生产者消费者问题

维基百科，自由的百科全书

生产者消费者问题（英語：Producer-consumer problem），也称**有限缓冲问题**（Bounded-buffer problem），是一个多进程同步问题的经典案例。该问题描述了共享固定大小缓冲区的两个进程——即所谓的“生产者”和“消费者”——在实际运行时会发生的问题。生产者的主要作用是生成一定量的数据放到缓冲区中，然后重复此过程。与此同时，消费者也在缓冲区消耗这些数据。该问题的关键就是要保证生产者不会在缓冲区满时加入数据，消费者也不会在缓冲区中空时消耗数据。

要解决该问题，就必须让生产者在缓冲区满时休眠（要么干脆就放弃数据），等到下次消费者消耗缓冲区中的数据的时候，生产者才能被唤醒，开始往缓冲区添加数据。同样，也可以让消费者在缓冲区空时进入休眠，等到生产者往缓冲区添加数据之后，再唤醒消费者。通常采用进程间通信的方法解决该问题，常用的方法有信号灯法[1]等。如果解决方法不够完善，则容易出现死锁的情况。出现死锁时，两个线程都会陷入休眠，等待对方唤醒自己。该问题也能被推广到多个生产者和消费者的情形。

目录

实现

不完善的实现

下面这个解决方法会导致竞争条件。如果程序员不够小心，那么他就有可能写出下面这种算法。该算法使用了两个系统库函数，sleep 和 wakeup。调用 sleep 的进程会被阻断，直到有另一个进程用 wakeup 唤醒之。代码中的 itemCount 用于记录缓冲区中的数据项数。

```
int itemCount = 0;

procedure producer() {
    while (true) {
        item = produceItem();
        if (itemCount == BUFFER_SIZE) {
            sleep();
        }
        putItemIntoBuffer(item);
        itemCount = itemCount + 1;
        if (itemCount == 1) {
            wakeup(consumer);
        }
    }
}

procedure consumer() {
    while (true) {
        if (itemCount == 0) {
            sleep();
        }
        item = removeItemFromBuffer();
        itemCount = itemCount - 1;
        if (itemCount == BUFFER_SIZE - 1) {
            wakeup(producer);
        }
        consumeItem(item);
    }
}
```

上面代码中的问题在于它可能导致竞争条件，进而引发死锁。考虑下面的情形：

1. 消费者把最后一个 itemCount 的内容读出来，注意它现在是零。消费者返回到while的起始处，现在进入 if 块；
2. 就在调用sleep之前，CPU决定将时间让给生产者，于是消费者在执行 sleep 之前就被中断了，生产者开始执行；
3. 生产者生产出一项数据后将其放入缓冲区，然后在 itemCount 上加 1；
4. 由于缓冲区在上一步加 1 之前为空，生产者尝试唤醒消费者；
5. 遗憾的是，消费者并没有在休眠，唤醒指令不起作用。当消费者恢复执行的时候，执行 sleep，一觉不醒。出现这种情况的原因在于，消费者只能被生产者在 itemCount 为 1 的情况下唤醒；
6. 生产者不停地循环执行，直到缓冲区满，随后进入休眠。

由于两个进程都进入了永远的休眠，死锁情况出现了。因此，该算法是不完善的。

使用信号灯的算法

信号灯可以避免上述唤醒指令不起作用的情况。该方法（见下面的代码）使用了两个信号灯，fillCount 和 emptyCount。fillCount 用于记录缓冲区中将被读取的数据项数（实际上就是有多少数据项在缓冲区里），emptyCount 用于记录缓冲区中空闲空间数。当有新数据项被放入缓冲区时，fillCount 增加，emptyCount 减少。如果在生产者尝试减少 emptyCount 的时候发现其值为零，那么生产者就进入休眠。等到有数据项被消耗，emptyCount 增加的时候，生产者才被唤醒。消费者的行为类似。

```
semaphore fillCount = 0; // 生产的项目
semaphore emptyCount = BUFFER_SIZE; // 剩余空间

procedure producer() {
    while (true) {
        item = produceItem();
        down(emptyCount);
            putItemIntoBuffer(item);
        up(fillCount);
    }
}

procedure consumer() {
    while (true) {
        down(fillCount);
            item = removeItemFromBuffer();
        up(emptyCount);
        consumeItem(item);
    }
}
```

上述方法在只有一个生产者和一个消费者时能解决问题。对于多个生产者或者多个消费者共享缓冲区的情况，该算法也会导致竞争条件，出现两个或以上的进程同时读或写同一个缓冲区槽的情况。为了说明这种情况是如何发生的，可以假设 putItemIntoBuffer() 的一种可能的实现：先寻找下一个可用空槽，然后写入数据项。下列情形是可能出现的：

1. 两个生产者都减少 emptyCount 的值；
2. 某一生产者寻找到下一个可用空槽；
3. 另一生产者也找到了下一个可用空槽，结果和上一步被找到的是同一个空槽；
4. 两个生产者向可用空槽写入数据。

为了解决这个问题，需要在保证同一时刻只有一个生产者能够执行 putItemIntoBuffer()。也就是说，需要寻找一种方法来互斥地执行临界区的代码。为了达到这个目的，可引入一个二值信号灯 mutex，其值只能为 1 或者 0。如果把线程放入 down(mutex) 和 up(mutex) 之间，就可以限制只有一个线程能被执行。多生产者、消费者的解决算法如下：

```
semaphore mutex = 1;
semaphore fillCount = 0;
semaphore emptyCount = BUFFER_SIZE;

procedure producer() {
    while (true) {
        item = produceItem();
        down(emptyCount);
            down(mutex);
                putItemIntoBuffer(item);
            up(mutex);
        up(fillCount);
    }
}
procedure consumer() {
    while (true) {
        down(fillCount);
            down(mutex);
                item = removeItemFromBuffer();
            up(mutex);
        up(emptyCount);
        consumeItem(item);
    }
}
```

使用管程的算法

下列伪代码展示的是使用管程来解决生产者消费者问题的办法。由于管程一定能保证互斥，不必特地考虑保护临界区[2]。也就是说，下面这个方法不用修改就可以推广适用于任意数量的生产者和消费者的情况。

```
monitor ProducerConsumer {
    int itemCount;
    condition full;
    condition empty;

    procedure add(item) {
        while (itemCount == BUFFER_SIZE)
            wait(full);
        putItemIntoBuffer(item);
        itemCount = itemCount + 1;
        if (itemCount == 1)
            notify(empty);
    }

    procedure remove() {
        while (itemCount == 0)
            wait(empty);
        item = removeItemFromBuffer();
        itemCount = itemCount - 1;
        if (itemCount == BUFFER_SIZE - 1)
            notify(full);
        return item;
    }
}

procedure producer() {
    while (true) {
        item = produceItem()
               ProducerConsumer.add(item)
    }
}

procedure consumer() {
    while (true) {
        item = ProducerConsumer.remove()
               consumeItem(item)
    }
}
```

注意代码中 while 语句的用法，都是用在测试缓冲区是否已满或空的时候。当存在多个消费者时，有可能造成竞争条件的情况是：某一消费者在一项数据被放入缓冲区中时被唤醒，但是另一消费者已经在管程上等待了一段时间并移除了这项数据。如果 while 语句被改成 if，则会出现放入缓冲区的数据项过多，或移除空缓冲区中的元素的情况。

不使用信号灯或者管程

对于生产者消费者问题来说，特别是当只有一个生产者和一个消费者时，实现一个先进先出结构或者通信通道非常重要。这样，生产者-消费者模式就可以在不依赖信号灯、互斥变量或管程的的情况下高效地传输数据。但如果采用这种模式，性能可能下降，因为实现这种模式的代价比较高。人们喜欢用先进先出结构或者通信通道，只是因为可以避免端与端之间的原子性同步。用 C 语言举例如下，请注意：

1. 该例绕开了对共享变量的原子性“读-改-写”访问：每个 Count 变量都由单进程更新；
2. 该例并不使进程休眠，这种做法依据系统不同是合理的。方法 sched_yield()只是为了看起来舒服点。完全可以去掉（注意：它后面的分号是不能去掉的）。 进程库通常会要求信号灯或者条

件变量控制进程的休眠和唤起，在多处理器环境中，进程的休眠和唤起发生的频率比传递数据符号要小，因此避开对数据原子性操作是有利的。

```
volatile unsigned int produceCount, consumeCount;
TokenType buffer[BUFFER_SIZE];

void producer(void) {
    while (1) {
        while (produceCount - consumeCount == BUFFER_SIZE)
            sched_yield(); // 缓冲区满
        buffer[produceCount % BUFFER_SIZE] = produceToken();
        produceCount += 1;
    }
}

void consumer(void) {
    while (1) {
        while (produceCount - consumeCount == 0)
            sched_yield(); // 缓冲区空
        consumeToken( buffer[consumeCount % BUFFER_SIZE]);
        consumeCount += 1;
    }
}
```

Java 中的例子

```java
import java.util.Stack;
import java.util.concurrent.atomic.AtomicInteger;

/**
    1个生产者 3个消费者 生产、消费10次

    @作者 pt

*/

public class ProducerConsumer {
    Stack<Integer> items = new Stack<Integer>();
    final static int NO_ITEMS = 10;

    public static void main(String args[]) {
        ProducerConsumer pc = new ProducerConsumer();
        Thread t1 = new Thread(pc.new Producer());
        Consumer consumer  = pc.new Consumer();
        Thread t2 = new Thread(consumer);
        Thread t3 = new Thread(consumer);
        Thread t4 = new Thread(consumer);
        t1.start();
        try {
            Thread.sleep(100);
        } catch (InterruptedException e1) {
            e1.printStackTrace();
        }
        t2.start();
        t3.start();
        t4.start();
        try {
            t2.join();
            t3.join();
            t4.join();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }

    class Producer implements Runnable {
        public void produce(int i) {
            System.out.println("Producing " + i);
            items.push(new Integer(i));
        }

        @Override
        public void run() {
            int i = 0;
            // 生产10次
            while (i++ < NO_ITEMS) {
                synchronized (items) {
                    produce(i);
                    items.notifyAll();
                }
                try {
                    // 休眠一段时间
                    Thread.sleep(10);
                } catch (InterruptedException e) {
                }
            }
        }
    }

    class Consumer implements Runnable {
        // consumed计数器允许线程停止
        AtomicInteger consumed = new AtomicInteger();

        public void consume() {
            if (!items.isEmpty()) {
                System.out.println("Consuming " + items.pop());
                consumed.incrementAndGet();
            }
        }

        private boolean theEnd() {
            return consumed.get() >= NO_ITEMS;
```

```
        }

        @Override
        public void run() {
            while (!theEnd()) {
                synchronized (items) {
                    while (items.isEmpty() && (!theEnd())) {
                        try {
                            items.wait(10);
                        } catch (InterruptedException e) {
                            Thread.interrupted();
                        }
                    }
                    consume();
                }
            }
        }
    }
```

相關條目

- 进程
- 进程间通信
- 多核处理器

参考资料

1. Modern Operating Systems (Third Edition)，Andrew S. Tanenbaum，2011. Prentice Hall
2. Operating System Concepts (Seventh Edition)，Abraham Silberschatz，2010. Wiley