PHIẾU GIAO BÀI TẬP

BÀI 3 - KỸ THUẬT LẬP TRÌNH ĐA LUỒNG (MULTITHREADING)

MỤC LỤC

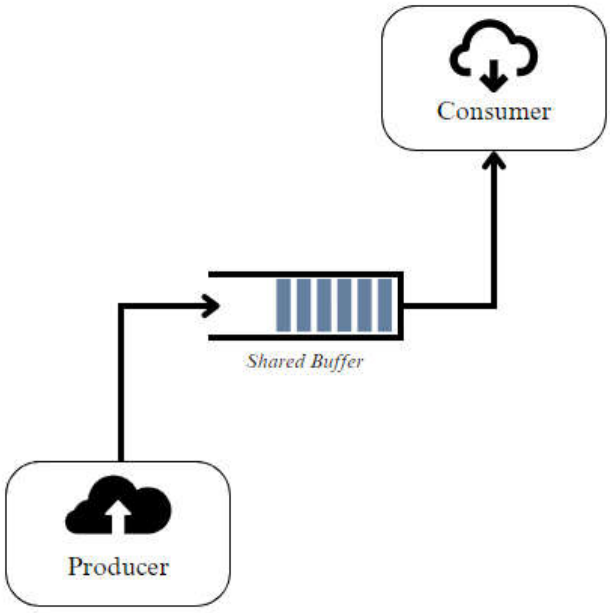
3.1 Bài toán Sản xuất – Tiêu dùng

3.1.1 Mô tả

Bài toán nhà sản xuất-người tiêu dùng (Producer-Consumer) là một bài toán đồng bộ hóa rất phổ biến.

Về vấn đề kỹ thuật bài toán nhà sản xuất-người tiêu dùng, có hai phân tuyến (Thread), một là *ProduceThread* có nhiệm vụ liên tục sản xuất các mặt hàng và thứ hai là *ConsumerThread* có nhiệm vụ liên tục tiêu thụ các mặt hàng.

Producer-Consumer chia sẻ cùng một bộ nhớ đệm có kích thước cố định trong đó nhà sản xuất đặt mặt hàng đã sản xuất và từ đó người tiêu dùng tiêu thụ mặt hàng đó.



Có nhiều vấn đề xảy ra với quy trình sản xuất và tiêu dùng theo khung nhìn kỹ thuật.

- ☞ Producer chỉ nên tạo dữ liệu khi bộ đệm chưa đầy. Nếu không, nó có thể khiến chương trình của chúng ta bị tràn bộ đệm (overflow).
- ☞ Consumer chỉ nên sử dụng dữ liệu khi bộ đệm không trống. Nếu không, nó có thể khiến chương trình của chúng ta rơi vào tình trạng rỗng bộ đệm (underflow).
- ☞ Producer và Consumer không nên truy cập bộ đệm cùng một thời điểm. Nếu không, khả năng xảy ra sự cố không nhất quán dữ liệu là rất cao.

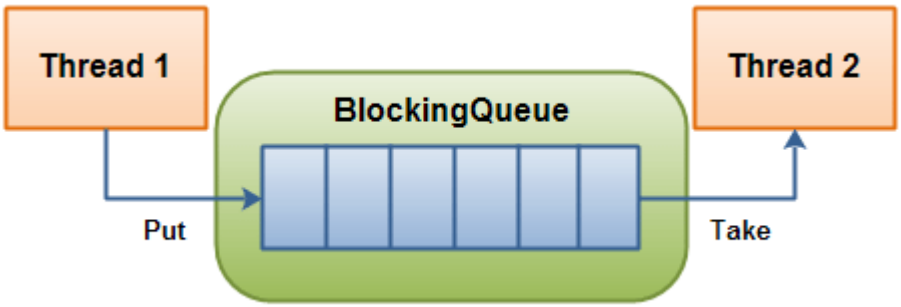
3.1.2 Với những vấn đề trên thì giải pháp có thể là gì?

- Đồng bộ hóa trên bộ đệm dùng chung trong khi sản xuất và tiêu thụ các mặt hàng.
- Khi bộ đệm trống, người tiêu dùng nên đợi cho đến khi nhà sản xuất sản xuất và thêm thứ gì đó vào bộ đệm.
- Khi bộ đệm đầy, nhà sản xuất phải đợi cho đến khi người tiêu dùng tiêu thụ thứ gì đó từ bộ đệm.

3.1.3 Cài đặt Giải pháp chương trình

Giải pháp cho vấn đề nhà sản xuất-người tiêu dùng này có thể được giải quyết dễ dàng trong Java bằng cách sử dụng BlockingQueue , dưới đây là kỹ thuật cài đặt sử dụng BlockingQueue.

Tạm minh họa cách thức hoạt động như hình vẽ sau



a) Lớp Producer chi tiết như sau

```
class Producer implements Runnable {
     protected BlockingQueue<Object> queue;
     Producer(BlockingQueue<Object> theQueue) {
          this.queue = theQueue;
     }
     public void run() {
          try {
               while (true) {
                    Object justProduced = getResource();
                    queue.put(justProduced);
                    System.out.println("Produced resource
- Kích thước bây giờ là "   + queue.size());
               }
          } catch (InterruptedException ex) {
               System.out.println("Producer
INTERRUPTED");
          }
     }
     Object getResource() {
```

```
            try {
                Thread.sleep(100); // mô phỏng 0,1 giây xử
lý
            } catch (InterruptedException ex) {
                System.out.println("Producer Read
INTERRUPTED");
            }
            return new Object();
        }
}
```

b) Lớp Consumer chi tiết như sau

```
class Consumer implements Runnable {
    protected BlockingQueue<Object> queue;
    Consumer(BlockingQueue<Object> theQueue) {
        this.queue = theQueue;
    }

    public void run() {
        try {
            while (true) {
                Object obj = queue.take();
                System.out.println("Consumed resource
- Queue size now = "   + queue.size());
                take(obj);
            }
        } catch (InterruptedException ex) {
            System.out.println("CONSUMER
INTERRUPTED");
        }
    }

    void take(Object obj) {
        try {
            Thread.sleep(100); // giả sử 0,1 giây lại
yêu cầu
        } catch (InterruptedException ex) {
            System.out.println("Consumer Read
INTERRUPTED");
        }
        System.out.println("Consuming object " + obj);
    }
}
```

c) Lớp ProducerConsumerExample thực thi

```
public class ProducerConsumerExample {
     public static void main(String[] args) throws
InterruptedException {
          int numProducers = 4;//Số lượng nhà sản xuất
          int numConsumers = 3;//Số lượng người tiêu dùng

          BlockingQueue<Object> myQueue = new
LinkedBlockingQueue<>(20);//Kích thước tối đa 20 sản
phẩm

          for (int i = 0; i < numProducers; i++){
               new Thread(new Producer(myQueue)).start();
          }

          for (int i = 0; i < numConsumers; i++){
               new Thread(new Consumer(myQueue)).start();
          }

          // Giả sử chờ đợi trong 10 giây
          Thread.sleep(10 * 1000);

          // Kết thúc
          System.exit(0);
     }
}
```

3.2 Bài toán Giao dịch ngân hàng

3.2.1 Mô tả bài toán

Giả lập một ngân hàng có 100 tài khoản. Xây dựng chương trình để có thể sinh ra các giao dịch chuyển tiền ngẫu nhiên giữa các tài khoản.

3.2.2 Phân tích

- Mỗi một tài khoản tương ứng với một Thread
- Giá trị chuyển khoản là ngẫu nhiên
- Xây dựng lớp Bank (ngân hàng) có phương thức thực thi chuyển khoản transfer()
- Phương thức transfer() sẽ được cài đặt synchronized

3.2.3 Mô tả chương trình và yêu cầu cài đặt

- **Lớp Bank gồm:**
    - Constructor để thiết lập số dư tài khoản ban đầu cho 100 tài khoản, khởi tạo số lượng giao dịch ban đầu = 0.

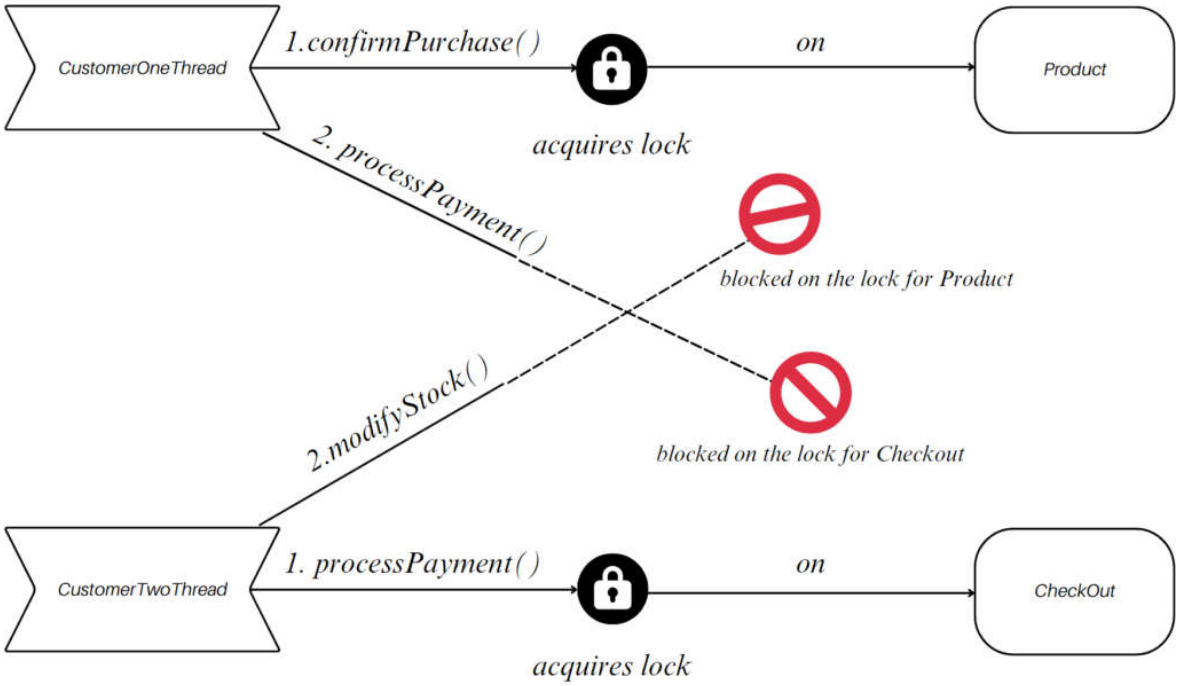
  - Phương thức transfer() - thực hiện chuyển một giá trị tài khoản từ (from) tài khoản này đến (to) tài khoản khác.
  - Phương thức test() - kiểm tra số lượng giao dịch với tổng lượng tiền là bao nhiêu.
  - Phương thức size() - lấy về số lượng tài khoản.
- **Lớp TransferThread:**
  - Là dẫn xuất từ Thread
  - Constructor thực hiện gán với mỗi tài khoản một Thread
  - Phương thức run() - thực hiện chuyển lượng tài khoản từ tài khoản (fromAccount) đến tài khoản khác (toAccount), giá trị chuyển khoản là ngẫu nhiên nhưng nhỏ hơn số dư ban đầu.
- **Chương trình chính:**
  - Xác định số tài khoản = 100
  - Xác định số dư tài khoản = 100000
  - Thiết lập Bank với các thông số trên
  - Tạo các phân tuyến tương ứng với các tài khoản
  - Thiết lập độ ưu tiên ngẫu nhiên (1-10) cho các phân tuyến
  - Thực thi các phân tuyến.

3.3 Bài toán Mua hàng và thanh toán

3.3.1 Yêu cầu Chương trình cài đặt gồm:

- Lớp Product xác nhận đặt sản phẩm và thay đổi kho lưu trữ
- Lớp Checkout thực hiện thanh toán
- Lớp CustomerOne và CustomerTwo minh họa cho 2 loại khách hàng cụ thể được cài đặt với Runnable
- Lớp DeadLockRealLifeExample chạy toàn bộ chương trình

3.3.2 Giải quyết kịch bản Deadlock được chỉ ra ở hình sau.



3.4 Bài toán tìm kiếm giá trị

Xây dựng chương trình tìm kiếm một giá trị khoá trong một mảng 1,000,000 phần tử. Mảng này được chia thành các đoạn và thực hiện bởi các phân tuyến. So sánh với các thuật toán tìm kiếm thông dụng.

3.5 Bài toán tính tổng số Nguyên tố

Xây dựng một phân tuyến tìm các số nguyên tố nằm trong khoảng 100-100,000. Phân tuyến thứ 2 tính tổng các số nguyên tố đó.

YÊU CẦU CHUNG:

- ☞ Chương trình cài đặt quản lý theo từng bài toán (theo từng package)
- ☞ Chuẩn bị dữ liệu đầy đủ và chạy demo
- ☞ Viết chú thích cho từng khối lệnh
- ☞ Viết báo cáo ngắn mô tả thuật toán và những kỹ thuật xử lý đối với Thread.