Java实现的生产者消费者问题模拟

PB17081531 沈鹏飞

算法介绍

信号量

在生产者消费者问题中，经常会遇到死锁问题。而解决这种问题的方法之一就是使用信号量。通过将信号量所需的所有算法封装成原子操作可以使得死锁问题得到解决，同时还会保证程序有一定的运行效率。

源码分析

在我实现的版本中：

1. 生产者生产的产品为随机整数
2. 缓存是一个存放整数的栈
3. 人为设定缓存的上限，并通过随时检查栈的大小来确保程序的正确运行
4. 打印当前缓存剩余容量和唤醒线程数量
5. 对buffer的读写锁通过统一的buffer对象的对象锁来实现，没有采用单独的mutex变量。

测试结果

原始版本

按照PPT上的伪代码进行构建，即先获取Buffer的信号量（full or empty），然后获取对buffer读写的锁（mutex），这样写出的代码通过了各种测试，包括打印出结果进行理论分析、多线程、超多线程并发长时间运行等，通过不输出来实现高频次运行，观察CPU占用来判断是否死锁，均没有问题。

交换获取资源信号量和资源锁的顺序

在上下文中，只需要把获取信号量的步骤放在syncronized内部即可。

经过这样的尝试，发现只要放在内部，程序会在运行开始不久就卡死，分析原因为：

只要在获取信号量时失败线程就会睡眠，并且不能释放mutex，此时其余的程序就无法获取锁，无法将该线程唤醒，即发生了死锁。（我在此实验中所使用的mutex是buffer的对象锁！并没有一个单独的mutex变量，但实际的效果并没有什么不同。）

源代码

主程序

```
package com.pnc;

import java.util.Stack;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
```

```
public class Main {

    public static int consumer_num=1;
    public static int producer_num=1;
    static int buffer_size=10;

    public static Stack<Integer> buffer=new Stack<Integer>();
    public static Semaphore empty=new Semaphore(buffer_size);
    public static Semaphore full=new Semaphore(0);

    public static void main(String args[])
    {
        ExecutorService consumerPool=
Executors.newFixedThreadPool(consumer_num);
        ExecutorService producerPool=
Executors.newFixedThreadPool(producer_num);
        for (int i=0;i<consumer_num;i++)
        {
            consumerPool.submit(new Consumer());
        }
        for (int i=0;i<producer_num;i++)
        {
            producerPool.submit(new Producer());
        }

    }
}
```

信号量

```
package com.pnc;

class Semaphore {

    Semaphore(int i){s =i;}

    int s;

    public int sleep_count=0;

    public synchronized void down() throws InterruptedException//通过syncronized
使down成为原子操作，可能会损失一定的性能，但是没有方法封装了sleep
    {
        if (s > 0) s--;
        else {
            sleep_count++;
            wait();
        }
    }

    public synchronized void up() {
        if (s == 0 && sleep_count > 0) {
            sleep_count--;
            notify();
        } else if (s == 0 && sleep_count == 0) {
            s++;
```

```
        } else s++;
    }

}
```

消费者

```
package com.pnc;
import java.util.Random;
import java.util.Stack;
import java.util.concurrent.atomic.AtomicInteger;

public class Consumer extends Thread {

    @Override
    public void run()
    {
        int item;
        while (true) {
            try {

                Main.full.down();
                item=remove_item(Main.buffer);

                Main.empty.up();
                consume_item(item);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }



    int remove_item(Stack<Integer> buffer) {
        synchronized (buffer) {
            return buffer.pop();
        }
    }

    void consume_item(int item) throws InterruptedException {
    }

}
```

生产者

```
package com.pnc;

import java.util.Random;
import java.util.Stack;

public class Producer extends Thread {

    Random r =new Random();
    @Override
```

```
    public void run()
    {
        int item;
        while (true) {
            try {
                item = generate_item();

                Main.empty.down();
                insert_item(item,Main.buffer);
                Main.full.up();

            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
    public void insert_item(int num, Stack<Integer> buffer)//对象锁，不会被remove抢走，相当于mutex的作用
    {
        synchronized (buffer)//由于获取的锁都是buffer的对象锁，因此就相当于有全局的mutex
        {
            buffer.push(num);

        }
    }

    int generate_item() throws InterruptedException {
        int item = r.nextInt();
        return item;
    }
}
```