113-1 Operating System Pthread

Group 45: 111062332 朱誼學, 111062333 高英耀

Contributions	111062332	111062333
Trace code	50%	50%
Implementation	50%	50%
Report	50%	50%
Explanation	We've done all our work in discord vc	

Implementation

TSQueue: in ts_queue.hpp

1. `TSQueue(int buffer_size)`:
    - We should initialize mutex and CV (condition variable) in the constructor.

```
template <class T>
TSQueue<T>::TSQueue(int buffer_size) : buffer_size(buffer_size) {
    // TODO: implements TSQueue constructor
    buffer = new T[buffer_size];
    size = 0;
    head = 0;
    tail = buffer_size - 1;

    pthread_mutex_init(&mutex, NULL);
    pthread_cond_init(&cond_enqueue, nullptr);
    pthread_cond_init(&cond_dequeue, nullptr);
}
```

2. `~TSQueue()`
    - we release memory and destroy synchronization tools in the destructor.

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

3. `enqueue(T item)`
    - In this function, we have to handle the sychronization problem because buffer is a shared data.
    - So first we have to justify a critical section by `pthread_mutex_lock(&mutex)` and `pthread_mutex_unlock(&mutex)`.
    - And in the critical section, `enqueue()` have to wait until the buffer is not full. It will be blocked by `pthread_cond_wait(&cond_enqueue, &mutex);` if the queue is full.
    - Notice that, in `pthread_cond_wait`, the thread will enter waiting state and release the lock. After `signal()` is triggered, it will re-aqcuire the lock again then continue the execution.
    - Before leaving critical section, call `pthread_cond_signal()` to wake `dequeue()` up.

```
template <class T>
void TSQueue<T>::enqueue(T item) {
```

```
    // TODO: enqueues an element to the end of the queue
    pthread_mutex_lock(&mutex); // enter critical section
    // ------ critical section ------
    // block if the queue is full
    while (size == buffer_size)
        pthread_cond_wait(&cond_enqueue, &mutex);

    buffer[head] = item;
    head = (head + 1) % buffer_size;
    ++size;

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

4. `dequeue()`
    - We also have to do synchronization in this function.
    - The difference between `dequeue()` and `enqueue()` is that `dequeue()` have to wait for the queue is not empty.
    - Before leaving critical section, signal `cond_enqueue` as well.
    - And at last, remember to return the pop out value.

```
template <class T>
T TSQueue<T>::dequeue() {
    // TODO: dequeues the first element of the queue
    T val;
    pthread_mutex_lock(&mutex); // enter critical section
    // ------ critical section ------
    // block if no element to be dequeue
    while (size == 0)
        pthread_cond_wait(&cond_dequeue, &mutex);

    tail = (tail + 1) % buffer_size;
    val = buffer[tail];
    --size;

    pthread_cond_signal(&cond_enqueue);
    // ------ critical section ------
    pthread_mutex_unlock(&mutex); // leave critical section
    return val;
}
```

5. `get_size()`
    - It's a getter function, so just directly return the size.

```
template <class T>
int TSQueue<T>::get_size() {
```

```
    // TODO: returns the size of the queue
    return this->size;
}
```

Producer: in producer.hpp

1. For producer, consumer_controller, and writer, we just need to compelete the `start()` and `process(void *arg)` function. For consumer, there is one more `cancel()` should be done.
2. `Producer::start()`
   - `pthread_create(&t, nullptr, Producer::process, (void *) this)` is used to create (fork) a new thread, which means the new thread will run `process(void *arg)` later, and the argument will be `this`, which is the `Producer` itself.
   - `pthread_create()` will return 0 if the thread is created successfully.

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

1. `Producer::process(void* arg)`
   - In this function, producer will do all its jobs.
   - Get the item from `input_queue`, tranform, and put into `worker_queue`
   - This function will run a infinite loop.

```
void* Producer::process(void* arg) {
    // TODO: implements the Producer's work
    Producer* producer = (Producer*) arg;
    while(true){
        Item * it = producer->input_queue->dequeue();
        // std::cout << "Producing: " << *it << std::endl;
        auto val = producer->transformer->producer_transform(it->opcode, it->val);
        producer->worker_queue->enqueue(new Item(it->key, val, it->opcode));
        delete it;
    }
    return nullptr;
}
```

Consumer: in consumer.hpp

1. `Consumer::start()`
    - Fork a new thread. Same as we've done in `Producer::start()`

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

2. `Consumer::cancel()`
    - Set `is_cancel` to `true`, then call `pthread_cancel(t)` to terminate the thread.

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

3. `Consume::process(void* arg)`
    - `pthread_setcanceltype(PTHREAD_CANCEL_DEFERRED, nullptr)` means that a thread's cancelation will be postponed to a nearest cancel point.
    - `pthread_setcancelstate(PTHREAD_CANCEL_DISABLE, nullptr)` and `pthread_setcancelstate(PTHREAD_CANCEL_ENABLE, nullptr)` are used to determine whether the thread is in enable to be canceled.
    - The main job in this function is still simple.
    - Take the data from `worker_queue`, transform, and put the data into `output_queue`.

```
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* it = consumer->worker_queue->dequeue();
        // std::cout << "Consuming: " << *it << std::endl;
        auto val = consumer->transformer->consumer_transform(it->opcode, it->val);
// new value
        consumer->output_queue->enqueue(new Item(it->key, val, it->opcode));
        // std::cout << "output queue size = " << consumer->output_queue-
>get_size() << std::endl;
        delete it;
```

```
        pthread_setcancelstate(PTHREAD_CANCEL_ENABLE, nullptr);
    }

    return nullptr;
}
```

ConsumerController: in consumer_controller.hpp

1. `ConsumerController::start()`
    - The same.

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

2. `ConsumerController::process(void* arg)`
    - Get `ConsumerController` instance from the argument.
    - `usleep(ctr->check_period)` can make this function work periodically.
    - Separate into two cases, if the size of `worker_queue` is smaller than `high_threshold`, it will create a new consumer. If the size of `worker_queue` is greater than `low_threshold`, it will terminate a thread.

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

    // check every check_period microseconds
    // std::cout << "Consumer Controller processing" << std::endl;
    while(true){
        usleep(ctr->check_period);
        int wsize = ctr->worker_queue->get_size(), csize = ctr->consumers.size();
        // std::cout << "Writer queue Size = " << ctr->writer_queue->get_size() <<
std::endl;
        if (wsize < ctr->low_threshold) {
            // scale down
            if (csize > 1) {
                std::cout << "Scaling down consumers from " << csize << " to " <<
csize - 1 << std::endl;
                Consumer* consumer = ctr->consumers.back();
                consumer->cancel();
                consumer->join();
                ctr->consumers.pop_back();
                delete consumer;
            }
        } else if (wsize > ctr->high_threshold) {
            // scale up
            std::cout << "Scaling up consumers from " << csize << " to " << csize
```

```
+ 1 << std::endl;
            Consumer* consumer = new Consumer(ctr->worker_queue, ctr-
>writer_queue, ctr->transformer);
            ctr->consumers.push_back(consumer);
            ctr->consumers.back()->start();
        }
    }
    return (void*) nullptr;
}
```

Writer: in writer.hpp

1. `Writer::start()`
    - The same

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

2. `Writer::process(void* arg)`
    - This function's job is get the data from `output_queue` and put them into the output file.
    - But this function is not running in a infinite loop, so the thread will terminate once it writes all the data into the file.

```
void* Writer::process(void* arg) {
    // TODO: implements the Writer's work
    Writer* writer = (Writer*)arg;
    int count = 0;
    while (count < writer->expected_lines) {
        Item* it = writer->output_queue->dequeue();
        writer->ofs << *it << std::endl;
        // std::cout << "Writing: " << *it << std::endl;
        count++;
    }
    return nullptr;
}
```

main function: in main.cpp

1. In main function, we instantiate three queues, a transformer, 4 producers, a consumer_controller, a reader, and a writer.
2. Call `start()` for each of the mto create threads.
3. `start()` function will return immediately, but main function will terminate until reader and writer are both finished their works.

4. This approach is implemented by using `reader->join()` and `writer->join()`, these two functions will block main function until reader and writer finish.
5. reader and writer is not running in a infinite loop, so they can return if they have already read all the data in the input file or have written all the data into the output file.

```
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*> *reader_queue = new TSQueue<Item*>(READER_QUEUE_SIZE);
    TSQueue<Item*> *worker_queue = new TSQueue<Item*>(WORKER_QUEUE_SIZE);
    TSQueue<Item*> *writer_queue = new TSQueue<Item*>(WRITER_QUEUE_SIZE);
    Transformer *transformer = new Transformer();

    Producer **producers = new Producer *[PRODUCER_NUM];
    Reader *reader = new Reader(n, input_file_name, reader_queue);
    Writer *writer = new Writer(n, output_file_name, writer_queue);
    ConsumerController *consumer_controller = new ConsumerController(worker_queue,
writer_queue, transformer, CONSUMER_CONTROLLER_CHECK_PERIOD, WORKER_QUEUE_SIZE *
CONSUMER_CONTROLLER_LOW_THRESHOLD_PERCENTAGE / 100, WORKER_QUEUE_SIZE *
CONSUMER_CONTROLLER_HIGH_THRESHOLD_PERCENTAGE / 100);

    // std::cout << "Start" << std::endl;
    // std::cout << "Reader started" << std::endl;
    reader->start();
    // std::cout << "Writer started" << std::endl;
    writer->start();
    // std::cout << "Consumer Controller started" << std::endl;
    consumer_controller->start();
    for(int i=0; i<PRODUCER_NUM; i++){
        producers[i] = new Producer(reader_queue, worker_queue, transformer);
        // std::cout << "Producer " << i << " started" << std::endl;
        producers[i]->start();
    }

    // wait for all threads to finish
    // std::cout << "Waiting for reader to terminate" << std::endl;
    reader->join();
    // std::cout << "Waiting for writer to terminate" << std::endl;
    writer->join();

    delete reader;
    delete writer;

    return 0;
}
```

Experiment

Using testcase 01 to do the experiment

0. Result with default constants

- from 0 to 10 then back to 1, forms a cycle. Total 2.5 cycles.

```
[os24team45@localhost NTHU-OS-Pthreads]$ ./main 4000 ./tests/01.in ./tests/01.out
 Scaling up consumers from 0 to 1
 Scaling up consumers from 1 to 2
 Scaling up consumers from 2 to 3
 Scaling up consumers from 3 to 4
 Scaling up consumers from 4 to 5
 Scaling up consumers from 5 to 6
 Scaling up consumers from 6 to 7
 Scaling up consumers from 7 to 8
 Scaling up consumers from 8 to 9
 Scaling up consumers from 9 to 10
 Scaling down consumers from 10 to 9
 Scaling down consumers from 9 to 8
 Scaling down consumers from 8 to 7
 Scaling down consumers from 7 to 6
 Scaling down consumers from 6 to 5
 Scaling down consumers from 5 to 4
 Scaling down consumers from 4 to 3
 Scaling down consumers from 3 to 2
 Scaling down consumers from 2 to 1
 Scaling up consumers from 1 to 2
 Scaling up consumers from 2 to 3
 Scaling up consumers from 3 to 4
 Scaling up consumers from 4 to 5
 Scaling up consumers from 5 to 6
 Scaling up consumers from 6 to 7
 Scaling up consumers from 7 to 8
 Scaling up consumers from 8 to 9
 Scaling up consumers from 9 to 10
 Scaling down consumers from 10 to 9
 Scaling down consumers from 9 to 8
 Scaling down consumers from 8 to 7
 Scaling down consumers from 7 to 6
 Scaling down consumers from 6 to 5
 Scaling down consumers from 5 to 4
 Scaling down consumers from 4 to 3
 Scaling down consumers from 3 to 2
 Scaling down consumers from 2 to 1
 Scaling up consumers from 1 to 2
 Scaling up consumers from 2 to 3
 Scaling up consumers from 3 to 4
 Scaling up consumers from 4 to 5
 Scaling up consumers from 5 to 6
 Scaling up consumers from 6 to 7
 Scaling up consumers from 7 to 8
 Scaling up consumers from 8 to 9
 Scaling up consumers from 9 to 10
```

1. Different values of `CONSUMER_CONTROLLER_CHECK_PERIOD`.

- Default: 1,000,000
- I tried two different cases, increase the value to 10,000,000 and decrease the value to 100,000
- If we increase the check period, it more possible to miss some chance for scaling. So the total number of scaling and the max number of consumers (4) are both lower than default.

```
[os24team45@localhost NTHU-OS-Pthreads]$ ./main 4000 ./tests/01.in ./tests/01.out
 Scaling up consumers from 0 to 1
 Scaling up consumers from 1 to 2
 Scaling up consumers from 2 to 3
 Scaling up consumers from 3 to 4
 Scaling down consumers from 4 to 3
 Scaling up consumers from 3 to 4
 Scaling down consumers from 4 to 3
 Scaling up consumers from 3 to 4
```

- On the other hand, if we decrease the check period, the number of scaling and the max number of consumers (15) are both higher than default. The result is too long so we just captured a part of it.

```
[os24team45@localhost NTHU-OS-Pthreads]$ ./main 4000 ./tests/01.in ./tests/01.out
Scaling up consumers from 0 to 1
Scaling up consumers from 1 to 2
Scaling up consumers from 2 to 3
Scaling up consumers from 3 to 4
Scaling up consumers from 4 to 5
Scaling up consumers from 5 to 6
Scaling up consumers from 6 to 7
Scaling up consumers from 7 to 8
Scaling up consumers from 8 to 9
Scaling up consumers from 9 to 10
Scaling up consumers from 10 to 11
Scaling up consumers from 11 to 12
Scaling up consumers from 12 to 13
Scaling up consumers from 13 to 14
Scaling up consumers from 14 to 15
Scaling down consumers from 15 to 14
Scaling down consumers from 14 to 13
Scaling down consumers from 13 to 12
Scaling down consumers from 12 to 11
Scaling down consumers from 11 to 10
Scaling down consumers from 10 to 9
Scaling down consumers from 9 to 8
Scaling down consumers from 8 to 7
Scaling down consumers from 7 to 6
Scaling down consumers from 6 to 5
Scaling down consumers from 5 to 4
Scaling down consumers from 4 to 3
Scaling down consumers from 3 to 2
Scaling down consumers from 2 to 1
Scaling up consumers from 1 to 2
Scaling up consumers from 2 to 3
Scaling up consumers from 3 to 4
Scaling up consumers from 4 to 5
Scaling up consumers from 5 to 6
Scaling up consumers from 6 to 7
Scaling up consumers from 7 to 8
Scaling down consumers from 8 to 7
Scaling down consumers from 7 to 6
Scaling down consumers from 6 to 5
Scaling down consumers from 5 to 4
```

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

- Default: (20, 80)
- Two cases, (10, 90) and (45, 55)
- In the first case, we set the low threshold to 10 and high threshold to 90, that is, expanding the intervals which will not activate the scaling.
- In this case, the result has no big difference with the default case, even though using (5, 95). But in first cycle, its max consumer count is slightly lower.
- We suppose that it's because we have very small space to change the value by extending the threshold interval.

```
[os24team45@localhost NTHU-OS-Pthreads]$ ./main 4000 ./tests/01.in ./tests/01.out
Scaling up consumers from 0 to 1
Scaling up consumers from 1 to 2
Scaling up consumers from 2 to 3
Scaling up consumers from 3 to 4
Scaling up consumers from 4 to 5
Scaling up consumers from 5 to 6
Scaling up consumers from 6 to 7
Scaling up consumers from 7 to 8
Scaling up consumers from 8 to 9
Scaling down consumers from 9 to 8
Scaling down consumers from 8 to 7
Scaling down consumers from 7 to 6
Scaling down consumers from 6 to 5
Scaling down consumers from 5 to 4
Scaling down consumers from 4 to 3
Scaling down consumers from 3 to 2
Scaling down consumers from 2 to 1
Scaling up consumers from 1 to 2
Scaling up consumers from 2 to 3
Scaling up consumers from 3 to 4
Scaling up consumers from 4 to 5
Scaling up consumers from 5 to 6
Scaling up consumers from 6 to 7
Scaling up consumers from 7 to 8
Scaling up consumers from 8 to 9
Scaling up consumers from 9 to 10
Scaling down consumers from 10 to 9
Scaling down consumers from 9 to 8
Scaling down consumers from 8 to 7
Scaling down consumers from 7 to 6
Scaling down consumers from 6 to 5
Scaling down consumers from 5 to 4
Scaling down consumers from 4 to 3
Scaling down consumers from 3 to 2
Scaling down consumers from 2 to 1
Scaling up consumers from 1 to 2
Scaling up consumers from 2 to 3
Scaling up consumers from 3 to 4
Scaling up consumers from 4 to 5
Scaling up consumers from 5 to 6
Scaling up consumers from 6 to 7
Scaling up consumers from 7 to 8
Scaling up consumers from 8 to 9
Scaling up consumers from 9 to 10
```

- In this case, we reduct the threshold interval which will not activate the scaling. But it still seems like not having a great effect.
- Eventhough we found that with threshold (45, 55), we do have a larger number of consumers and more scalings, it's still not very obvious.
- So we consider that changes on interval won't have significant effect on the file with only 4000 lines in total. Maybe a larger file can have better result.
- Since the result is too long, we just capture a part of it.

```
[os24team45@localhost NTHU-OS-Pthreads]$ ./main 4000 ./tests/01.in ./tests/01.out
 Scaling up consumers from 0 to 1
 Scaling up consumers from 1 to 2
 Scaling up consumers from 2 to 3
 Scaling up consumers from 3 to 4
 Scaling up consumers from 4 to 5
 Scaling up consumers from 5 to 6
 Scaling up consumers from 6 to 7
 Scaling up consumers from 7 to 8
 Scaling up consumers from 8 to 9
 Scaling up consumers from 9 to 10
 Scaling down consumers from 10 to 9
 Scaling down consumers from 9 to 8
 Scaling down consumers from 8 to 7
 Scaling down consumers from 7 to 6
 Scaling down consumers from 6 to 5
 Scaling down consumers from 5 to 4
 Scaling down consumers from 4 to 3
 Scaling down consumers from 3 to 2
 Scaling down consumers from 2 to 1
 Scaling up consumers from 1 to 2
 Scaling up consumers from 2 to 3
 Scaling up consumers from 3 to 4
 Scaling up consumers from 4 to 5
 Scaling up consumers from 5 to 6
 Scaling up consumers from 6 to 7
 Scaling up consumers from 7 to 8
 Scaling up consumers from 8 to 9
 Scaling up consumers from 9 to 10
 Scaling up consumers from 10 to 11
 Scaling down consumers from 11 to 10
 Scaling down consumers from 10 to 9
 Scaling down consumers from 9 to 8
 Scaling down consumers from 8 to 7
 Scaling down consumers from 7 to 6
```

3. Different values of `WORKER_QUEUE_SIZE`.

- Default: 200
- Two cases, 10 and 1000
- With smaller worker queue size, the actual range between two threshold (not percentage) are much smaller than default.
- So the scaling may be triggered more frequently.

```
Scaling up consumers from 7 to 8
Scaling down consumers from 8 to 7
Scaling down consumers from 7 to 6
Scaling down consumers from 6 to 5
Scaling down consumers from 5 to 4
Scaling down consumers from 4 to 3
Scaling down consumers from 3 to 2
Scaling down consumers from 2 to 1
Scaling up consumers from 1 to 2
Scaling up consumers from 2 to 3
Scaling down consumers from 3 to 2
Scaling down consumers from 2 to 1
Scaling up consumers from 1 to 2
Scaling up consumers from 2 to 3
Scaling down consumers from 3 to 2
Scaling up consumers from 2 to 3
Scaling up consumers from 3 to 4
Scaling up consumers from 4 to 5
Scaling up consumers from 5 to 6
Scaling up consumers from 6 to 7
Scaling up consumers from 7 to 8
Scaling up consumers from 8 to 9
Scaling down consumers from 9 to 8
Scaling down consumers from 8 to 7
Scaling down consumers from 7 to 6
Scaling down consumers from 6 to 5
```

- If with larger worker queue size, time lapse between scaling will be longer because it's harder to trigger scaling.

```
[os24team45@localhost NTHU-OS-Pthreads]$ ./main 4000 ./tests/01.in ./tests/01.out
 Scaling up consumers from 0 to 1
 Scaling up consumers from 1 to 2
 Scaling up consumers from 2 to 3
 Scaling up consumers from 3 to 4
 Scaling up consumers from 4 to 5
 Scaling up consumers from 5 to 6
 Scaling up consumers from 6 to 7
 Scaling up consumers from 7 to 8
 Scaling down consumers from 8 to 7
 Scaling down consumers from 7 to 6
 Scaling down consumers from 6 to 5
 Scaling down consumers from 5 to 4
 Scaling down consumers from 4 to 3
 Scaling down consumers from 3 to 2
 Scaling down consumers from 2 to 1
 Scaling up consumers from 1 to 2
 Scaling down consumers from 2 to 1
```

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

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

- Origin: (200, 4000)
- Experiment: (1, 4000), (200, 1), (1, 1)
- For these two questions, all the three results are similar to the default case, even the scaling time lapse.
- We guess it's because there are 4 producers, and the file output rate is fast enough, so the speed of read/write is fast enough to ignore the size of reader and writer queue.
- We've tried to print messages while reading and writing, and it turns out no difference between default case and small reader/writer queue cases.