Lecture 25

Synchronization and Deadlock
We’ll revisit the Producer-Consumers example from the previous lecture, where two consumers read values from a queue written to by a producer.

We’ll see how data corruption can arise through time-sliced threads accessing the shared resource of the queue. (Which means that our implementation doesn’t encapsulate the ADT of queues.)

We’ll also see how to solve this problem by using a different form of the `synchronized` keyword: one which applies to methods in a class, rather than to a block of code.
Producer-Consumers

We’ll revisit the Producer-Consumers example from the previous lecture, where two consumers read values from a queue written to by a producer.

We’ll see how data corruption can arise through time-sliced threads accessing the shared resource of the queue. (Which means that our implementation doesn’t encapsulate the ADT of queues.)

We’ll also see how to solve this problem by using a different form of the `synchronized` keyword: one which applies to methods in a class, rather than to a block of code.
Producer-Consumers

We’ll revisit the Producer-Consumers example from the previous lecture, where two consumers read values from a queue written to by a producer.

We’ll see how data corruption can arise through time-sliced threads accessing the shared resource of the queue. (Which means that our implementation doesn’t encapsulate the ADT of queues.)

We’ll also see how to solve this problem by using a different form of the `synchronized` keyword: one which applies to methods in a class, rather than to a block of code.
Consumers and the Queue

The **ClosableQueue** object is a resource shared by two **Consumer** objects.

As an implementation of the abstract data type of queues, the **ClosableQueue** class should ensure that:

- all values put into the queue are passed on to one of the **Consumer** objects, and
- each value in the queue is passed on to *only one* of the **Consumer** objects.
Consumers and the Queue

The ClosableQueue object is a resource shared by two Consumer objects.

As an implementation of the abstract data type of queues, the ClosableQueue class should ensure that:

- all values put into the queue are passed on to one of the Consumer objects, and
- each value in the queue is passed on to only one of the Consumer objects.
Consumers and the Queue

The `ClosableQueue` object is a resource shared by two `Consumer` objects.

As an implementation of the abstract data type of queues, the `ClosableQueue` class should ensure that:

- all values put into the queue are passed on to one of the `Consumer` objects, and
- each value in the queue is passed on to *only one* of the `Consumer` objects.
Suppose the array storing the ClosableQueue values looks like this:

```
0 1 2 3 4 5
```

(For simplicity, we’ll just consider an array of size 6.) Values 0 and 1 have already been read by the Consumers.

Suppose now the Producer thread is running, and executes `qInputs.add(6).`
public void add(Integer i) {
    while (numItems > 6) {}  
    if (isClosed) return;
    if (endPoint >= 6) {
        for (int x = 0; x < numItems; x++) {
            items[x] = items[startPoint + x];
        }
        startPoint = 0;
        endPoint = numItems;
    }
    items[endPoint++] = i;
    numItems++;
}
Suppose the time-slicer halts this thread after shuffling the values down the array, but before executing `startPoint = 0`. The array now looks like this:

```
2 3 4 5 4 5
```

Suppose the time-slicer now starts a consumer thread, which reads three values (4, 5, and 4), leaving the array like this:

```
2 3 4 5 4 5
```
Suppose the time-slicer halts this thread after shuffling the values down the array, but before executing `startPoint = 0`. The array now looks like this:

\[
\begin{array}{cccccc}
2 & 3 & 4 & 5 & 4 & 5 \\
\uparrow & & & & \uparrow \\
\end{array}
\]

Suppose the time-slicer now starts a consumer thread, which reads three values (4, 5, and 4), leaving the array like this:

\[
\begin{array}{cccccc}
2 & 3 & 4 & 5 & 4 & 5 \\
\uparrow & & \uparrow \\
\end{array}
\]
If the time-slicer now goes back to the Producer thread, which executes the remainder of the `add(Integer)` method

```java
    startPoint = 0;
    endPoint = numItems;
}
items[endPoint++] = i;
numItems++;```

The array is now:

```
  2  3  4  5  4  5
↑  ↑
```

The array is now:

```
  2  6  4  5  4  5
↑  ↑
```
If the time-slicer now goes back to the Producer thread, which executes the remainder of the `add(Integer)` method

```java
    ...  
    startPoint = 0;
    endPoint = numItems;
    }
    items[endPoint++] = i;
    numItems++;
    }
```

The array is now:

```
2 3 4 5 4 5
↑  ↑
```

The array is now:

```
2 6 4 5 4 5 5
↑  ↑
```
If the time-slicer now goes back to the Producer thread, which executes the remainder of the `add(Integer)` method

```java
...
    startPoint = 0;
    endPoint = numItems;
}
    items[endPoint++] = i;
    numItems++;
}
```

The array is now:

2 3 4 5 4 5
↑ ↑

The array is now:

2 6 4 5 4 5 5
↑ ↑
Where Things Went Wrong

Clearly, the queue isn’t working properly. The numbers read by the Consumers so far are 0, 1, 4, 5, and 4, and the next values read will be 2 and 6. The number 4 has been read twice, and 3 has disappeared completely.

The problem is that time-slicing allowed accesses to the shared resource (the queue) to interfere with each other.

**Interference** is where time-slicing causes erroneous updates of a shared resource.
Where Things Went Wrong

Clearly, the queue isn’t working properly. The numbers read by the Consumers so far are 0, 1, 4, 5, and 4, and the next values read will be 2 and 6.

The number 4 has been read twice, and 3 has disappeared completely.

The problem is that time-slicing allowed accesses to the shared resource (the queue) to *interfere* with each other.

*Interference* is where time-slicing causes erroneous updates of a shared resource.
Monitors to the Rescue!

A general solution to the problem of interference is *mutual exclusion*.

*Mutual exclusion* ensures that only one thread can access a shared resource at a time, and that the access to the shared resource is allowed to terminate before any other thread can access the resource.

(I.e., accessing the shared resource is treated as a critical section.)
A general solution to the problem of interference is *mutual exclusion*.

**Mutual exclusion** ensures that only one thread can access a shared resource at a time, and that the access to the shared resource is allowed to terminate before any other thread can access the resource.

(I.e., accessing the shared resource is treated as a *critical section.*)
Locking the Queue

Monitors can be invoked by including the keyword `synchronized` in the declaration of a method.

If an object has one or more synchronized methods, the Java interpreter ensures that a thread can only call one of those synchronized methods when it has the key.

(Again, the Java interpreter does all the book-keeping involved in creating monitors; all the programmer does is put the keyword `synchronized` in the method declarations.)
Monitors can be invoked by including the keyword `synchronized` in the declaration of a method.

If an object has one or more synchronized methods, the Java interpreter ensures that a thread can only call one of those synchronized methods when it has the key.

(Again, the Java interpreter does all the book-keeping involved in creating monitors; all the programmer does is put the keyword `synchronized` in the method declarations.)
Locking the Queue

Monitors can be invoked by including the keyword `synchronized` in the declaration of a method.

If an object has one or more synchronized methods, the Java interpreter ensures that a thread can only call one of those synchronized methods when it has the key.

(Again, the Java interpreter does all the book-keeping involved in creating monitors; all the programmer does is put the keyword `synchronized` in the method declarations.)
Monitors

When an object belongs to a class with one or more synchronized methods, the monitor ensures mutual exclusion of calls on that object’s methods.

It does so by associating a ‘key’ to each object: before a thread can call any one of that object’s synchronized methods, it must obtain the key.
Monitors

Once a thread has the key, it keeps the key until execution of the synchronized method has finished, at which point the key is returned to the monitor so that other threads can call the monitored object’s synchronized methods.

If an object in a thread attempts to call a synchronized method from a monitored object whose key is not available, that thread is returned to the ready-pool of candidate threads.
Monitors

Once a thread has the key, it keeps the key until execution of the synchronized method has finished, at which point the key is returned to the monitor so that other threads can call the monitored object’s synchronized methods.

If an object in a thread attempts to call a synchronized method from a monitored object whose key is not available, that thread is returned to the ready-pool of candidate threads.
The key applies to \textit{all} the synchronized methods of an instance (i.e., each instance of the class has one key that is shared by all synchronized methods).

The requirement to obtain the key only applies to synchronized methods; threads can freely access any of the methods that are not declared to be \textit{synchronized}. 
Notes

- The key applies to *all* the synchronized methods of an instance (i.e., each instance of the class has one key that is shared by all synchronized methods).
- The requirement to obtain the key only applies to synchronized methods; threads can freely access any of the methods that are not declared to be *synchronized*. 
A Monitored Queue

We add the keyword `synchronized` to the `add()` and `next()` methods in the `ClosableQueue` class:

```java
public synchronized void add(Integer i) {
    ...
}
```
A Monitored Queue

in class ClosableQueue

```java
public synchronized Integer next() {
    ...
}
```

That is all we need do;

the Java interpreter will take care of installing a monitor when the code runs.
Synchronized Methods

Note that since the monitor key applies to both synchronized methods in the queue, this prevents possible interference that could arise from one thread calling `add()` while another thread calls `next()`, as in the example we looked at at the start of this lecture.
What Could Possibly Go Wrong?

There are two very general kinds of properties that ensure that interactions between threads don’t cause problems:

- **Safety Properties**: Bad things don’t happen
- **Liveness Properties**: Something good eventually happens
What Could Possibly Go Wrong?

There are two very general kinds of properties that ensure that interactions between threads don’t cause problems:

- **Safety Properties:**
  - Bad things don’t happen

- **Liveness Properties:**
  - Something good eventually happens
Interference and Deadlock

Absence of interference is an example of a Safety Property. For example, using synchronized methods ensures we have a ‘safe’ implementation of queues in the Producer-Consumer example.

Absence of Deadlock is a Liveness Property: Deadlock is where all threads are blocked while waiting for a resource to be freed, making further progress impossible.
Absence of interference is an example of a Safety Property. For example, using synchronized methods ensures we have a ‘safe’ implementation of queues in the Producer-Consumer example.

Absence of Deadlock is a Liveness Property:

Deadlock is where all threads are blocked while waiting for a resource to be freed, making further progress impossible.
Although synchronizing methods in the `ClosableQueue` class prevented interference, it does give rise to the possibility of deadlock.

Consider the following scenario: let us suppose that the queue is empty, and that in a thread `t1` the consumer `c1` obtains the key to the queue and calls the queue’s `next()` method.
public synchronized Integer next() {
    while (numItems == 0) {
        if (isClosed && numItems == 0) return null;
    }
    Integer n = items[startPoint++];
    numItems--;
    return n;
}

In t1, execution enters the ‘busy wait’: 
while (numItems == 0) { ... }
The idea of this busy wait is that eventually another thread will add a value to the queue.

However, $t_1$ now has the key to the queue, so any other thread trying to add a value to the queue will be blocked.

As a result, $t_1$ will remain in the busy wait forever, and all other threads that try to access the queue will remain blocked forever.

This is an example of **deadlock**.
The Dining Philosophers

A well-known example of deadlock arises in the so-called *Dining Philosophers Problem* (due to Tony Hoare).

Four philosophers are dining at a table. In the middle of the table is a bowl of spaghetti, and between each pair of philosophers is a fork. In order to eat, a philosopher must pick up two forks (to their left and right).
The Dining Philosophers

Each philosopher can pick up only one fork at a time, and when they pick up a fork they will wait for the other fork to become available.

Each philosopher will put down their forks only after they have eaten.
The Dining Philosophers

Suppose all the philosophers pick up the fork to their left at the same time; the forks to their right are unavailable (their neighbour has picked it up).
Now each philosopher will hold on to their single fork, waiting (in vain) for the other fork to become available.

The system is in **deadlock**.
Preventing Deadlock

Deadlock generally arises when all threads are waiting for some event to happen before they can make progress, but while they are waiting, some resource is tied up, preventing that event from happening.

One way of preventing deadlock is to allow processes to free resources (e.g., forks in the Dining Philosophers Problem, or the monitor key in the Queue-Consumer example).
Freeing Resources

Java provides methods `Object.wait()` and `Object.notify()` (or `notifyAll()`) that are intended to prevent deadlock by allowing threads to surrender monitor keys.

For each object with a monitor, the Java interpreter maintains a pool of threads that are waiting for some update to the monitored object. These are objects that have at some point gained the key to that object, but have surrendered that key while they wait for some update to take place.
Java provides methods `Object.wait()` and `Object.notify()` (or `notifyAll()`) that are intended to prevent deadlock by allowing threads to surrender monitor keys.

For each object with a monitor, the Java interpreter maintains a pool of threads that are waiting for some update to the monitored object. These are objects that have at some point gained the key to that object, but have surrendered that key while they wait for some update to take place.
Freeing Resources

We can think of these objects as ‘sleeping’ while they are in the pool. If an update to the monitored object takes place, they can be woken up (i.e., restored to the ready-pool of candidate threads waiting to run). When they are woken up, they need to re-aquire the monitor key before they can proceed running.

If they succeed in getting the key, they go back to the point of execution where they surrendered the key.
Freeing Resources

We can think of these objects as ‘sleeping’ while they are in the pool. If an update to the monitored object takes place, they can be woken up (i.e., restored to the ready-pool of candidate threads waiting to run). When they are woken up, they need to re-aquire the monitor key before they can proceed running.

If they succeed in getting the key, they go back to the point of execution where they surrendered the key.
The Wait and Notify Methods

The effects of these methods are as follows:

- **wait()**: halts execution of the thread and returns any monitor keys held by that thread. The thread is put in the ‘waiting’ pool.
- **notify()**: wakes up one (randomly-chosen) object in the waiting pool.
- **notifyAll()**: wakes up all of the objects in the waiting pool.
The Wait and Notify Methods

The effects of these methods are as follows:

- **wait()**: halts execution of the thread and returns any monitor keys held by that thread. The thread is put in the 'waiting' pool.
- **notify()**: wakes up one (randomly-chosen) object in the waiting pool.
- **notifyAll()**: wakes up all of the objects in the waiting pool.
The Wait and Notify Methods

The effects of these methods are as follows:

- **wait()**: halts execution of the thread and returns any monitor keys held by that thread. The thread is put in the ‘waiting’ pool.
- **notify()**: wakes up one (randomly-chosen) object in the waiting pool.
- **notifyAll()**: wakes up all of the objects in the waiting pool.
The Wait and Notify Methods

When a thread is taken out of the waiting pool (after a call of `notify()` or `notifyAll()`), it is put into the ready-pool.

When it is chosen to run, it requests the key it surrendered when it called `wait();`
if the key is not available, the thread goes back to the ready-pool;
if the key is available, the thread gets the key, and execution resumes at the point where the `wait()` method was called.
The Wait and Notify Methods

When a thread is taken out of the waiting pool (after a call of `notify()` or `notifyAll()`), it is put into the ready-pool.

When it is chosen to run, it requests the key it surrendered when it called `wait()`;
if the key is not available, the thread goes back to the ready-pool;
if the key is available, the thread gets the key, and execution resumes at the point where the `wait()` method was called.
The Wait and Notify Methods

When a thread is taken out of the waiting pool (after a call of `notify()` or `notifyAll()`), it is put into the ready-pool.

When it is chosen to run, it requests the key it surrendered when it called `wait()`;

if the key is not available, the thread goes back to the ready-pool;

if the key is available, the thread gets the key, and execution resumes at the point where the `wait()` method was called.
The Wait and Notify Methods

When a thread is taken out of the waiting pool (after a call of `notify()` or `notifyAll()`), it is put into the ready-pool.

When it is chosen to run, it requests the key it surrendered when it called `wait()`;
if the key is not available, the thread goes back to the ready-pool;
if the key is available, the thread gets the key, and execution resumes at the point where the `wait()` method was called.
The Wait and Notify Methods

Generally, the wait() method is called at the beginning of a synchronized method, and the notifyAll() method at the end of a synchronized method.

To prevent threads languishing in the waiting pool forever, every call of wait() should be matched by a subsequent call of notify() or notifyAll().
The Queue/Consumer Example

We replace the busy wait loops in the ClosableQueue class with calls of wait(), and add notifyAll() calls at the end of the synchronized methods:
public synchronized void add(Integer i) {
    while (numItems > 9) {
        try {
            wait();
        } catch (InterruptedException ie) { }
    }
    if (...) {
        ...
    }
    items[endPoint++] = i;
    numItems++;
    notifyAll();
}
public Integer next() {
    while (numItems <= 0) {
        if (isClosed & numItems <= 0)
            return null;
        try {
            wait();
        } catch (InterruptedException ie) { }
    }
    Integer i = items[startPoint++];
    numItems--;
    notifyAll();
    return i;
}
Safe, Live Queues

Synchronization ensures that only one thread accesses the queue at any given time. This gives us the mutual exclusion we need for Safety.

Calling `wait()` at the beginning of a synchronized method, and `notifyAll()` at the end means:

- we don’t compromise safety, and
- we prevent deadlock.
Safe, Live Queues

Synchronization ensures that only one thread accesses the queue at any given time. This gives us the mutual exclusion we need for Safety.

Calling `wait()` at the beginning of a synchronized method, and `notifyAll()` at the end means:

- we don’t compromise safety, and
- we prevent deadlock.
Thread States

Diagram:
- Waiting
- Ready
- Running
- Dead
- Blocked

Transitions:
- notify(): Waiting to Ready
- wait(): Waiting to Running
- sleep(): Running to Blocked

Actions:
- start(): Ready to Running

That’s All, Folks!

Summary

- Synchronized methods
- Deadlock
- Waiting Pool

Next:

Invariants