::string input_file_name(argv[2]);
        std::string output_file_name(argv[3]);

        // TODO: implements main function
        // Create queues
        TSQueue<Item*> inputQueue(READER_QUEUE_SIZE);
    TSQueue<Item*> workerQueue(WORKER_QUEUE_SIZE);
    TSQueue<Item*> outputQueue(WRITER_QUEUE_SIZE);

        // Create transformer
    Transformer transformer;

        // Create reader
    Reader reader(n, input_file_name, &inputQueue);


        // Create producers
        std::vector<Producer> producers;
    for (int i = 0; i < 4; i++) {
        producers.emplace_back(&inputQueue, &workerQueue, &transformer);
    }

        // Create consumer controller
    int low_threshold = (CONSUMER_CONTROLLER_LOW_THRESHOLD_PERCENTAGE *
                WORKER_QUEUE_SIZE) / 100;
    int high_threshold = (CONSUMER_CONTROLLER_HIGH_THRESHOLD_PERCENTAGE *
                WORKER_QUEUE_SIZE) / 100;

    ConsumerController consumerController(&workerQueue, &outputQueue,
                &transformer,
        CONSUMER_CONTROLLER_CHECK_PERIOD, low_threshold,
        high_threshold);


    // Create writer
```

```
    Writer writer(n, output_file_name, &outputQueue);

        // Start reader, writer, producer, and consumer controller threads
    reader.start();
    writer.start();

    for (auto& producer : producers) {
        producer.start();
    }

    consumerController.start();

    // Wait for termination
    reader.join();
    writer.join();

    return 0;
}
```

1. Create 1 input, 1 worker and 1 output queue
2. Create a transformer
3. Create a reader
4. Create 4 producers
5. Create a consumer controller and convert the upper and lower limits into percentages.
6. Create a writer
7. Start each of the above classes
8. After waiting for the reader and writer to finish, the program can be terminated.

2. Experiment

1. Different values of CONSUMER_CONTROLLER_CHECK_PERIOD.

CONSUMER_CONTROLLER_CHECK_PERIOD→4s

```
real    1m14.213s
user    7m8.477s
sys     0m0.222s
```

CONSUMER_CONTROLLER_CHECK_PERIOD→0.25s

```
real    0m50.833s
user    7m8.111s
sys     0m0.201s
```

CONSUMER_CONTROLLER_CHECK_PERIOD→1s

```
real    0m59.564s
user    7m8.578s
sys     0m0.230s
```

Faster update frequency can increase performance because it can more dynamically respond to bursts during reading. The disadvantage is that there is some overhead when checking the consumer controller.

2. Different values of CONSUMER_CONTROLLER_LOW_THRESHOLD_PERCENTAGE and CONSUMER_CONTROLLER_HIGH_THRESHOLD_PERCENTAGE.

#define CONSUMER_CONTROLLER_LOW_THRESHOLD_PERCENTAGE 10

#define CONSUMER_CONTROLLER_HIGH_THRESHOLD_PERCENTAGE 90

```
real    1m0.340s
user    7m8.356s
sys     0m0.231s
```

#define CONSUMER_CONTROLLER_LOW_THRESHOLD_PERCENTAGE 30

#define CONSUMER_CONTROLLER_HIGH_THRESHOLD_PERCENTAGE 70

```
real    0m59.551s
user    7m8.335s
sys     0m0.187s
```

#define CONSUMER_CONTROLLER_LOW_THRESHOLD_PERCENTAGE 50

#define CONSUMER_CONTROLLER_HIGH_THRESHOLD_PERCENTAGE 80

```
real    0m59.681s
user    7m8.192s
sys     0m0.203s
```

The benefit of changing these two is not great, because it is just the upper and lower limits of the boundary, but when a burst occurs, it usually exceeds the upper limit quickly. The reason is that the buffer size is only 200. Perhaps changing the upper and lower limits will be more effective when the buffer size becomes larger.

3. Different values of WORKER_QUEUE_SIZE.

#define WORKER_QUEUE_SIZE 800

```
real    0m59.692s
user    7m8.100s
```

```
sys     0m0.171s
```

It takes a long time to wait at first, because it takes more than 160 items to start the first consumer.

In addition, the number of consumers is adjusted less often because the worker queue size is larger and the range that can float in the middle is larger, so it is not easy to reach the upper or lower limit.

#define WORKER_QUEUE_SIZE 800

#define CONSUMER_CONTROLLER_LOW_THRESHOLD_PERCENTAGE 10

#define CONSUMER_CONTROLLER_HIGH_THRESHOLD_PERCENTAGE 90

```
real    1m42.734s
user    7m7.842s
sys     0m0.189s
```

After increasing the worker queue size, adjusting the upper and lower limits will have an effect.

If the upper limit is higher, workers will not be added until items almost fill the worker queue, which will result in poor dynamic load balancing.

#define WORKER_QUEUE_SIZE 800

#define CONSUMER_CONTROLLER_LOW_THRESHOLD_PERCENTAGE 10

#define CONSUMER_CONTROLLER_HIGH_THRESHOLD_PERCENTAGE 50

```
real    0m57.261s
user    7m7.653s
sys     0m0.203s
```

The number of consumers will be increased relatively early, but the balance will soon return.

Therefore, in summary, adjusting the upper and lower limits does not help performance much.

4. What happens if WRITER_QUEUE_SIZE is very small?

#define WRITER_QUEUE_SIZE 20

```
real    1m0.660s
user    7m8.131s
sys     0m0.271s
```

#define WRITER_QUEUE_SIZE 2

```
real    0m59.646s
user    7m8.321s
sys     0m0.228s
```

Adjusting the writer queue seems to have no impact on performance.

Some effects are that because the queue size is very small, it is almost impossible to temporarily store things, causing the queue to lose its intended functionality.

5.What happens if READER_QUEUE_SIZE is very small?

#define READER_QUEUE_SIZE 20

```
real    0m59.501s
user    7m8.375s
sys     0m0.222s
```

#define READER_QUEUE_SIZE 2

```
real    0m59.550s
user    7m8.353s
sys     0m0.215s
```

Changing the reader queue size does not seem to have any impact on performance.

When the reader queue size is small, it means that there are few reader items that can be temporarily stored, which may reduce the parallel efficiency of the producer.

In summary, because the input queue, worker queue, and output are queues between the reader and the writer, the queue size actually has little impact, because the bottleneck is still on the leftmost and rightmost readers and writers.

If the number of readers or writers is different, the queue size will have a greater impact.

3. Explain what additional experiments you have done and what are the results. You are encouraged to do more experiments.

In addition, we conducted experiments to test the fastest performance, and tried to mix and adjust various parameters. We found that the maximum speed of the program when running test01 was about 50 seconds, and it seemed that it could not go any faster.

As for the parameters, the biggest impact is the consumer controller check period, but it seems that the maximum performance can be achieved by checking once every 0.25 seconds. Changing it to once every 0.1 seconds does not help much.

for other parameters, they hardly contribute much to performance.

The best configuration tested so far is as follows. Nearly 10 seconds faster than the origin.

```
#define READER_QUEUE_SIZE 100
#define WORKER_QUEUE_SIZE 100
#define WRITER_QUEUE_SIZE 4000
#define CONSUMER_CONTROLLER_LOW_THRESHOLD_PERCENTAGE 10
#define CONSUMER_CONTROLLER_HIGH_THRESHOLD_PERCENTAGE 40
#define CONSUMER_CONTROLLER_CHECK_PERIOD 100000
```

```
real    0m50.278s
user    7m8.028s
sys     0m0.234s
```

Below are the results for default values.

```
#define READER_QUEUE_SIZE 200
#define WORKER_QUEUE_SIZE 200
#define WRITER_QUEUE_SIZE 4000
#define CONSUMER_CONTROLLER_LOW_THRESHOLD_PERCENTAGE 20
#define CONSUMER_CONTROLLER_HIGH_THRESHOLD_PERCENTAGE 80
#define CONSUMER_CONTROLLER_CHECK_PERIOD 1000000

real    0m59.564s
user    7m8.578s
sys     0m0.230s
```

4. What difficulties did you encounter when implementing this assignment?

The difficulty encountered in implementation was that it took a lot of time to understand the relationship between each class at the beginning, because the program has many files, and you have to understand their relationship when tracing.

The critical section when dealing with ts_queue is more complicated, and it takes long time to write it.

In addition, after the initial implementation, when testing, we found that both the reader and the writer could complete their respective tasks. The output file also had something, but the values in the middle were not transformed. Later, after debugging, we discovered that we didn't update the value to item→val when calling the producer and consumer transform. After updating, the answer becomes correct.