CS342301: Operating System
Pthread
Team Number: 71

Team Members & Contributions

- Members: 108022138 楊宗諺、111062649 許峻源
- Contributions: 合力製作Pthread

Experiment

1. Different values of CONSUMER_CONTROLLER_CHECK_PERIOD

預設值1000000
Thread 的最大數量來到7條

```
[os22team71@localhost NTHU-OS-Pthreads]$ ./main 500 ./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 down consumers from 7 to 6
```

提高至5000000
因為檢查的間隔變長，所以新增的thread 數量變少

```
[os22team71@localhost NTHU-OS-Pthreads]$ ./main 500 ./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
```

減少至500000
因為檢查頻率變高, Thread 最大數量提昇到10條

```
[os22team71@localhost NTHU-OS-Pthreads]$ ./main 500 ./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
```

2. Different value of threshold

預設值 low:20, high: 80

```
[os22team71@localhost NTHU-OS-Pthreads]$ ./main 500 ./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 down consumers from 7 to 6
```

提高low_threshold: 60
結束時的thread數量從6變成5

```
[os22team71@localhost NTHU-OS-Pthreads]$ ./main 500 ./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 down consumers from 7 to 6
Scaling down consumers from 6 to 5
```

提高high_threshold: 95
跟原本的預設值結果一樣, 可見worker queue一直都保持在相當滿的狀態

```
[os22team71@localhost NTHU-OS-Pthreads]$ ./main 500 ./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 down consumers from 7 to 6
```

降低high threshold : 40
最大thread數量提高到8條, 且沒有scaling down

```
[os22team71@localhost NTHU-OS-Pthreads]$ ./main 500 ./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
```

3. Different values of worker queue size

預設值 size : 200

```
[os22team71@localhost NTHU-OS-Pthreads]$ ./main 500 ./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 down consumers from 7 to 6
```

降低到 size : 100
Thread 最大數量達到8條

```
[os22team71@localhost NTHU-OS-Pthreads]$ ./main 500 ./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
```

提高 size : 500
可以看到size 變大後, queue在最後的狀態相對空, 所以scaling down 多次發生

```
[os22team71@localhost NTHU-OS-Pthreads]$ ./main 500 ./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 down consumers from 3 to 2
Scaling down consumers from 2 to 1
```

4. WRITER_QUEUE_SIZE is very small

Writer queue size 從4000 變成 100, 結果相差無幾

```
[os22team71@localhost NTHU-OS-Pthreads]$ ./main 500 ./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 down consumers from 7 to 6
```

5. **READER_QUEUE_SIZE is very small**

   Reader queue size 從200 變成 10, 但執行的結果相差無幾

```
[os22team71@localhost NTHU-OS-Pthreads]$ ./main 500 ./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 down consumers from 7 to 6
```

Implementation

1. **Ts_queue**

   利用circular queue with array 的方式來實做ts_queue, 並且在enqueue與dequeue function 加上wait condition 和 signal condition, 在shared variable(如: size, head, tail) 加上mutex lock 確保一次只有一個thread 可以在 critical section內工作

   Enqueue

```
template <class T>
void TSQueue<T>::enqueue(T item) {
    pthread_mutex_lock(&mutex);
    while(is_full())
    {
        //printf("queue is full. waiting.\n");
        pthread_cond_wait(&cond_enqueue,&mutex);
    }
    //printf("queue \n");
    buffer[ tail%buffer_size ] = item;
    ++tail;
    ++size;
    pthread_cond_signal(&cond_dequeue);
    pthread_mutex_unlock(&mutex);
}
```

   Dequeue

```
template <class T>
T TSQueue<T>::dequeue() {

    T item;
    pthread_mutex_lock(&mutex);
    while(is_empty())
    {
        pthread_cond_wait(&cond_dequeue,&mutex);
    }
    item = buffer[head%buffer_size];
    //printf("dequeue \n");
    ++head;
    --size;
    pthread_cond_signal(&cond_enqueue);
    pthread_mutex_unlock(&mutex);
    return item;
}
```

2. **Producer**

只要start一個producer object, 就會產生一個producer thread, 將item 從reader queue 中 dequeue 出來, 並且利用transformer 產生新的value, 在把有著新value的item enqueue 到worker queue 中

```
void Producer::start() {
    pthread_create(&t,0,Producer::process,(void*)this);
}

void* Producer::process(void* arg) {
    Item* it;
    Producer* producer = (Producer*) arg;
    int new_val;
    while(1)
    {
        it=producer->input_queue->dequeue();
        new_val=producer->transformer->producer_transform(it->opcode,it->val);

        it->val=new_val;
        producer->worker_queue->enqueue(it);
    }
    return nullptr;
}
```

3. **Consumer**

在consumer 的cancel function中, 會將is_cancel設為true, 變且呼叫pthread_cancel function, 傳入此object的thread id

```
int Consumer::cancel() {
    is_cancel=true;
    pthread_cancel(this->t);
    //pthread_testcancel();
    return 1;
}
```

在consumer process 內, 只要is_cancel不為true, 除了基本的把worker queue 的item

transform後, 放入writer queue 中, 為了確保consumer 不會執行到一半被cancel, 所以執行前要先disable Cancel state, 直到完成作業後再enable cancel state, 並且設下一個cancel point, 來檢查是否收到cancel request, 有的話則會刪除此thread, 並delete consumer

```
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);
        Item *it;
        Consumer *consumer= (Consumer*)arg;
        int new_val;
        it=consumer->worker_queue->dequeue();
        new_val=consumer->transformer->consumer_transform(it->opcode,it->val);
        it->val=new_val;
        consumer->output_queue->enqueue(it);
        pthread_setcancelstate(PTHREAD_CANCEL_ENABLE, nullptr);
        pthread_testcancel();
    }

    delete consumer;

    return nullptr;
}
```

4. Consumer Controller

在consumer controller 的process中, 必須持續check worker queue的狀態, 每隔一個period 就check 一次, 計算出worker queue 的 item percentage, 如果超過high threshold 則新增一個consumer object, 加入到consumer vector 中, 並把它start ; 反之, 低於low threshold , 則將一個consumer 從vector 中 pop out 出來, 呼叫他的cancel function, 並把它delete

```
void* ConsumerController::process(void* arg) {
    ConsumerController *con_con = (ConsumerController*) arg;
    Consumer *consumer;
    double percentage =0;
    while(1)
    {
        double size=(double)con_con->worker_queue->get_size();
        percentage = 100*(size / con_con->worker_queue->get_max_size() );
        //printf("percentage of workerqueue: %f\n", percentage);
        if(percentage>=con_con->high_threshold)
        {
            consumer = new Consumer(con_con->worker_queue,con_con->writer_queue,con_con->transformer);
            printf("Scaling up consumers from %d ",con_con->consumers_vector.size());
            con_con->consumers_vector.push_back(consumer);
            printf("\b to %d\n",con_con->consumers_vector.size());
            con_con->consumers_vector.back()->start();
        }
        else if(con_con->consumers_vector.size()>1 && percentage<con_con->low_threshold)
        {
            printf("Scaling down consumers from %d ",con_con->consumers_vector.size());
            consumer = con_con->consumers_vector.back();
            con_con->consumers_vector.pop_back();
            printf("\b to %d\n",con_con->consumers_vector.size());
            consumer->cancel();
            delete consumer;
        }
        usleep(con_con->check_period);
    }

    return nullptr;
```

5. Writer

把item 從writer queue中dequeue出來, 利用item 的 operator “<<” 把data output 到 Output file 中

```
void* Writer::process(void* arg) {
    Item *it;
    Writer *writer=(Writer*)arg;
    while(writer->expected_lines--)
    {
        it=writer->output_queue->dequeue();
        writer->ofs << *it;
    }

    return nullptr;
}
```

6. main

首先將會需要用到的物件都initialize

```
TSQueue<Item*>* input_queue;
TSQueue<Item*>* worker_queue;
TSQueue<Item*>* output_queue;

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

Reader *reader = new Reader(n, input_file_name,input_queue);
Writer *writer = new Writer(n, output_file_name, output_queue);

Transformer *transformer = new Transformer;

ConsumerController *consumercontroller;
consumercontroller = new ConsumerController(
                worker_queue,
                output_queue,
                transformer,
                CONSUMER_CONTROLLER_CHECK_PERIOD,
                CONSUMER_CONTROLLER_LOW_THRESHOLD_PERCENTAGE,
                CONSUMER_CONTROLLER_HIGH_THRESHOLD_PERCENTAGE);

Producer *p1 =new Producer(input_queue, worker_queue, transformer);
Producer *p2 =new Producer(input_queue, worker_queue, transformer);
Producer *p3 =new Producer(input_queue, worker_queue, transformer);
Producer *p4 =new Producer(input_queue, worker_queue, transformer);
```

接著依序執行reader ,writer, producer, concumer_controller, 在最後要確保reader和writer執行結束後程式才能結束, 所以要呼叫join function, 等thread 都執行完畢後將各個物件delete

```
reader->start();
writer->start();

p1->start();
p2->start();
p3->start();
p4->start();

consumercontroller->start();

reader->join();
writer->join();
delete writer;
delete reader;
delete p1;
delete p2;
delete p3;
delete p4;
delete transformer;
delete consumercontroller;
return 0;
```

Another experiments

將worker queue的size 改為 1, thread 的數量變化變得較沒規律, 可能是因為check到 worker queue 時, 狀態非滿及空, 所以如同抽獎一般, 每check 一次就會有thread 數量上的變化