Concurrent Programming

Prof. Michele Loreti

Programmazione Avanzata
Corso di Laurea in Informatica (L31)
Scuola di Scienze e Tecnologie

Threading. . .

Threading is the creation and management of multiple units of execution within a single process.

Threading. . .

Threading is the creation and management of multiple units of execution within a single process.

Threading is a significant source of programming error, through the introduction of data races and deadlocks.

Threading...

Threading is the creation and management of multiple units of execution within a single process.

Threading is a significant source of programming error, through the introduction of data races and deadlocks.

The topic of threading can—and indeed does—fill whole books. Those works tend to focus on the myriad interfaces in a particular threading library.

Threading. . .

Threading is the creation and management of multiple units of execution within a single process.

Threading is a significant source of programming error, through the introduction of data races and deadlocks.

The topic of threading can—and indeed does—fill whole books. Those works tend to focus on the myriad interfaces in a particular threading library.

While we will focus on basics of the Linux threading API.

Binaries, Processes and Threads. . .

Binaries are dormant programs residing on a storage medium, compiled to a format accessible by a given operating system and machine architecture, ready to execute but not yet in motion.

Binaries, Processes and Threads...

Binaries are dormant programs residing on a storage medium, compiled to a format accessible by a given operating system and machine architecture, ready to execute but not yet in motion.

Processes are the operating system abstraction representing those binaries in action: the loaded binary, virtualised memory, kernel resources such as open files, an associated user, and so on.

Binaries, Processes and Threads. . .

Binaries are dormant programs residing on a storage medium, compiled to a format accessible by a given operating system and machine architecture, ready to execute but not yet in motion.

Processes are the operating system abstraction representing those binaries in action: the loaded binary, virtualised memory, kernel resources such as open files, an associated user, and so on.

Threads are the unit of execution within a process: a virtualised processor, a stack, and program state.

Binaries, Processes and Threads...

Binaries are dormant programs residing on a storage medium, compiled to a format accessible by a given operating system and machine architecture, ready to execute but not yet in motion.

Processes are the operating system abstraction representing those binaries in action: the loaded binary, virtualised memory, kernel resources such as open files, an associated user, and so on.

Threads are the unit of execution within a process: a virtualised processor, a stack, and program state.

Processes are running binaries and threads are the smallest unit of execution schedulable by an operating system's process scheduler.

Threading. . .

A process contains one or more threads.

Threading...

A process contains one or more threads.

If a process contains but one thread, there is only a single unit of execution in the process and only one thing going on at a time. We call such processes single threaded.

Threading. . .

A process contains one or more threads.

If a process contains but one thread, there is only a single unit of execution in the process and only one thing going on at a time. We call such processes single threaded.

If a process contains more than one thread, then there is more than one thing going on at once. We call such processes multithreaded.

Concurrent programming. . .

The first step to develop concurrent programs is to split activities in task.

Concurrent programming...

The first step to develop concurrent programs is to split activities in task.

In `Java` the interface Runnable it is used to describe a task that we want to run perhaps concurrently with other tasks.

Concurrent programming. . .

The first step to develop concurrent programs is to split activities in task.

In `Java` the interface Runnable it is used to describe a task that we want to run perhaps concurrently with other tasks.

```
public intercace Runnable {
  void run();
}
```

Concurrent programming...

The first step to develop concurrent programs is to split activities in task.

In `Java` the interface Runnable it is used to describe a task that we want to run perhaps concurrently with other tasks.

```
public intercace Runnable {
  void run();
}
```

The run method is executed in thread.

A task can be executed:

- in a specifically created thread;
- via an executor.

Executor service scheduler…

`Java` API provides executor services that schedule and execute task, choosing the thread on which to run them:

Executor service scheduler. . .

`Java` API provides executor services that schedule and execute task, choosing the thread on which to run them:

```
Runnable task = () -> {...} //Task to be executed
ExecutorService executor = ...;
executor.execute(task);
```

Executor service scheduler...

Java API provides executor services that schedule and execute task, choosing the thread on which to run them:

```
Runnable task = () -> {...} //Task to be executed
ExecutorService executor = ...;
executor.execute(task);
```

The Executors class has factory methods for executor services with different scheduling processes:

Executor service scheduler...

`Java` API provides executor services that schedule and execute task, choosing the thread on which to run them:

```
Runnable task = () -> {...} //Task to be executed
ExecutorService executor = ...;
executor.execute(task);
```

The Executors class has factory methods for executor services with different scheduling processes:

```
static ExecutorService newCachedThreadPool()
static ExecutorService newFixedThreadPool(int nThreads)
static ExecutorService newWorkStealingPool()
```

Executor service scheduler. . .

`Java` API provides executor services that schedule and execute task, choosing the thread on which to run them:

```
Runnable task = () -> {...} //Task to be executed
ExecutorService executor = ...;
executor.execute(task);
```

The Executors class has factory methods for executor services with different scheduling processes:

```
static ExecutorService newCachedThreadPool()
static ExecutorService newFixedThreadPool(int nThreads)
static ExecutorService newWorkStealingPool()
```

A ThreadFactory can be passed to control the creation of new threads.

Example

```java
Runnable hellos = () -> {
  for( int i=0 ; i<1000 ; i++ ) {
    System.out.println("Hello "+i);
  }
};
Runnable goodbyes = () -> {
  for( int i=0 ; i<1000 ; i++ ) {
    System.out.println("Goodbye "+i);
  }
};
ExecutorService executor = Executors.newCachedThreadPool();
executor.execute(hellos);
executor.execute(goodbyes);
```

Futures. . .

An instance of Runnable executes a task, but it does not return a value.

Futures. . .

An instance of `Runnable` executes a task, but it does not return a value.

When we have to obtain a value from the task computation, interface `Callable<V>` can be used:

Futures. . .

An instance of Runnable executes a task, but it does not return a value.

When we have to obtain a value from the task computation, interface Callable<V> can be used:

```
public interface Callable<V> {
  V call() throws Exception;
}
```

Futures. . .

An instance of Runnable executes a task, but it does not return a value.

When we have to obtain a value from the task computation, interface Callable<V> can be used:

```
public interface Callable<V> {
  V call() throws Exception;
}
```

The call method can throw arbitrary exceptions which can be relayed to the code that obtains the result.

Future. . .

A `Callable` can be submitted to an executor service:

Future...

A `Callable` can be submitted to an executor service:

```
ExecutorService executor = ...;
Callable<V> task = ...;
Future<V> result = executor.submit(task);
```

Future...

A Callable can be submitted to an executor service:

```
ExecutorService executor = ...;
Callable<V> task = ...;
Future<V> result = executor.submit(task);
```

The result of this submission is a future, this is an object that represents a computation whose result will be available at some future time:

Future...

A `Callable` can be submitted to an executor service:

```
ExecutorService executor = ...;
Callable<V> task = ...;
Future<V> result = executor.submit(task);
```

The result of this submission is a future, this is an object that represents a computation whose result will be available at some future time:

```
V get() throws Interrupted Exception, ExecutionException

V get(long timeout, TimeUnit unit)
  throws Interrupted Exception, ExecutionException,
  TimeoutException

boolean cancel(boolean mayInterruptIfRunning)
boolean isCancelled()
boolean isDone()
```

Multiple tasks. . .

If we have to wait for the results of multiple tasks, method `invokeAll`, that takes a `Collection` of `Callable`, can be used:

Multiple tasks. . .

If we have to wait for the results of multiple tasks, method `invokeAll`, that takes a `Collection` of `Callable`, can be used:

```
List<Callable<V>> tasks = ...
List<Future<V>> results = executor.invokeAll(tasks);
```

Multiple tasks...

If we have to wait for the results of multiple tasks, method invokeAll, that takes a Collection of Callable, can be used:

```
List<Callable<V>> tasks = ...
List<Future<V>> results = executor.invokeAll(tasks);
```

The execution of current thread is blocked until all the tasks have terminated (either successfully or unsuccessfully).

Multiple tasks. . .

If we have to wait for the results of multiple tasks, method `invokeAll`, that takes a `Collection` of `Callable`, can be used:

```
List<Callable<V>> tasks = ...
List<Future<V>> results = executor.invokeAll(tasks);
```

The execution of current thread is blocked until all the tasks have terminated (either successfully or unsuccessfully).

Another option we can use when we have to work on multiple tasks is `invokeAny`. In this case the result of the first (successfully) terminating task is returned. Other tasks are cancelled.

A lot of work is done by the `ExecutorService` that is responsible for execution and coordination of tasks!

Asynchronous computations

Asynchronous computations

When we have a `Future`, we need to call `get` to obtain the result and block the computation until it is available.

Asynchronous computations

When we have a `Future`, we need to call `get` to obtain the result and block the computation until it is available.

The use of `CompletableFuture` allow us to register a callback that is invoked (in some thread) with the result once it is available:

Asynchronous computations

When we have a Future, we need to call get to obtain the result and block the computation until it is available.

The use of CompletableFuture allow us to register a callback that is invoked (in some thread) with the result once it is available:

```
CompletableFuture<V> f = ...;
f.thenAccept( (V v) -> process results );
```

Asynchronous computations

When we have a Future, we need to call get to obtain the result and block the computation until it is available.

The use of CompletableFuture allow us to register a callback that is invoked (in some thread) with the result once it is available:

```
CompletableFuture<V> f = ...;
f.thenAccept( (V v) -> process results );
```

In this way the result is processed, without blocking, as soon as it is available!

Asynchronous computations

Asynchronous computations

To run a task asynchronously, (static) method supplyAsync can be used:

```
static <U> CompletableFuture<U> supplyAsync(
    Supplier<U> supplier , Executor executor )

static <U> CompletableFuture<U> supplyAsync(
     Supplier<U> supplier )
```

Asynchronous computations

To run a task asynchronously, (static) method supplyAsync can be used:

```
static <U> CompletableFuture<U> supplyAsync(
    Supplier<U> supplier , Executor executor )

static <U> CompletableFuture<U> supplyAsync(
     Supplier<U> supplier )
```

A CompletableFuture can complete in two ways:

- a result is computed;
- an exception is thrown.

Asynchronous computations

To run a task asynchronously, (static) method supplyAsync can be used:

```
static <U> CompletableFuture<U> supplyAsync(
    Supplier<U> supplier , Executor executor )

static <U> CompletableFuture<U> supplyAsync(
     Supplier<U> supplier )
```

A CompletableFuture can complete in two ways:

- a result is computed;
- an exception is thrown.

To handle termination, method whenComplete can be used:

```
public CompletableFuture<T> whenComplete(
  BiConsumer<? super T,? super Throwable> action
)
```

Asynchronous computations

The CompletableFuture is called completable because it can be manually completed.

Asynchronous computations

The `CompletableFuture` is called completable because it can be manually completed.

Method `complete` and `completeExceptionally` can be used to complete a future:

Asynchronous computations

The CompletableFuture is called completable because it can be manually completed.

Method complete and completeExceptionally can be used to complete a future:

```
boolean complete(T value)
```

```
boolean completeExceptionally(Throwable ex)
```

Asynchronous computations

The `CompletableFuture` is called completable because it can be manually completed.

Method `complete` and `completeExceptionally` can be used to complete a future:

```
boolean complete(T value)

boolean completeExceptionally(Throwable ex)
```

A future can be completed by multiple threads (only the first one is stored).

Composing futures...

Class CompletableFuture<T> provides a set of methods that can be used to process values in a chain:

```
<U> CompletableFuture<U> thenApply(
      Function<? super T,? extends U> fn)

CompletableFuture<Void> thenAccept(Consumer<? super T> fn)

<U> CompletableFuture<U> thenCompose(
    Function<? super T,? extends CompletionStage<U>> fn)

<U> CompletableFuture<U> handle(
    BiFunction<? super T,Throwable,? extends U> fn)

CompletableFuture<Void> thenRun(Runnable action)
```

Another example. . .

```
private static boolean done = false;

public static void main(String[] argv) {
  Runnable hellos = () -> {
    for( int i=0 ; i<1000 ; i++ ) {
      System.out.println("Hello "+i);
    }
    done = true;
  };
  Runnable goodbyes = () -> {
    int i=0;
    while (!done) { i++; }
    System.out.println("Goodbye "+i);
  };
  ExecutorService executor = Executors.newCachedThreadPool();
  executor.execute(hellos);
  executor.execute(goodbyes);
}
```

Visibility. . .

Rules. . .

By default, `Java` compiler performs optimisations while assuming that there are no concurrent memory access.

Visibility. . .

Rules. . .

By default, `Java` compiler performs optimisations while assuming that there are no concurrent memory access.

If there are, virtual machine has to know to avoid possible error.

Visibility. . .

Rules. . .

By default, `Java` compiler performs optimisations while assuming that there are no concurrent memory access.

If there are, virtual machine has to know to avoid possible error.

There are ways to ensure that an update to a variable is visible:

- The value of a final value is visible after initialisation;
- The initial value of a static variable is visible after static initialisation;
- Changes to volatile variables are visible;
- Changes happening before realising a lock are visible to anyone acquiring the lock.

Visibility. . .

Rules. . .

By default, `Java` compiler performs optimisations while assuming that there are no concurrent memory access.

If there are, virtual machine has to know to avoid possible error.

There are ways to ensure that an update to a variable is visible:

- The value of a `final` value is visible after initialisation;
- The initial value of a `static` variable is visible after static initialisation;
- Changes to `volatile` variables are visible;
- Changes happening before realising a lock are visible to anyone acquiring the lock.

To solve the problem in previous example, we have to declare `done` `volatile` .

Race condition. . .

```
private static volatile int count = 0;

public static void main(String[] argv) {
  ExecutorService executor = Executors.newCachedThreadPool();
  for( int i=0 ; i<100 ; i++ ) {
    int taskId = i;
    Runnable task =  () -> {
      for(int k=0 ; k<1000; k++) {
        count++;
      }
      System.out.println(taskId+": "+count);
    };
    executor.execute(task);
  }
}
```

Race condition

There are few strategies to try to tame race condition:

Race condition

There are few strategies to try to tame race condition:

Confinement: reduce the amount of shared data.

Race condition

There are few strategies to try to tame race condition:

Confinement: reduce the amount of shared data.

Immutability: share immutable objects.

Race condition

There are few strategies to try to tame race condition:

Confinement: reduce the amount of shared data.

Immutability: share immutable objects.

Critical Section/Locking: granting exclusive access to shared resource.

Synchronized blocks. . .

To guarantee that only a single thread executes some portions of code, block synchronized can be used:

Synchronized blocks...

To guarantee that only a single thread executes some portions of code, block synchronized can be used:

```
synchronize(value) {
  ...//Critical section
}
```

Synchronized blocks. . .

To guarantee that only a single thread executes some portions of code, block synchronized can be used:

```
synchronize(value) {
  ...//Critical section
}
```

In a synchronized block, object value act as a label.

Synchronized blocks...

To guarantee that only a single thread executes some portions of code, block synchronized can be used:

```
synchronize(value) {
  ...//Critical section
}
```

In a synchronized block, object value act as a label.

It is guaranteed that at most one thread is executing a synchronized block labelled with a given object o.

Synchronized blocks. . .

In a synchronized block, value `o` acts as a lock:

- lock is acquired when a thread enters in the block;
- lock is release when a thread exits from the block.

Synchronized blocks. . .

In a synchronized block, value `o` acts as a lock:

- lock is acquired when a thread enters in the block;
- lock is release when a thread exits from the block.

Example:

```
Runnable task =  () -> {
  for(int k=0 ; k<1000; k++) {
    synchronized (executor) {
      count++;
    }
  }
  System.out.println(taskId+": "+count);
};
```

Method synchronized. . .

A method can be declared synchronized:

Method synchronized. . .

A method can be declared synchronized:

```
public synchronized void increment() {
  count++;
}
```

Method synchronized...

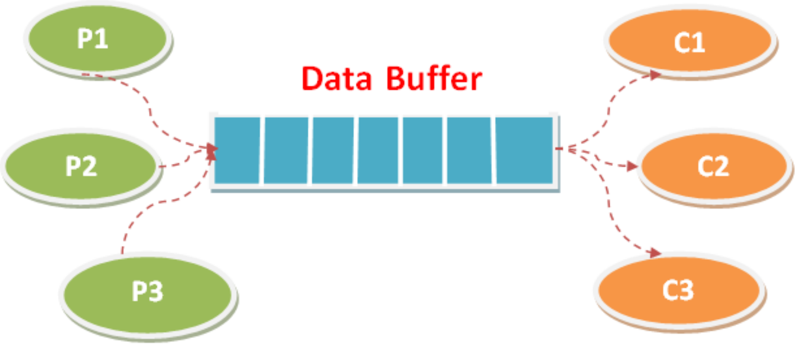
A method can be declared synchronized:

```
public synchronized void increment() {
  count++;
}
```

This code is equivalent to:

```
public void increment() {
  synchronized (this) {
    count++;
  }
}
```

Example: Producer/Consumer



Producer/Consumer

To implement the Producer/Consumer pattern we need a shared data structure with the following features:

- A method add that is used by the producer to store new items;
- A method get that is used by the receiver to store new items.

Producer/Consumer

To implement the Producer/Consumer pattern we need a shared data structure with the following features:

- A method add that is used by the producer to store new items;
- A method get that is used by the receiver to store new items.

What happen when there are not items to be collected?

Producer/Consumer

To implement the Producer/Consumer pattern we need a shared data structure with the following features:

- A method add that is used by the producer to store new items;
- A method get that is used by the receiver to store new items.

What happen when there are not items to be collected? Action get is blocking!

Producer/Consumer

To implement the Producer/Consumer pattern we need a shared data structure with the following features:

- A method add that is used by the producer to store new items;
- A method get that is used by the receiver to store new items.

What happen when there are not items to be collected? Action get is blocking!

What happen it the buffer is full?

Producer/Consumer

To implement the Producer/Consumer pattern we need a shared data structure with the following features:

- A method add that is used by the producer to store new items;
- A method get that is used by the receiver to store new items.

What happen when there are not items to be collected? Action get is blocking!

What happen it the buffer is full? Action add is blocking!

Producer/Consumer

To implement the Producer/Consumer pattern we need a shared data structure with the following features:

- A method add that is used by the producer to store new items;
- A method get that is used by the receiver to store new items.

What happen when there are not items to be collected? Action get is blocking!

What happen it the buffer is full? Action add is blocking!

What happen when a new item is inserted?

Producer/Consumer

To implement the Producer/Consumer pattern we need a shared data structure with the following features:

- A method add that is used by the producer to store new items;
- A method get that is used by the receiver to store new items.

What happen when there are not items to be collected? Action get is blocking!

What happen it the buffer is full? Action add is blocking!

What happen when a new item is inserted? Threads waiting for a new item are notified!

Producer/Consumer

To implement the Producer/Consumer pattern we need a shared data structure with the following features:

- A method add that is used by the producer to store new items;
- A method get that is used by the receiver to store new items.

What happen when there are not items to be collected? Action get is blocking!

What happen it the buffer is full? Action add is blocking!

What happen when a new item is inserted? Threads waiting for a new item are notified!

What happen when a new item is removed?

Producer/Consumer

To implement the Producer/Consumer pattern we need a shared data structure with the following features:

- A method add that is used by the producer to store new items;
- A method get that is used by the receiver to store new items.

What happen when there are not items to be collected? Action get is blocking!

What happen it the buffer is full? Action add is blocking!

What happen when a new item is inserted? Threads waiting for a new item are notified!

What happen when a new item is removed? Threads waiting for adding an item are notified!

Monitors

In concurrent programming, a monitor is a synchronisation construct that allows threads to have both mutual exclusion and the ability to wait (block) for a certain condition to become true.

Monitors

In concurrent programming, a monitor is a synchronisation construct that allows threads to have both mutual exclusion and the ability to wait (block) for a certain condition to become true.

Monitors also have a mechanism for signalling other threads that their condition has been met.

Monitors

In concurrent programming, a monitor is a synchronisation construct that allows threads to have both mutual exclusion and the ability to wait (block) for a certain condition to become true.

Monitors also have a mechanism for signalling other threads that their condition has been met.

A monitor consists of a mutex (lock) object and condition variables.

Monitors

In concurrent programming, a monitor is a synchronisation construct that allows threads to have both mutual exclusion and the ability to wait (block) for a certain condition to become true.

Monitors also have a mechanism for signalling other threads that their condition has been met.

A monitor consists of a mutex (lock) object and condition variables.

A condition variable is basically a container of threads that are waiting for a certain condition (thread's computation is suspended until the condition is satisfied).

Monitor in `Java`

Any object in `Java` can play the role of a monitor.

Monitor in `Java`

Any object in `Java` can play the role of a monitor.

To guarantee atomic executions of methods (that are the monitor's actions), these are declared synchronize.

Monitor in `Java`

Any object in `Java` can play the role of a monitor.

To guarantee atomic executions of methods (that are the monitor's actions), these are declared synchronize.

Each `Java` object provides methods that allow a thread to suspend its execution and then waiting for a notification!

Monitor in Java

Any object in Java can play the role of a monitor.

To guarantee atomic executions of methods (that are the monitor's actions), these are declared `synchronize`.

Each Java object provides methods that allow a thread to suspend its execution and then waiting for a notification!

These methods are:

- `void wait() throws InterruptedException`
- `void wait(long) throws InterruptedException`
- `notify ()`
- `notifyAll ()`

Producer/Consumer in Java

```
public class ProducerConsumer<T> {

  private final LinkedList<T> buffer;
  private final int size;

  public ProducerConsumer( int size ) {
    this.buffer = new LinkedList<>();
    this.size = size;
  }

  public synchronized boolean isEmpty() {
    return buffer.size()==0;
  }

  public synchronized boolean isFull() {
    return buffer.size()==size;
  }
```

Producer/Consumer in Java

```
  public synchronized void add(T item) throws
   InterruptedException {
    while (!this.isFull()) {
      wait();
    }
    this.notifyAll();
    buffer.add(item);
  }

  public T get() throws InterruptedException {
    while (!this.isEmpty()) {
      wait();
    }
    this.notifyAll();
    return buffer.poll();
  }
}
```

High Level Concurrency Objects

The Java Collections Framework provides high level data structures that simplifies concurrent programming.

High Level Concurrency Objects

The Java Collections Framework provides high level data structures that simplifies concurrent programming.

Lock objects support locking idioms that simplify many concurrent applications.

High Level Concurrency Objects

The Java Collections Framework provides high level data structures that simplifies concurrent programming.

Lock objects support locking idioms that simplify many concurrent applications.

Executors define a high-level API for launching and managing threads.

High Level Concurrency Objects

The Java Collections Framework provides high level data structures that simplifies concurrent programming.

Lock objects support locking idioms that simplify many concurrent applications.

Executors define a high-level API for launching and managing threads.

Concurrent collections make it easier to manage large collections of data, and can greatly reduce the need for synchronization.

High Level Concurrency Objects

The Java Collections Framework provides high level data structures that simplifies concurrent programming.

Lock objects support locking idioms that simplify many concurrent applications.

Executors define a high-level API for launching and managing threads.

Concurrent collections make it easier to manage large collections of data, and can greatly reduce the need for synchronization.

Atomic variables have features that minimize synchronization and help avoid memory consistency errors.

Lock Objects

Synchronized code relies on a simple kind of reentrant lock. This kind of lock is easy to use, but has many limitations.

Lock Objects

Synchronized code relies on a simple kind of reentrant lock. This kind of lock is easy to use, but has many limitations.

`Lock` objects work very much like the implicit locks used by synchronized code:

- only one thread can own a `Lock` object at a time;
- support a wait/notify mechanism, through their associated `Condition` objects.

Lock Objects

Synchronized code relies on a simple kind of reentrant lock. This kind of lock is easy to use, but has many limitations.

`Lock` objects work very much like the implicit locks used by synchronized code:

- only one thread can own a `Lock` object at a time;
- support a wait/notify mechanism, through their associated `Condition` objects.

The biggest advantage of `Lock` objects over implicit locks is their ability to back out of an attempt to acquire a lock:

- `tryLock` method backs out if the lock is not available immediately or before a timeout expires (if specified);
- `lockInterruptibly` method backs out if another thread sends an interrupt before the lock is acquired.

Lock Objects

`void lock()`, Acquires the lock.

`void lockInterruptibly ()`, Acquires the lock unless the current thread is interrupted.

`Condition newCondition()`, Returns a new Condition instance that is bound to this Lock instance.

`boolean tryLock()`, Acquires the lock only if it is free at the time of invocation.

`boolean tryLock(long time, TimeUnit unit)`, Acquires the lock if it is free within the given waiting time and the current thread has not been interrupted.

`void unlock()`, Releases the lock.

Conditions

`Condition` factors out the Object monitor methods (`wait`, `notify` and `notifyAll`) into distinct objects to give the effect of having multiple wait-sets per object, by combining them with the use of arbitrary `Lock` implementations.

Conditions

`Condition` factors out the Object monitor methods (`wait`, `notify` and `notifyAll`) into distinct objects to give the effect of having multiple wait-sets per object, by combining them with the use of arbitrary `Lock` implementations.

Conditions (also known as condition queues or condition variables) provide a means for one thread to suspend execution until notified by another thread that some state condition may now be true.

Conditions

Condition factors out the Object monitor methods (wait, notify and notifyAll) into distinct objects to give the effect of having multiple wait-sets per object, by combining them with the use of arbitrary Lock implementations.

Conditions (also known as condition queues or condition variables) provide a means for one thread to suspend execution until notified by another thread that some state condition may now be true.

The key property that waiting for a condition provides is that it atomically releases the associated lock and suspends the current thread, just like Object.wait.

Conditions

`Condition` factors out the Object monitor methods (`wait`, `notify` and `notifyAll`) into distinct objects to give the effect of having multiple wait-sets per object, by combining them with the use of arbitrary `Lock` implementations.

Conditions (also known as condition queues or condition variables) provide a means for one thread to suspend execution until notified by another thread that some state condition may now be true.

The key property that waiting for a condition provides is that it atomically releases the associated lock and suspends the current thread, just like `Object.wait`.

A Condition instance is intrinsically bound to a lock. To obtain a Condition instance for a particular Lock instance use its newCondition() method.

Producer/Consumer in Java

Lock based implementation

```
public class ProducerConsumerLock<T> {
  private final Lock lock = new ReentrantLock();
  private final Condition notFull = lock.newCondition();
  private final Condition notEmpty = lock.newCondition();
  private final LinkedList<T> buffer;
  private final int size;

  public ProducerConsumerLock( int size ) {
    this.buffer = new LinkedList<>();
    this.size = size;
  }

  public boolean isEmpty() {
    return buffer.size()==0;
  }

  public boolean isFull() {
    return buffer.size()==size;
  }
```

Condition

`void await()`, Causes the current thread to wait until it is signalled or interrupted.

`boolean await(long time, TimeUnit unit)`, Causes the current thread to wait until it is signalled or interrupted, or the specified waiting time elapses.

`long awaitNanos(long nanosTimeout)`, Causes the current thread to wait until it is signalled or interrupted, or the specified waiting time elapses.

`void awaitUninterruptibly ()`, Causes the current thread to wait until it is signalled.

`boolean awaitUntil (Date deadline)`, Causes the current thread to wait until it is signalled or interrupted, or the specified deadline elapses.

`void signal ()`, Wakes up one waiting thread.

`void signalAll ()`, Wakes up all waiting threads.

Producer/Consumer in Java

Lock based implementation

```
public void add(T item) throws InterruptedException {
  lock.lock();
  try {
    while (this.isFull()) {
      System.out.println("Buffer is full! Waiting for space
  ...");
      notFull.await();;
    }
    notEmpty.signal();
    buffer.add(item);
    System.out.println("Item added (size="+buffer.size()+")
 ");
  } finally {
    lock.unlock();
  }
}
```

Producer/Consumer in Java

Lock based implementation

```
public T get() throws InterruptedException {
  lock.lock();
  try {
    while (this.isEmpty()) {
      System.out.println("Buffer is empty! Waiting for an
 item...");
      notEmpty.await();;
    }
    notFull.signal();
    System.out.println("Item removed (size="+(buffer.size()
 -1)+")");
    return buffer.poll();
  } finally {
    lock.unlock();
  }
}
```

Atomic Variables

The `java.util.concurrent.atomic` package defines classes that support atomic operations on single variables.

Atomic Variables

The `java.util.concurrent.atomic` package defines classes that support atomic operations on single variables.

All classes have get and set methods that work like reads and writes on volatile variables.

Atomic Variables

The `java.util.concurrent.atomic` package defines classes that support atomic operations on single variables.

All classes have get and set methods that work like reads and writes on volatile variables.

The atomic compareAndSet method also has these memory consistency features, as do the simple atomic arithmetic methods that apply to integer atomic variables.

To be continued. . .