Java ConditionObject (Part 1)

Douglas C. Schmidt
d.schmidt@vanderbilt.edu
www.dre.vanderbilt.edu/~schmidt

Institute for Software
Integrated Systems
Vanderbilt University
Nashville, Tennessee, USA
Learning Objectives in this Part of the Lesson

- Understand what condition variables are & what pattern they implement

```java
Lock l = new Lock();
Condition cond = l.newCondition();
...
cond.await();
doOperationProcessing();
```
Learning Objectives in this Part of the Lesson

• Understand what condition variables are & what pattern they implement

```java
Lock l = new Lock()
Condition cond = l.newCondition()
...
l.lock()
while (conditionNotSatisfied())
    cond.await()
doOperationProcessing()
```

Condition variables are tricky, so you might want to rewatch this lesson & read the links carefully.
Overview of Condition Variables
Overview of Condition Variables

• A CV is a synchronizer that allows a thread to (repeatedly) suspend its execution until a condition becomes true

See blog.dcoles.net/2012/02/understanding-how-to-use-condition.html

Wheel of Pain – Conan the Barbarian
Overview of Condition Variables

• A CV is a synchronizer that allows a thread to (repeatedly) suspend its execution until a condition becomes true

• A thread whose execution is suspended on a CV is said to be “blocked” on the CV

Tree of Woe – Conan the Barbarian
Overview of Condition Variables

- A CV is implemented as a queue of threads that wait for certain condition(s) to become true

See en.wikipedia.org/wiki/Monitor_(synchronization)#Condition_variables
A CV is implemented as a queue of threads that wait for certain condition(s) to become true

- This queue of threads is known as the “wait set”

```java
Lock l = new Lock();
Condition cond = l.newCondition();
...
l.lock()
while (conditionNotSatisfied())
    cond.await()
doOperationProcessing()
```
Overview of Condition Variables

• CVs are often used when mutual exclusion alone is inadequate
Overview of Condition Variables

- CVs are often used when *mutual exclusion* alone is inadequate, e.g.
  - Inefficient use of resources
    - e.g., due to excessive “spinning”

See [en.wikipedia.org/wiki/Spinlock](en.wikipedia.org/wiki/Spinlock)
Overview of Condition Variables

- CVs are often used when *mutual exclusion* alone is inadequate, e.g.
  - Inefficient use of resources
  - Insufficient to ensure *coordination*
    - e.g., what to do when a thread encounters shared state that it can't do any work on (yet)
Overview of Condition Variables

- CVs are often used when *mutual exclusion* alone is inadequate, e.g.
  - Inefficient use of resources
  - Insufficient to ensure *coordination*
    - e.g., what to do when a thread encounters shared state that it can't do any work on (yet)

See [en.wikipedia.org/wiki/Waiting_for_Godot](en.wikipedia.org/wiki/Waiting_for_Godot)
Implementing Guarded Suspension with CVs
Implementing Guarded Suspension with CVs

- CVs are most often used to implement the *Guarded Suspension* pattern

See [en.wikipedia.org/wiki/GuardedSuspension](en.wikipedia.org/wiki/GuardedSuspension)
Implementing Guarded Suspension with CVs

- CVs are most often used to implement the *Guarded Suspension* pattern
- Applied to operations that can run only when certain things happen

```java
Lock l = new Lock()
Condition cond = l.newCondition()
...
l.lock()
while (conditionNotSatisfied())
    cond.await()
doOperationProcessing()
```
Implementing Guarded Suspension with CVs

• CVs are most often used to implement the Guarded Suspension pattern
  • Applied to operations that can run only when certain things happen, e.g.
    • a lock is acquired

```
Lock l = new Lock()
Condition cond = l.newCondition()
...
...
1.lock()
while (conditionNotSatisfied())
    cond.await()
doOperationProcessing()
```
CVs are most often used to implement the *Guarded Suspension* pattern.

Applied to operations that can run only when certain things happen, e.g.
- A lock is acquired
- A precondition is satisfied

```java
Lock l = new Lock()
Condition cond =
  l.newCondition()
...
  l.lock()
while (!conditionNotSatisfied())
  cond.await()
doOperationProcessing()
```
Implementing Guarded Suspension with CVs

- e.g., Thread$_1$ suspends its execution on a CV until Thread$_n$ notifies it that shared state it's waiting on may now be true

```java
Lock l = new Lock();
Condition cond = l.newCondition();
...
l.lock();
while (!conditionNotSatisfied()) {
    cond.await();
    doOperationProcessing();
}
```
Implementing Guarded Suspension with CVs

- e.g., Thread₁ suspends its execution on a CV until Threadₙ notifies it that shared state it's waiting on *may* now be true

```
Lock l = new Lock();
Condition cond = l.newCondition();
...
l.lock();
while (conditionNotSatisfied())
    cond.await();
doOperationProcessing();
```

Note the tentative nature of “may”...
Implementing Guarded Suspension with CVs

- e.g., Thread\(_1\) suspends its execution on a CV until Thread\(_n\) notifies it that shared state it's waiting on may now be true

```java
Lock l = new Lock()
Condition cond = l.newCondition()
...
l.lock()
while (conditionNotSatisfied())
  cond.await()
doOperationProcessing()
```

**First, a lock must be acquired.**
Implementing Guarded Suspension with CVs

- e.g., $\text{Thread}_1$ suspends its execution on a CV until $\text{Thread}_n$ notifies it that shared state it's waiting on *may* now be true.

Second, a condition is checked (in a loop) with the lock held.

```java
Lock l = new Lock()
Condition cond = l.newCondition()
...
l.lock()
while (!conditionNotSatisfied())
    cond.await()
doOperationProcessing()
```
Implementing Guarded Suspension with CVs

- e.g., Thread₁ suspends its execution on a CV until Threadₙ notifies it that shared state it's waiting on may now be true

The calling thread will block (possibly repeatedly) while the condition is not satisfied (await() atomically releases the lock)

```
Lock l = new Lock()
Condition cond = l.newCondition()
...
l.lock()
while (conditionNotSatisfied())
    cond.await()
doOperationProcessing()
```
Implementing Guarded Suspension with CVs

- e.g., Thread_1 suspends its execution on a CV until Thread_n notifies it that shared state it's waiting on may now be true

ConditionVariable

```java
Lock l = new Lock();
Condition cond = l.newCondition();
...
l.lock();
while (conditionNotSatisfied())
    cond.await()
doOperationProcessing();
```

After the condition is satisfied, the operation is performed (with lock still held)
Implementing Guarded Suspension with CVs

- e.g., Thread₁ suspends its execution on a CV until Threadₙ notifies it that shared state it's waiting on *may* now be true.

```
Lock l = new Lock()
Condition cond = l.newCondition()
...
l.lock()
while (conditionNotSatisfied())
    cond.await()
    doOperationProcessing()
```

Another thread can signal the condition when shared state may now be true.
Implementing Guarded Suspension with CVs

- E.g., Thread₁ suspends its execution on a CV until Threadₙ notifies it that shared state it's waiting on *may* now be true

```java
Lock l = new Lock()
Condition cond = l.newCondition()
...
l.lock()
while (conditionNotSatisfied())
    cond.await()
doOperationProcessing()
```

The lock is reacquired & the condition is rechecked in the loop

ConditionVariable

<table>
<thead>
<tr>
<th>Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>await()</td>
</tr>
<tr>
<td>signal()</td>
</tr>
<tr>
<td>signalAll()</td>
</tr>
</tbody>
</table>
A condition can be arbitrarily complex, e.g.
- A method call, an expression involving shared state, etc.

```java
Lock l = new Lock();
Condition cond = l.newCondition();
...
lock()
while (conditionNotSatisfied())
    cond.await()
doOperationProcessing()
```
Implementing Guarded Suspension with CVs

- A condition can be arbitrarily complex, e.g.
  - A method call, an expression involving shared state, etc.
- Any state shared between threads must therefore be protected by a lock associated with the CV

```
Lock l = new Lock();
Condition cond = l.newCondition();
...
...
l.lock()
while (conditionNotSatisfied())
  cond.await()
doOperationProcessing()
```
Implementing Guarded Suspension with CVs

- Waiting on a CV releases the lock & suspends the thread *atomically*

```
Lock l = new Lock()
Condition cond = l.newCondition()
...
while (conditionNotSatisfied())
    cond.await()
doOperationProcessing()
```

The lock is released when the thread is suspended on the CV
Implementing Guarded Suspension with CVs

- Waiting on a CV releases the lock & suspends the thread \textit{atomically}
- Thread\textsubscript{1} is suspended until Thread\textsubscript{n} signals the CV

Lock
\begin{itemize}
  \item lock()
  \item unlock()
\end{itemize}

ConditionVariable
\begin{itemize}
  \item await()
  \item signal()
  \item signalAll()
\end{itemize}

```
Lock l = new Lock();
Condition cond = l.newCondition();
...
  l.lock();
  while (conditionNotSatisfied())
    cond.await();
  doOperationProcessing();
```
Implementing Guarded Suspension with CVs

- Waiting on a CV releases the lock & suspends the thread *atomically*
- Thread$_1$ is suspended until Thread$_n$ signals the CV

```
cond.signal()
```

When a thread is signaled it wakes up & must re-acquire its associated lock

```
lock()
unlock()
```

```
T3  T2  T1
```

```
ConditionVariable

await()
signal()
signalAll()
```

```
Tn
```
Implementing Guarded Suspension with CVs

- Waiting on a CV releases the lock & suspends the thread *atomically*
- Thread\(_1\) is suspended until Thread\(_n\) signals the CV

```java
Lock l = new Lock();
Condition cond = l.newCondition();
...
l.lock()
while (conditionNotSatisfied())
    cond.await()
doOperationProcessing()
```

After lock is re-acquired the thread can reevaluate its condition to see if it’s true.
Implementing Guarded Suspension with CVs

- Waiting on a CV releases the lock & suspends the thread *atomically*
- Thread$_1$ is suspended until Thread$_n$ signals the CV

```java
Lock l = new Lock()
Condition cond = l.newCondition()
...
l.lock()
while (conditionNotSatisfied())
    cond.await()
doOperationProcessing()
```

*If condition is not true the thread must wait (which releases the lock atomically)*
Implementing Guarded Suspension with CVs

- Waiting on a CV releases the lock & suspends the thread *atomically*
- Thread\(_1\) is suspended until Thread\(_n\) signals the CV

```java
Lock l = new Lock();
Condition cond = l.newCondition();
...
l.lock();
while (conditionNotSatisfied())
    cond.await();
doOperationProcessing();
```

After lock is re-acquired & condition is true the operation can proceed (with lock held)
End of Java ConditionObject (Part 1)
Java ConditionObject (Part 2)

Douglas C. Schmidt
d.schmidt@vanderbilt.edu
www.dre.vanderbilt.edu/~schmidt

Institute for Software Integrated Systems
Vanderbilt University
Nashville, Tennessee, USA
Learning Objectives in this Part of the Lesson

- Understand what condition variables are & what pattern they implement
- Recognize how condition variables are often applied in practice
Learning Objectives in this Part of the Lesson

• Understand what condition variables are & what pattern they implement

• Recognize how condition variables are often applied in practice

• Be aware of a human known use of condition variables
Applying Condition Variables in Practice
Applying Condition Variables in Practice

- CVs are powerful, but can be hard to grok & apply correctly

See en.wikipedia.org/wiki/Grok
Applying Condition Variables in Practice

- CVs are powerful, but can be hard to grok & apply correctly, e.g.
- The protocol for using CVs involves several “moving parts”

CAUTION
BE ALERT!!
MOVING PARTS
CVs are powerful, but can be hard to grok & apply correctly, e.g.

- The protocol for using CVs involves several “moving parts”
- i.e., a condition variable & a lock
Applying Condition Variables in Practice

- CVs are powerful, but can be hard to grok & apply correctly, e.g.
  - The protocol for using CVs involves several “moving parts”
- The non-determinism of concurrency is tricky
Applying Condition Variables in Practice

- CVs are powerful, but can be hard to grok & apply correctly, e.g.
  - The protocol for using CVs involves several “moving parts”
  - The non-determinism of concurrency is tricky
  - i.e., a loop may be needed to ensure a resource is available

See stackoverflow.com/a/38313778
Applying Condition Variables in Practice

- CVs are therefore often not used directly by apps, but instead are “hidden” within other abstractions.
Applying Condition Variables in Practice

- CVs form the basis for higher-level synchronizers in Java

See docs.oracle.com/javase/8/docs/api/java/util/concurrent/locks/AbstractQueuedSynchronizer.ConditionObject.html
CVs form the basis for higher-level synchronizers in Java, e.g.

CVs are used in blocking queues & deques in java.util.concurrent* packages

See docs.oracle.com/javase/tutorial/collections/implementations/queue.html
Applying Condition Variables in Practice

- CVs form the basis for higher-level synchronizers in Java, e.g.
  - CVs are used in blocking queues & deques in java.util.concurrent* packages
  - e.g., ArrayBlockingQueue

See upcoming part 2 on “Java Built-in Monitor Objects”
Applying Condition Variables in Practice

- CVs form the basis for higher-level synchronizers in Java, e.g.
  - CVs are used in blocking queues & deques in java.util.concurrent* packages
- Java built-in monitor objects

See upcoming lesson on “Java Built-in Monitor Objects”
Applying Condition Variables in Practice

- CVs form the basis for higher-level synchronizers in Java, e.g.
  - CVs are used in blocking queues & deques in `java.util.concurrent` packages
  - Java built-in monitor objects
  - The *Monitor Object* pattern

See [www.dre.vanderbilt.edu/~schmidt/PDF/monitor.pdf](http://www.dre.vanderbilt.edu/~schmidt/PDF/monitor.pdf)
Human Known Use of Condition Variables
Human Known Uses of Condition Variables

- A human known use is a pizza delivery protocol
  - Must acquire both the pizza & the keys to deliver the pizza
End of Java ConditionObject (Part 2)