



AARHUS UNIVERSITET

Software Engineering and Architecture

Producer Consumer



- The three categories of concurrent programs
 - Independent threads
 - Like running your Media player program while coding in IntelliJ
 - Shared resources
 - Like two threads reading/writing to the *same* account object
 - Typical case: web servers handling resources
 - You cannot book the same train seat twice!
 - **Collaborating processes**
 - Like one thread inserting into a buffer and assuming some other thread will remove those items from the buffer

Collaborating Threads

- The last, and most challenging, class of concurrent programs is *collaborating processes*
- The typical example is the *Producer Consumer*



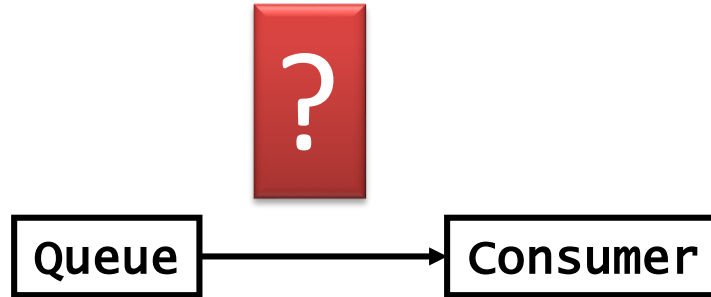
- For instance
 - Printer queue; high volume web traffic; disk read/write; ...

- Producer(s) and Consumer(s) are threads, and
- Our Queue (Buffer) class has methods
 - Synchronized void **put**(Object o) Insert 'o' into buffer
 - Synchronized Object **take**() Retrieve 'o' from buffer
- *Both must be critical regions - guarded by Lock or Synchronized*
- Consider the 'consumer' that wants to take the next item 'o' to process it
 - But if there is no item in the queue, it of course has to wait until there is an item available; that is, a producer has put something into the queue...



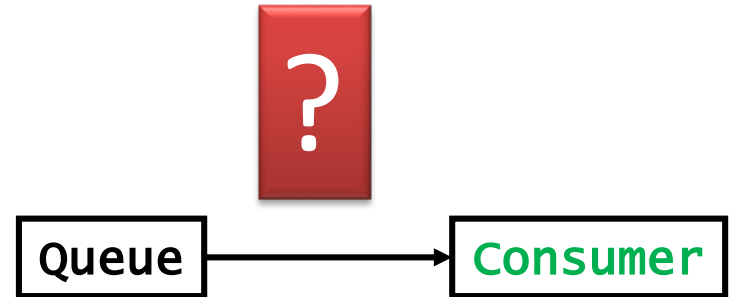
Waiting for item to be available

- Waiting for item 'o' to become available



Waiting for 'o'

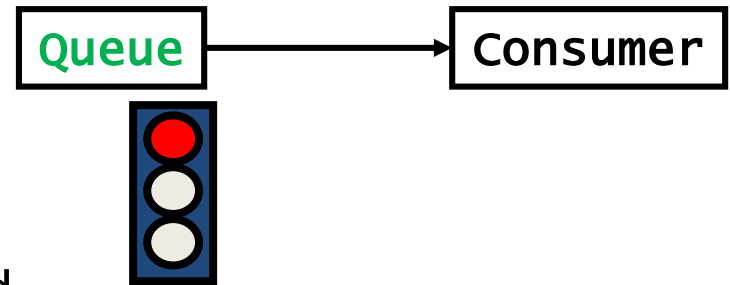
- So, how do we arrange to wait for the item?



- In the **consumer** code
 - Call 'take()' repeatedly until it returns a non-null value
 - *This is called polling (busy waiting), and wastes a lot of CPU cycles on nothing*
 - *... And there is a waiting time from item available to processing*
- Similar to picking up the phone every 1 minute to see if any has called you ...

Waiting for 'o'

- So, how do we arrange to wait for the item?



- In the `queue` code's `take()` method
 - Wait inside – but hey! It is a critical region and thus *no producer can ever enter the 'put()' method*

- ***Deadlock***

- *A thread waits infinitely for an event that will never happen*

- Repeat (as it is important!)
- We would *like* to wait in the queue code
 - Take() is called and then just returns when there is item available
- But we cannot because
 - Take() will take the 'lock' on the object and thus no other thread will ever be able to call the method put()
 - Waiting for the lock outside...

```
public synchronized void put(Object o) { ... }  
public synchronized Object take() { ... }
```



So We Want...

- ... a mechanism that
 - *awaits that a condition becomes true*
 - *Let other threads acquire the lock so they can make that condition true*

```
class Queue {  
    Object p;  
    boolean empty = true;  
  
    public synchronized void put(Object p) {  
        await (empty);  
        this.p = p; empty = false;  
    }  
  
    public synchronized Object take() {  
        await (!empty);  
        empty = true;  
        return p;  
    }  
}
```

Not Java code!



- Scenario
 - C call take()
 - Takes lock
 - Empty == true
 - Release lock and
 - Enter Waiting state
 - P calls put()
 - Takes lock
 - No wait (empty)
 - Finish and release lock
 - Later: Scheduler force P into ready state (not running)
 - C enters 'running' state
 - Takes lock
 - Empty = false!
 - Finish and release lock

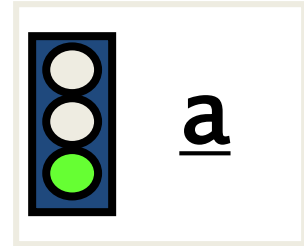
```
class Queue {
    Object p;
    boolean empty = true;

    public synchronized void put(Object p) {
        await (empty);
        this.p = p; empty = false;
    }

    public synchronized Object take() {
        await (!empty);
        empty = true;
        return p;
    }
}
```

Java Primitives (Java 1.4)

- Java objects maintain a *wait-set* in addition to the lock
 - `a.wait()` does *atomically*
 - Force current thread into waiting state,
 - Add current thread in object's wait-set
 - Release the lock on the object, `a`
 - `a.notify()` does
 - Choose one random thread, `T`, in `a`'s wait-set
 - `T` must take the lock on `a`
 - May fail if another thread has already taken the lock!
 - `T` resumes execution (becomes runnable) from the `wait()` statement
 - `a.notifyAll()` does
 - The same except *all* threads in `a`'s wait-set become 'runnable'...





Java 1.4 Code

```
class Channel {
    Object p;
    boolean empty = true;

    synchronized public void store(Object p) {
        while (!empty) {
            try { wait(); } catch (InterruptedException ie) {}
        }
        this.p = p; empty = false;
        notifyAll();
    }

    synchronized public Object retrieve() {
        while (empty) {
            try { wait(); } catch (InterruptedException ie) {}
        }
        empty = true;
        notifyAll(); return p;
    }
}
```



Java 1.4 Code

Why a loop around
wait() ???

```
class Channel {
    Object p;
    boolean empty = true;

    synchronized public void store(Object p) {
        while (!empty) {
            try { wait(); } catch (InterruptedException ie) {}
        }
        this.p = p; empty = false;
        notifyAll();
    }

    synchronized public Object retrieve() {
        while (empty) {
            try { wait(); } catch (InterruptedException ie) {}
        }
        empty = true;
        notifyAll(); return p;
    }
}
```

- The wait-set only makes sense inside a critical region
 - You cannot call 'wait()' or 'notify()' if you are not in a synchronize method / critical region
 - Will throw exceptions at your if you try...



```
csdev@m31: ~/proj/frsproject/producerconsumer-j14
File Edit Tabs Help
csdev@m31:~/proj/frsproject/producerconsumer-j14$ java ProducerConsumerMain
=== Using BufferJava14 ===
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40
41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78
79 80 81 82 83 84 85 86 87 88 ^Ccsdev@m31:~/proj/frsproject/producerconsumer-j14$
```

```
public static void main(String[] args) throws InterruptedException {
    // The buffer shared between the producer and consumer.
    Buffer b =
        new BufferJava14();
        // new BufferBlockingQueue();
        // new BufferJava15Lock();

    Thread producer = new Thread(new Producer(b));
    Thread consumer = new Thread(new Consumer(b));

    consumer.start();
    producer.start();
}
```



```
public synchronized void store(int item) throws InterruptedException {
    while(!empty) {
        // Wait to be notified of the buffer being empty.
        wait();
    }
    this.item = item;
    empty = false;
    notifyAll();
}

public synchronized int retrieve() throws InterruptedException {
    while(empty) {
        // Wait to be notified of an item becoming available.
        wait();
    }
    // Retrieve the item before we notify waiting threads.
    int item = this.item;
    empty = true;
    notifyAll();
    return item;
}
```




AARHUS UNIVERSITET

Java 5 Onwards

- Java was the first mainstream language to have internal threading
- Brink Hansen should have said that all his whole lifelong research into concurrency was a complete waste 😞
- Morale: It had to be improved...
 - Package: `java.util.concurrent`
 - Much more fine-grained concurrency control
 - A lot of default implementations without bugs!



Java 1.5 Code

```
class BufferJava15Lock implements Buffer {  
    Lock lock = new ReentrantLock();  
    Condition notEmpty = lock.newCondition();  
    Condition notFull = lock.newCondition();  
}
```

The Lock

Two *different* wait-sets associated...

```
public int retrieve() throws InterruptedException {  
    int returnvalue = -1;  
    lock.lock();  
    try {  
        while (!available) {  
            notEmpty.await();  
        }  
        // Retrieve the item before we notify waiting threads.  
        returnvalue = item;  
        available = false;  
        // Signal to waiting threads that queue is not full  
        notFull.signal();  
    } finally {  
        lock.unlock();  
    }  
    return returnvalue;  
}
```

Now, producers are waiting in one wait-set; while consumers are in another! We are sure to signal the right one!

```
public synchronized void store(int item) throws InterruptedException {  
    lock.lock();  
    try {  
        while (available) {  
            notFull.await();  
        }  
        // Fill it.  
        this.item = item;  
        available = true;  
        // Signal to waiting threads that queue is not empty  
        notEmpty.signal();  
    } finally {  
        lock.unlock();  
    }  
}
```

And Even More Easy!

- It is already implemented !

```
class BufferBlockingQueue implements Buffer {  
    private BlockingQueue<Integer> buffer =  
        new ArrayBlockingQueue<Integer>(1);
```

```
public int retrieve() throws InterruptedException {  
    return buffer.take();  
}  
  
public void store(int item) throws InterruptedException {  
    buffer.put(item);  
}
```



AARHUS UNIVERSITET

Moving On...



Vast Subject Area

- Lots of properties of concurrent programs
 - Liveliness
 - Fairness
 - Starvation
 - Deadlocks
 - Performance / blocked threads
 - Thread priority
- And library support
 - Java Collection classes are not thread safe ☹️
 - But *Decorators* exists
 - `List newList = Collections.synchronizedList(oldList);`

Vast Subject Area

- And Parallelism – the other side of concurrency
 - Java Stream processing
 - Runs concurrently if you use `parallelStream()`

- Map-Reduce

- Why not use 1.000 machines to compute 'f'?

- *And on, and on, and on...*

```

Collection
  ↓
db.orders.mapReduce(
  map   → function() { emit( this.cust_id, this.amount ); },
  reduce → function(key, values) { return Array.sum( values ); },
  query → { query: { status: "A" },
  output → out: "order_totals"
  )
  
```

