Progress® | Orbacus™

PROGRESS® ORBACUS®

JThreads/C++ Guide

Version 4.3.4, February 2010

BUSINESS MAKING PROGRESS™ PROGRESS SOFTWARE

Updated: February 3, 2010

Contents

Preface

The Orbacus Library

The Orbacus documentation library consists of the following books:

- Orbacus Guide
- FreeSSL for Orbacus Guide
- JThreads/C++ Guide (this book)
- Orbacus Notify Guide
- .NET Connector Programmer’s Guide

Orbacus Guide

This manual describes how Orbacus implements the CORBA standard, and describes how to develop and maintain code that uses the Orbacus ORB. This is the primary developer’s guide and reference for Orbacus.

FreeSSL for Orbacus Guide

This manual describes the FreeSSL plug-in, which enables secure communications using the Orbacus ORB in both Java and C++.

JThreads/C++ Guide

This manual describes JThreads/C++, which is a high-level thread abstraction library that gives C++ programmers the look and feel of Java threads.

Orbacus Notify Guide

This manual describes Orbacus Notify, an implementation of the Object Management Group’s Notification Service specification.

.NET Connector Programmer's Guide

This manual describes the Orbacus .NET Connector, which enables transparent communication between clients running in a Microsoft .NET environment and servers running in a CORBA environment.

Audience

Manuals in the Orbacus library are written for intermediate to advanced level programmers who are:

- Experienced with Java or C++ programming
- Familiar with the CORBA standard and its specifications

These manuals do not teach the CORBA specification or CORBA programming in general, which are prerequisite skills. These manuals concentrate on how Orbacus implements the CORBA standard.

Getting the Latest Version

The latest updates to the Orbacus documentation can be found at http://www.iona.com/support/docs.

Compare the version dates on the web page for your product version with the date printed on the copyright page of the PDF edition of the book you are reading.

Searching the Orbacus Library

You can search the online documentation by using the **Search** box at the top right of the documentation home page:

http://www.iona.com/support/docs

To search a particular library version, browse to the required index page, and use the **Search** box at the top right.

You can also search within a particular book. To search within a HTML version of a book, use the **Search** box at the top left of the page. To search within a PDF version of a book, in Adobe Acrobat, select **Edit**|**Find**, and enter your search text.

Additional Resources

If you need help with Orbacus or any other products, contact technical support:

http://web.progress.com/support

The Knowledge Base contains helpful articles written by experts about Orbacus and other products. This is available from:

http://web.progress.com/en/orbacus/orbacus-support.html

Document Conventions

Typographical conventions

This book uses the following typographical conventions:

`Fixed width`	Fixed width (Courier font) in normal text represents portions of code and literal names of items such as classes, functions, variables, and data structures. For example, text might refer to the `IT_Bus::AnyType` class. Constant width paragraphs represent code examples or information a system displays on the screen. For example: `#include <stdio.h>`
`Fixed width italic`	Fixed width italic words or characters in code and commands represent variable values you must supply, such as arguments to commands or path names for your particular system. For example: `% cd /users/`*`YourUserName`*
Italic	Italic words in normal text represent *emphasis* and introduce *new terms*.
Bold	Bold words in normal text represent graphical user interface components such as menu commands and dialog boxes. For example: the **User Preferences** dialog.

Keying conventions

This book uses the following keying conventions:

No prompt	When a command's format is the same for multiple platforms, the command prompt is not shown.
`%`	A percent sign represents the UNIX command shell prompt for a command that does not require root privileges.
`#`	A number sign represents the UNIX command shell prompt for a command that requires root privileges.
`>`	The notation > represents the MS-DOS or Windows command prompt.

`...`
 `.`
 `.`
 `.`
	Horizontal or vertical ellipses in format and syntax descriptions indicate that material has been eliminated to simplify a discussion.

`[]`	Brackets enclose optional items in format and syntax descriptions.
`{}`	Braces enclose a list from which you must choose an item in format and syntax descriptions.
`|`	In format and syntax descriptions, a vertical bar separates items in a list of choices enclosed in `{}` (braces). In graphical user interface descriptions, a vertical bar separates menu commands (for example, select **File|Open**).

CHAPTER 1

Introduction to JThreads/C++

This chapter gives an overview of JThreads.

In this chapter

This chapter contains the following section:

Overview

What is JThreads/C++?

JThreads/C++ is the short-form of “Java-like Threads for C++”. JThreads/C++ is a high-level thread abstraction library that gives C++ programmers the look and feel of Java threads.

Java supports multi-threaded programming using the classes `java.lang.Thread` and `java.lang.ThreadGroup`, the interface `java.lang.Runnable`, and the `synchronized` keyword together with the methods `wait`, `notify` and `notifyAll` in `java.lang.Object`.

Let’s have a look how JThreads/C++ translates this to C++:

- The Java classes `java.lang.Thread` and `java.lang.ThreadGroup` are directly translated into the C++ classes `JTCThread` and `JTCThreadGroup`. The only difference is that the JThreads/C++ classes have `JTC` as a prefix instead of the Java package `java.lang`. The Java interface `java.lang.Runnable` is implemented as the abstract C++ class `JTCRunnable`, which contains the pure virtual method `run`.
- Support for the `synchronized` keyword is slightly more difficult, since it is not possible to add new keywords to C++. JThreads/C++ solves this using the classes `JTCMonitor` and `JTCSynchronized`. Instances of `JTCSynchronized` can be used as a replacement for the `synchronized` keyword, provided that an instance of `JTCMonitor` was created for the object to be synchronized. `JTCMonitor` also provides the methods `wait`, `notify` and `notifyAll`.

There are some features of Java’s thread model that are not implemented in JThreads/C++. These are:

- The security API. This is because some parts of the API simply can’t be implemented in C++. In general, this issue is not as important as in Java, since C++ is not used for Internet applications (applets) in the same way as Java.
- The thread control primitives `java.lang.Thread.stop`, `java.lang.Thread.suspend`, and `java.lang.Thread.resume` cannot be implemented with the same semantics as the Java thread model in a portable fashion. The WIN32 thread API supports primitives for these operations, but the POSIX thread API does not. In general, it is not a

good idea to use these API primitives as they exist in the Java thread model, for they can easily lead to deadlock situations.[1] These primitives are deprecated in JDK 1.2 [4], and therefore won't be supported in upcoming versions of Java.

About this document

This manual is not a substitute for a good thread programming book. This manual only describes how Java thread constructs translate to JThreads/C++.

There are excellent books available on Java thread programming, such as [2] and [3]. We highly recommend use of these books while learning JThreads/C++ programming. With the help of this manual it is easy to translate the examples provided there to JThreads/C++ programs.

1. In fact, the WIN32 programmers guide recommends against using `TerminateThread` (the API call to stop a thread's execution) since it can easily lead to application misbehavior.

CHAPTER 2

Hello World

We begin with the first program most programmers start with: a program that displays the text Hello World and then exits. However, our example is different from the typical Hello World program in that it is multi-threaded. That is, our version starts a new thread whose sole purpose is to print Hello World on the display.

In this chapter

This chapter contains the following sections:

Hello World in Java

In Java, this program can be written as:

```
 1 public class HelloWorld extends Thread
 2 {
 3     public void run()
 4     {
 5         System.out.println("Hello World");
 6     }
 7
 8     static public void main(String args[])
 9     {
10         Thread t = new HelloWorld();
11         t.start();
12     }
13 }
```

Line 1 A class `HelloWorld` is defined, extending the class `java.lang.Thread`.

Lines 3-6 A `run` method is defined, displaying “Hello World” on standard output.

Lines 8-12 A static `main` method is defined which creates an object of type `HelloWorld`. The `start` method is called which starts a new thread of execution. This thread then invokes the `run` method of the `HelloWorld` object.

Hello World in C++

Let's convert the Java program to a JThreads/C++ program:

```
 1 #include <JTC/JTC.h>
 2
 3 class HelloWorld : public JTCThread
 4 {
 5 public:
 6     virtual void run()
 7     {
 8         cout << "Hello World" << endl;
 9     }
10 };
11
12 int
13 main(int argc, char** argv)
14 {
15     JTCInitialize initialize;
16     JTCThread* t = new HelloWorld;
17     t -> start();
18     return 0;
19 }
```

Line 1 All JThreads/C++ programs must include the header file `JTC/JTC.h`, which contains (among other useful things) all of the necessary JThreads/C++ class definitions.

Line 3 Just like in the Java example, a class `HelloWorld` is defined. This class is derived from `JTCThread` instead of the Java equivalent `java.lang.Thread`.

Lines 6-9 A `run` method is defined which prints "Hello World" on standard output. `System.out` is replaced by the familiar C++ iostreams object `cout`.

Lines 12-19 A `main method` is defined, not as a static class member as in the Java example, but as the standard C++ global `main` function. `main` creates an object of type `HelloWorld` and calls the `start` method which starts a new thread of execution.

Note: You might think that the Hello World program has a memory leak because the thread object is created with `new` but never deleted with `delete`, but this is not the case. See "Reference Counting" on page 44 for more information.

The only other change is that the JThreads/C++ thread library must be initialized in `main`. This is done by creating an instance of the class `JTCInitialize`.

At first sight this application seems to indicate a problem. Can the application terminate due to return from `main` before the thread gets a chance to run? The answer is No, because the destructor for `JTCInitialize` doesn't return until all of the threads have terminated. The `JTCInitialize` destructor allows JThreads/C++ applications to have the same behavior as multi-threaded Java applications.

Hello World with Runnable

Java provides the `Runnable` interface, so that an application developer may use threads without using inheritance. The JThreads/C++ equivalent of the `Runnable` interface is the class `JTCRunnable`.

The Hello World example using `Runnable` in Java looks like this:

```
 1 public class HelloWorld implements Runnable
 2 {
 3     public void run()
 4     {
 5         System.out.println("Hello World");
 6     }
 7
 8     static public void main(String[] args)
 9     {
10         Thread t = new Thread(new HelloWorld());
11         t.start();
12     }
13 }
```

Line 1 A class `HelloWorld` is declared that implements the interface `Runnable`.

Line 10 A new thread is created with a `Runnable` object as the parameter, which in this case is an instance of the `HelloWorld` class.

Line 11 The thread is started. Since the `Thread` object was created with a `Runnable` object parameter, the `run` method of this `Runnable` is invoked.

The Java version can be translated directly into a JThreads/C++ application as follows:

```
 1 #include <JTC/JTC.h>
 2
 3 class HelloWorld : public JTCRunnable
 4 {
 5 public:
 6     virtual void run()
 7     {
 8         cout << "Hello World" << endl;
 9     }
10 };
11
12 int
13 main(int argc, char** argv)
14 {
15     JTCInitialize initialize;
16     JTCThread* t = new JTCThread(new HelloWorld);
17     t -> start();
18     return 0;
19 }
```

Line 3 As in the Java example, the class `HelloWorld` inherits from the JThreads/C++ class `JTCRunnable`.

Line 16 Create a new thread, using a new instance of the `HelloWorld` class as the required `JTCRunnable` parameter.

Line 17]Start the new thread, which invokes the `run` method.

CHAPTER 3

Working With Threads

This chapter describes how JThreads/C++ implements Java Monitors used in multithreading.

In this chapter

This chapter contains the following sections:

Synchronization

Example

Let's write a plain C++ class, which can be used for the buffering of characters. This class defines the methods `addChar` and `writeBuffer`. `addChar` adds a character to an internal character buffer and `writeBuffer` prints the buffer contents on standard output:

```
class CharacterBuffer
{
    char* data_;
    int max_;
    int len_;

public:

    CharacterBuffer()
        : data_(0), len_(0), max_(0)
    {
    }

    ~CharacterBuffer()
    {
        delete[] data_;
    }

    void addChar(char c)
    {
        if(len_ == max_)
        {
            char* newData = new char[len_ + 128];
            memcpy(newData, data_, len_);
            delete[] data_;
            data_ = newData;
            max_ += 128;
        }
        data_[len_++] = c;
    }
void writeBuffer()
    {
        cout.write(data_, len_) << flush;
        len_ = 0;
    }
};
```

Lines 3-5 Several data members are defined:

- `data_` is a character pointer to the buffered characters.
- `max_` is the maximum length of the buffer pointed to by `data_`.
- `len_` is the current length of the buffer (the number of valid characters in the buffer pointed to by `data_`). `len_` must be less than or equal to `max_`.

Line 10 The constructor initializes the class data members `data_`, `max_` and `len_`.

Line 14 The destructor deletes `data_`, freeing the buffer memory.

Lines 21-28 If the buffer is full (that is, if `len_` is equal to `max_`), allocate more memory. This is done by allocating a new, larger character buffer, copying the existing buffer contents into the new buffer, deleting the old buffer and assigning the pointer to the new buffer to `data_`. Finally `max_` must be updated to reflect the new buffer size.

Line 29 A character is added to the buffer and `len_` is incremented by one.

Lines 32-36 The `writeBuffer` method prints `len_` characters from the buffer on standard output and then resets `len_` to zero.

Milt-threading environment

The above class works fine as long as there is only a single thread of execution, but it will not work properly in a multi-threaded environment.

For example, if two threads execute `addChar` simultaneously, things can easily go wrong. Let's assume that the first thread runs until after the `delete[] data_` statement has been executed. At this point the operating system switches from the execution of the first thread to the second. Since `max_` has not yet been incremented by the first thread, the second thread also enters the conditional and accesses the `data_` variable, which now points to memory already deleted by the first thread. This will most likely crash the program.

Monitors

To solve the above problem, Java uses a concept known as *monitors*. This is described in the following sections.

Thread Safe Version in Java

Thread-safe Java example

A thread-safe Java version of the code in the previous section can be written as follows:

```
 1 public class CharacterBuffer
 2 {
 3     private char[] data_ = null;
 4     private int len_ = 0;
 5
 6     synchronized public void addChar(char c)
 7     {
 8         if(data_ == null || len_ == data_.length)
 9         {
10             byte[] newData = new byte[len_+128];
11             if (data_ != null)
12                 System.arraycopy(data_, 0, newData, 0, len_);
13             data_ = newData;
14         }
15         data_[len_++] = c;
16     }
17
18     synchronized public void writeBuffer()
19     {
20         System.out.write(data_, 0, len_);
21         System.out.flush();
22         len_ = 0;
23     }
```

Lines 3-4 Two data members are defined:

- `data_` is a character array, which holds the buffered characters.
- `len_` is the current length of the buffer (the number of valid characters in the buffer pointed to by `data_`).

In contrast to the C++ version of this program, it's not necessary to have a `max_` data member, since `data_.length` can be used instead.

Lines 6-14 If no buffer has been created yet or if the buffer is full (that is, if `len_` is equal to `data_.length)` a new, larger buffer is allocated. This is similar to the C++ version.

Line 15 A character is added to the buffer and `len_` is incremented by one.

Lines 18-22 Like in the C++ example, the `writeBuffer` method prints `len_` characters from the buffer on standard output and then resets `len_` to zero.

What is different?

The only conceptual change to make the program thread-safe was to add the `synchronized` keyword to the definitions of `addChar` and `writeBuffer`. In Java every object implicitly has an associated monitor. On entry to a `synchronized` method, the monitor belonging to the object is locked, preventing other threads from entering any other synchronized method of the object. On exit, the monitor is unlocked, thus allowing access by other threads. This makes sure that the scenario described above won't ever arise, since it is impossible for two threads to enter the `addChar` method simultaneously.

Thread Safe Version in C++

Thread-safe C++ example

JThreads/C++ supports monitors with two classes: `JTCMonitor` and `JTCSynchronized`. The `JTCSynchronized` class uses the *initialization is acquisition* concept to acquire the monitor's lock. The associated monitor's lock is acquired on construction and released on destruction.

Here is the thread-safe C++ version of the example:

```
class CharacterBuffer : public JTCMonitor
{
    char* data_;
    int len_;
    int max_;

public:

    CharacterBuffer()
        : data_(0), len_(0), max_(0)
    {
    }

    ~CharacterBuffer()
    {
        delete[] data_;
    }

    void addChar(char c)
    {
        JTCSynchronized synchronized(*this);
        if (len_ >= max_)
        {
            char* newData = new char[len_+128];
            memcpy(newData, data_, len_);
            delete[] data_;
            data_ = newData;
            max_ += 128;
        }
        data_[len_++] = c;
    }

33     void writeBuffer()
34     {
35         JTCSynchronized synchronized(*this);
36         cout.write(data_, len_) << flush;
37         len_ = 0;
38     }
39 };
```

Line 1 The class `CharacterBuffer` is now derived from `JTCMonitor`. In Java this is not necessary, since all Java objects inherit implicitly from `java.lang.Object`, which provides the monitor functionality.

Lines 21, 35 The `addChar` and `writeBuffer` methods are now thread safe. Instead of declaring the operations as `synchronized` (as is done in Java), the functions first create an instance of `JTCSynchronized` with the `CharacterBuffer`'s monitor object as argument.

So all that has to be done to translate a thread-safe (that is, synchronized) Java class to a thread-safe JThreads/C++ class is to:

- Derive the class from `JTCMonitor`.
- Replace `synchronized` methods by methods which contain `JTCSynchronized synchronized(*this)` as the first statement in the function body.

That's quite easy, isn't it?

Block Synchronization

Code blocks

Java not only supports synchronized methods, but also synchronized code blocks.

For example, let’s assume that we want to write a thread class whose `run` method puts a string into a `CharacterBuffer` object using `addChar`. In Java, this could be written as follows.

```
 1 class Writer extends Thread
 2 {
 3     private CharacterBuffer buffer_;
 4     private String str_;
 5
 6     public Writer(CharacterBuffer buffer, String str)
 7     {
 8         buffer_ = buffer;
 9         str_ = str;
10     }
11
12     public void run()
13     {
14         for(int i = 0 ; i < str.length() ; i++)
15             buffer_.addChar(str_.characterAt(i));
16     }
17 };
```

Line 1 A `Writer` class is defined, which inherits from `Thread`.

Lines 6-10 The constructor initializes the `buffer_` and `str_` data members.

Lines 12-16 The thread’s `run` method puts the string `str_` into the buffer, character by character, using the buffer’s `addChar` method.

This class does not work as we want it to, however. Suppose we start two new threads, one to add “123” to the buffer and another one to add “abc”:

```
CharacterBuffer buffer = new CharacterBuffer();
JTCHandleT<Writer> w1 = new Writer(buffer, "123");
JTCHandleT<Writer> w2 = new Writer(buffer, "abc");
w1 -> start();
w2 -> start();
```

Line 1 A `CharacterBuffer` is created.

Lines 2, 3 Two `Writer` threads are created, one with “123” as argument, and the other with “abc”. Both threads use the same `CharacterBuffer` object. The `JTCHandleT` template is explained in “The JTCHandleT Template” on page 47. It would be wrong to use just a plain C++ pointer `Writer*` here, but for now let’s just assume that `JTCHandleT<Writer>` and `Writer*` are the same.

Lines 4, 5 The two `Writer` threads are started.

Now consider the following scenario: `w1` runs first, but after writing “12” into the buffer the operating system switches to the execution of `w2`, which writes “abc”. After that `w1` continues to write “3”. The result is that the buffer now contains the character sequence “12abc3” instead of “123abc”.

We can easily avoid this by rewriting the `run` method to lock the monitor of the `CharacterBuffer` object before starting to write into the buffer:

```
1 public void run()
2 {
3     synchronized(buffer_)
4     {
5         for(int i = 0 ; i < str.length() ; i++)
6             buffer_.addChar(str_.characterAt(i));
7     }
8 }
```

Lines 3-7 The `for` loop is now placed in a code block synchronized with the `CharacterBuffer`’s monitor lock. This will make sure that the characters are put into the buffer in the proper sequence.

This is called a *synchronized code block*, in contrast to a synchronized method. The example translates to JThreads/C++ as follows:

```
 1 class Writer : public JTCThread
 2 {
 3     CharacterBuffer* buffer_;
 4     const char* str_;
 5
 6 public:
 7
 8     Writer(CharacterBuffer* buffer, const char* str)
 9     {
10         buffer_ = buffer;
11         str_ = str;
12     }
13
14     virtual void run()
15     {
16         {
17             JTCSynchronized synchronized(*buffer_);
18             int len = strlen(str_);
19             for(int i = 0 ; i < len ; i++)
20                 buffer_ -> addChar(str_[i]);
21         }
22     }
23 };
```

Lines 16-21 Instead of using `*this`, `*buffer_` is used for synchronization.

Static Monitors

Synchronized static monitors

In Java it is possible to have static methods which are synchronized. Here is an example:

```
public class StaticCounter
{
    static long counter_;

    public static synchronized void increment()
    {
        ++counter_;
    }

    public static synchronized void decrement()
    {
        --counter_;
    }

    public static synchronized long value()
    {
        return counter_;
    }
};
```

This class allows global access to a protected counter. This class must be synchronized because access to a `long` value in Java is not atomic.

It is not possible to inherit from `JTCMonitor` if static member functions need to be synchronized, since the `JTCSynchronized` class requires `*this` as the argument to its constructor (which is not available within static member functions).

To solve this problem, a static data member of type `JTCMonitor` is used to synchronize static member functions.

This is the Java example converted to C++.

```
 1 class StaticCounter
 2 {
 3     static long counter_;
 4     static JTCMonitor mon_;
 5
 6 public:
 7
 8     static void increment()
 9     {
10         JTCSynchronized sync(mon_);
11         ++counter_;
12     }
13
14     static void decrement()
15     {
16         JTCSynchronized sync(mon_);
17         --counter_;
18     }
19
20     static long value()
21     {
22         JTCSynchronized sync(mon_);
23         return counter_;
24     }
25 };
26
27 long StaticCounter::counter_ = 0;
28 JTCMonitor StaticCounter::mon_;
```

Line 4 A static `JTCMonitor` instance variable is declared. This allows the class to be synchronized.

Lines 10, 16, 22 The methods are synchronized. Instead of using `*this`, the static variable `mon_` is used.

Note that there are certain restrictions on the use of static monitors. It is not correct to use[1] a static monitor before an instance of `JTCInitialize` has been created. Any use before initialization of JThreads/C++ will result in undefined behavior. Additionally, the monitor class must not be used after the final instance of the `JTCInitialize` object was destroyed.

1. Construction and destruction of static monitors (which is out of the control of the application programmer) is not using of monitors in this context.

Note that the only JThreads/C++ classes that can be used as a static member are the `JTCMutex`, `JTCRecursiveMutex` and `JTCMonitor` classes. All other classes must not be used as static members.

The Wait, Notify and NotifyAll Methods

Inter-thread communication

Like in Java, JThreads/C++ offers the `wait`, `notify` and `notifyAll` methods for inter-thread communication. As an example, let's return to our previous example involving the `CharacterBuffer` class. This time, we want the `writeBuffer` operation to behave in a slightly different way: `writeBuffer` should only print the buffer's contents if there are at least 80 characters in the buffer.

Using Wait/Notify with Java

Let's start with rewriting the `writeBuffer` method in Java, using `wait`:

```
 1 synchronized void writeBuffer()
 2 {
 3     while(len_ < 80)
 4     {
 5         try
 6         {
 7             wait()
 8         }
 9         catch(InterruptedException ex)
10         {
11         }
12     }
13     System.out.write(data_, 0, len_);
14     System.out.flush();
15     len_ = 0;
15 }
```

Line 1 The `writeBuffer` method must be declared `synchronized`. This makes sure that the monitor lock is acquired on entry to the method.

Line 3 The `while` loop is executed until there are at least 80 characters available.

Line 7 `wait` is called. This releases the monitor lock (which was acquired on entry to the `writeBuffer` method) and waits for another thread to call either `notify` or `notifyAll` on the monitor.

Lines 5, 9 It is possible that `wait` throws an `InterruptedException`. Therefore this exception must be caught.

Now let's change the `addChar` method so that it calls `notify` whenever there are at least 80 characters in the buffer:

```
 1 synchronized public void addChar(char c)
 2 {
 3     if(data_ == null || len_ >= data_.length)
 4     {
 5         byte[] newData = new byte[len_+128];
 6         if (data_ != null)
 7             System.arraycopy(data_, 0, newData, 0, len_);
 8         data_ = newData;
 9     }
10     data_[len_++] = c;
11     if(len_ >= 80)
12         notify();
13 }
```

Line 1 Again, `addChar` is declared `synchronized` so that the monitor lock is acquired. This is a requirement for using `wait`, `notify` or `notifyAll`.

LInes 11-12 If, after the addition of a new character, the number of characters in the buffer is equal to or larger than 80, `notify` is called. This wakes exactly one thread which is waiting using `wait`. *Waking* in this context means that the `wait` call of the waiting thread returns and implicitly locks the monitor again, making sure that only one thread at a time can run the `synchronized` method.

The difference between `notify` and `notifyAll` is that `notify` only wakes one thread, while `notifyAll` wakes all waiting threads. If more than one thread is waiting, and `notify` is used, a random thread is woken. If more than one thread is waiting and `notifyAll` is used, then all threads are woken, but the order in which the waiting threads return from their call to `wait` is random. Remember that only one thread at a time can return from `wait`, since returning from `wait` requires the monitor's lock to be acquired. This is not possible if another thread has previously returned from `wait` and has not yet released the monitors lock by exiting the `synchronized` method.

For this example, `notify` is used as we know that there is going to be only one waiting thread. However, it doesn't matter whether `notify` or `notifyAll` is used because if more than one thread is waiting, the number of characters is reset to zero once a thread has returned from `wait`. When the other threads subsequently return from `wait`, they will wait again because of the while loop.

Using Wait/Notify with C++

Let’s see how this example translates to JThreads/C++:

```
 1 void writeBuffer()
 2 {
 3     JTCSynchronized synchronized(*this);
 4     while(len_ < 80)
 5     {
 6         try
 7         {
 8             wait();
 9         }
10         catch(const JTCInterruptedException&)
11         {
12         }
13     }
14     cout.write(data_, len_) << flush;
15     len_ = 0;
16 }
```

Line 3 As in the Java example the `writeBuffer` method must be synchronized. Calling `wait`, `notify` or `notifyAll` without having the monitor locked results in the exception `JTCIllegalMonitorStateException` being thrown.

Line 4 Like in the Java example, the `while` loop is executed until there are at least 80 characters available.

Line 8 `wait` is called just in the same way as in the Java example. This releases the monitor lock (which was acquired with the synchronization through the `JTCSynchronize` class) and waits for notification.

Lines 5-9 The equivalent to `java.lang.InterruptedException` is the JThreads/C++ exception `JTCInterruptedException`.

```
 1 void addChar(char c)
 2 {
 3     JTCSynchronized synchronized(*this);
 4     if (len_ >= max_)
 5     {
 6         char* newData = new char[len_+128];
 7         memcpy(newData, data_, len_);
 8         delete[] data_;
 9         data_ = newData;
10         max_ += 128;
11     }
12     data_[len_++] = c;
13     if(len_ >= 80)
14         notify();
15 }
```

Line 3 `addChar` is made synchronized.

Lines 12-13 `notify` is called if 80 characters are available, waking a waiting thread.

As you can see, the semantics of `wait`, `notify` and `notifyAll` in JThreads/C++ are exactly the same as in Java.

The Stop and Suspend Methods

Terminating and suspending execution of a thread

We have already introduced the `start` method of the `JTCThread` class. The opposite of `start` is `stop`, which terminates the execution of a thread. Besides `stop`, there is also a `suspend` method which suspends the execution of a thread until `resume` is called.

Control points

`stop` and `suspend` do not terminate or suspend a thread immediately, because this is not supported by every underlying low-level thread API (for example, POSIX threads). Instead, the JThreads/C++ library uses the concept of *control points* to implement the `suspend` and `stop` methods. This is similar to the cancellation points concept used in the POSIX threads library. If `suspend` or `stop` is called from outside the thread that is to be suspended or stopped, the thread is marked as *control-pending*. When this thread calls a method which is a *control point* the thread is stopped or suspended, respectively.

Once a thread has been suspended, execution for that thread is halted until it is resumed. If a thread has been stopped, the exception `JTCThreadDeath` is raised. If this exception is caught by user code, it must be re-thrown to ensure the proper termination of the thread.

The control points in the JThreads/C++ library are:

- JTCThread::suspend()
- JTCThread::join()
- JTCThread::sleep()
- JTCThread::yield()
- JTCSynchronized::JTCSyncronized()
- JTCSynchronized::~JTCSyncronized()
- JTCMonitor::wait()

Implementing a thread termination method

The Java versions of `stop`, `resume` and `suspend` are deprecated [4]. The reason is that under very rare circumstances, these methods can lead to a deadlock of the Java Virtual Machine.

The JThreads/C++ implementation of `stop`, `resume` and `suspend` is completely portable and does not suffer from the same shortcomings as the Java counterpart. However, in order to keep your source code compatible with Java, we recommend that you provide your own termination method for your thread classes. As an example, let's visit our `CharacterBuffer` class once more. We now want to have a separate thread, which waits for 80 characters to become available. It then prints these 80 characters on standard output, resets the buffer's contents and starts over again. The thread should only stop if a `terminate` method is called on the `CharacterBuffer` class. We can write this class as follows:

```
class CharacterBuffer : public JTCMonitor, public JTCThread
{
    char* data_;
    int max_;
    int len_;
    bool done_;

public:

    CharacterBuffer()
        : data_(0), len_(0), max_(0), done_(false)
    {
    }

    ~CharacterBuffer()
    {
        delete[] data_;
    }

    void addChar(char c)
    {
        JTCSynchronized synchronized(*this);
        if (len_ >= max_)
        {
            char* newData = new char[len_+128];
            memcpy(newData, data_, len_);
            delete[] data_;
            data_ = newData;
            max_ += 128;
        }
        data_[len_++] = c;
        if(len_ >= 80)
            notify();
    }

    virtual void run()
    {
        JTCSynchronized synchronized(*this);
        while(true)
        {
            while(!done_ && len_ < 80)
            {
                try
                {
                    wait();
                }
                catch(const JTCInterruptedException&)
                {
                }
            }
            if(done_)
                break;
            cout.write(data_    , len_) << flush;
            len_ = 0;
        }
    }

    void terminate()
    {
        JTCSynchronized synchronized(*this);
        done_ = true;
        notify();
    }
};
```

Line 1 The `CharacterBuffer` class is now also derived from `JTCThread` in order to provide the separate thread for printing the buffer's contents.

Lines 6, 11 We added a `done_` flag, initially set to `false` in the constructor.

Lines 20-34 Nothing has changed in the `addChar` method. The implementation is the same as in section [TBD].

Lines 36-56 The `writeBuffer` method is obsolete. We now have a `run` method instead, which prints the buffer's contents in an endless loop.

Lines 41-50 This is similar to the implementation shown in section [TBD]. However, the `while` loop now not only checks whether 80 characters are available, but also whether the `done_` flag is set to `true`.

Lines 50-51 If the inner `while` loop was terminated because `done_` was set to `true`, `break` is called. This causes the thread to exit the outer `while` loop, to return from the `run` method and to terminate.

Lines 53-54 If the inner `while` loop was terminated for any other reason, there are 80 characters available now, which are printed on standard output.

Lines 58-63 The `terminate` method serves as a replacement for `stop`. It first acquires the monitor's lock with an instance of `JTCSynchronize`, then sets the `done_` flag to `true` and notifies the waiting thread.

The Join and IsAlive Methods

Waiting for threads to terminate

In some applications it is necessary to explicitly wait for threads to terminate. For instance, if a set of threads is performing a complex parallel calculation, the application may have to wait for the calculation to be completed before continuing.

As an example, let's assume that we want the `main` function of our Hello World program from Chapter 2 to wait for the `HelloWorld` thread to terminate. One way this can be done is as follows:

```
 1 int
 2 main(int argc, char** argv)
 3 {
 4     JTCInitialize initialize;
 5     JTCThreadHandle t = new HelloWorld;
 6     t -> start();
 7     while(t -> isAlive())
 8         ;
 9     return 0;
10 }
```

Line 5 It is absolutely necessary to use `JTCThreadHandle` instead of `JTCThread*` here. See "Introducing Handles" on page 45 for more information. For now, let's just think of a `JTCThreadHandle` as if it would be a `typedef` for `JTCThread*`.

Lines 7, 8 The `isAlive` method is used to wait for the thread to terminate. `isAlive` returns `true` if the thread is alive (that is, if it was started and not yet terminated), or `false` otherwise.

The code above has the obvious problem of busy-looping, which should be avoided at all costs. Fortunately there is alternative approach: `join` can be used for this purpose. This method waits for the thread to terminate and then returns.

```
 1 int
 2 main(int argc, char** argv)
 3 {
 4     JTCInitialize initialize;
 5     JTCThreadHandle t = new HelloWorld;
 6     t -> start();
 7     t -> join();
 8     return 0;
 9 }
```

Line 7 The `join` method is used to wait for the thread to die.

However, this example has a bug. The `join` method can throw the exception `JTCInterruptedException`. Therefore this example should be re-written as follows:

```
 1 int
 2 main(int argc, char** argv)
 3 {
 4     JTCInitialize initialize;
 5     JTCThreadHandle t = new HelloWorld;
 6     t -> start();
 7     do
 8     {
 9         try
10         {
11             t -> join();
12         }
13         catch(const JTCInterruptedException&)
14         {
15         }
16     }
17     while(t -> isAlive());
18     return 0;
19 }
```

Lines 9-15 `join` is called on the thread, which should wait until the thread has terminated. However, if `JTCInterruptedException` is thrown we ignore it.

Line 17 This makes sure that the loop is only terminated if no `JTCInterruptedException` was thrown, that is, if the thread is not alive anymore.

CHAPTER 4

Memory Management

This chapter discusses the memory management features JThreads/C++ such as reference counting and handle classes.

In this chapter

This chapter contains the following sections:

Reference Counting

Avoiding memory leaks

You may have thought that the Hello World examples from Chapter 2 all have memory leaks, since the thread objects are created with `new` but never deleted with `delete`. However, you would be wrong. Why? The magic comes in the form of *reference counting*.

Every `JTCThread` object (and also `JTCThreadGroup` and `JTCRunnable` objects) has a reference counter. When a new thread object is created, this counter is set to 1. When the thread terminates (that is, the `run` method returns), the counter is decremented by 1. Whenever the counter's value drops to 0, the thread object is deleted with `delete`.

Since in our Hello World example the reference count is never incremented, the reference count drops to 0 as soon as `run` returns, meaning that the thread object is deleted upon thread termination - so there is no memory leak.

One drawback of using reference counting is that it is not possible to allocate reference counted objects on the stack. It is only possible to allocate them with `new` (on the heap), since they will be deleted with `delete` as soon as the reference count becomes 0.

Introducing Handles

Smart pointers

You might think that reference counting is pretty complicated, because you now have to remember when to increment or to decrement the counter of a reference counted object. However, this is not the case. JThreads/C++ provides handle classes (sometimes also called *smart pointers*) that take care of incrementing and decrementing the reference counter for you.

Let's go back to the example from "The Join and IsAlive Methods" on page 40. There we told you that it is absolutely necessary to use `JTCThreadHandle` instead of `JTCThread*`. Now we will reveal the secret behind it.

Consider how we would have written the example without `JTCThreadHandle`:

```
// Wrong example!
int
main(int argc, char** argv)
{
    JTCInitialize initialize;
    // Don't do this! Use JTCThreadHandle instead of JTCThread*
    JTCThread* t = new HelloWorld;
    t -> start();
    do
    {
        try
        {
            t -> join();
        }
        catch(const JTCInterruptedException&)
        {
        }
    }
    while(t -> isAlive());
    return 0;
}
```

This example is wrong and the program will most certainly crash. When the thread terminates, its reference count is decremented from 1 to 0 and thus the thread object is deleted with `delete`. However, we are still trying to `join` with the thread and check whether it's still alive using `isAlive` even though the thread object has already been deleted.

So what we would have to do is increase the reference count by 1 after the `new` and to decrement it by 1 after the `while` loop. This would make sure that the reference count drops to 0 *after* the `join` and `isAlive` methods were called.

This is exactly what handles are doing for you. Whenever you assign a thread object to a handle, it increases the reference count of the thread object by 1. The same is true if you assign a handle to another handle. Whenever a handle is destroyed, the destructor of the handle decrements the reference count of the thread object it points to by 1.

For the example above, this means that if you replace `JTCThread*` by `JTCThreadHandle`, the reference count of the thread object will be 2 instead of 1 after the `new`, because the handle increased the counter by 1. After the thread has terminated, the counter is still 1, and thus the thread object is not deleted, so that it is safe to use operations like `isAlive` or `join` on the thread object. When the handle is destroyed at the end of the `main` function, the handle's destructor decrements the thread object's counter by 1, so that the thread object is then also deleted.

The JTCHandleT Template

Handle classes

JThreads/C++ provides the following handle classes:

- `JTCThreadHandle` as a replacement for `JTCThread*`.
- `JTCRunnableHandle` as a replacement for `JTCRunnable*`.
- `JTCThreadGroupHandle` as a replacement for `JTCThreadGroup*`.

These classes are all `typedef`s for a more general handle type written as a C++ template:

```
typedef JTCHandleT<JTCThread> JTCThreadHandle;
typedef JTCHandleT<JTCRunnable> JTCRunnableHandle;
typedef JTCHandleT<JTCThreadGroup> JTCThreadGroupHandle;
```

In case you want to access methods from classes derived from `JTCThread` (or from `JTCRunnable`) you must define your own handle type. As an example, let's go back to "Implementing a thread termination method" on page 37, in which we defined a `terminate` method. In case we actually want to call this method, we cannot use `JTCThreadHandle` as shown below:

```
JTCThreadHandle t = new CharacterBuffer;
... // Do something with the CharacterBuffer
t -> terminate(); // This does not work, compiler will complain
```

Just as you cannot use `JTCThread*` to access methods from classes derived from `JTCThread`, you cannot use `JTCThreadHandle` for this either. You must use the handle class for `CharacterBuffer*`. This can be done by using the `JTCHandleT` template:

```
typedef JTCHandleT<CharacterBuffer> CharacterBufferHandle;
CharacterBufferHandle t = new CharacterBuffer;
... // Do something with the CharacterBuffer
t -> terminate(); // This works
```

Rules of Thumb

Rules

Keep the following rules in mind when using JThreads/C++:

- Always use handle types instead of plain C++ pointers. The only exception can be made if it is absolutely certain that after `start` is called on the thread object the C++ pointer is not used anymore.
- Never allocate thread objects, runnable objects or thread group objects on the stack. Always use `new`.
- Never attempt to delete thread objects, runnable objects or thread group objects with `delete`. They will be deleted automatically.
- Define your own handle types by using the `JTCHandleT` template whenever you must access methods of classes derived from `JTCThread`, `JTCThreadGroup` or `JTCRunnable`.

As long as you follow these basic rules, memory management in JThreads/C++ is virtually automatic.

The nice thing about reference counting and handle classes is that it makes JThreads/C++ even more Java-like. Reference counting emulates the Java garbage collector, and handles emulate Java references.

APPENDIX A

Class Reference

This chapter provides a reference to the classes in the JThreads/C++ library.

In this appendix

This appendix contains the following sections:

JTCInitialize

Overview

An instance of this class must be instantiated before JThreads/C++ is used. If no instance of this class is created, the JThreads/C++ library will not work properly.

`JTCInitialize` can be instantiated multiple times. However, only the first instantiation has any effect. When the last `JTCInitialize` instance is destroyed, the destructor will wait for all running threads to terminate.

`JTCInitialize` interprets arguments starting with `-JTC`. All of these arguments, passed through the `argc` and `argv` parameters, are automatically removed from the argument list.

JTCOptions

The following JThreads/C++ options can be used:

-JTCversion

Shows the JThreads/C++ version number.

-JTCss stack-size

This option sets the thread stack size to `stack-size` kilobytes.

Constructors

JTCInitialize

`JTCInitialize()`

Initializes the JThreads/C++ library.

Throws:

`JTCSystemCallException` - Indicates a failed system call.

JTCInitialize

`JTCInitialize(int& argc, char** argv)`

Initializes the JThreads/C++ library and interprets arguments starting with `-JTC`.

Throws:

`JTCSystemCallException` - Indicates a failed system call.

`JTCInitializeError` - Indicates a that an invalid option or option argument was specified.

Member functions

waitTermination

```
void waitTermination()
```

Waits for all threads to terminate.

initialized

```
static bool initialized()
```

Determines if the JThreads/C++ library has been initialized.

Returns:

`true` if JThreads/C++ has been initialized and `false` otherwise.

JTCAdoptCurrentThread

Overview

When integrating with third-party libraries, it is often necessary to call JThreads/C++ methods from a thread that was not created using JThreads/C++. In this situation, the thread must create an instance of `JTCAdoptCurrentThread` prior to using any other JThreads/C++ classes. Failure to instantiate `JTCAdoptCurrentThread` will result in undefined behavior.

Constructors

JTCAdoptCurrentThread

`JTCAdoptCurrentThread()`

Informs the JThreads/C++ library about the existence of this thread.

Throws:

`JTCSystemCallException` - Indicates a failed system call.

JTCThread

Overview

This class is used to create a new thread of execution. The thread functionality can be added by either deriving a class from `JTCThread` and overriding the `run` method, or by passing an object of a class derived from `JTCRunnable` to the `JTCThread` constructor.

Constructors

JTCThread

```
JTCThread(JTCRunnableHandle target, const char* name = 0)
```

Create a new thread object with a target object and a name.

Parameters:

`target` - The object whose `run` method is invoked when `start` is called. If no object is specified, the `run` method of the thread object must be overridden in a derived class.

`name` - The name of the thread. If no name is specified, a default name is used. This default name is the string "`thread-`" concatenated with the thread id. A thread id is a system-specific identifier generated by the operating system when a new thread is created. Application developers are encouraged to use the `JTCThreadId` class to refer to thread ids.

Throws:

`JTCSystemCallException` - Indicates a failed system call.

JTCThread

```
JTCThread(const char* name)
```

Create a new thread object with a name.

Parameters:

`name` - The name of the thread.

Throws:

`JTCSystemCallException` - Indicates a failed system call.

JTCThread

```
JTCThread(JTCThreadGroupHandle& group, JTCRunnableHandle
    target,  const char* name = 0)
```

Create a new thread object belonging to a group, with a target object and a name.

Parameters:

`group` - The thread group.

`target` - The object whose `run` method is invoked when `start` is called. If no target object is specified, the `run` method of the thread object must be overridden in a derived class.

`name` - The name of the thread. If no name is specified, a default name is used. This default name is the string "`thread-`" concatenated with the thread id. A thread id is a system-specific identifier generated by the operating system when a new thread is created. Application developers are encouraged to use the `JTCThreadId` class to refer to thread ids.

Throws:

`JTCSystemCallException` - Indicates a failed system call.

JTCThread

```
JTCThread(JTCThreadGroupHandle& group, const char* name = 0)
```

Create a new thread object belonging to a group, with a name.

Parameters:

`group` - The thread group.

`name` - The name of the thread. If no name is specified, a default name is used. This default name is the string "`thread-`" concatenated with the thread id. A thread id is a system-specific identifier generated by the operating system when a new thread is created. Application developers are encouraged to use the `JTCThreadId` class to refer to thread ids.

Throws:

`JTCSystemCallException` - Indicates a failed system call.

Member functions

getThreadGroup

`JTCThreadGroupHandle getThreadGroup()`

Returns a thread group handle for the thread group object to which this thread object belongs.

Returns:

A handle for the thread group object.

setName

`void setName(const char* name)`

Sets the name of the thread object.

Parameters:

`name` - The new name for the thread object. If a `null` pointer is used, a default name is used. This default name is the string "`thread-`" concatenated with the thread id. A thread id is a system-specific identifier generated by the operating system when a new thread is created. Application developers are encouraged to use the `JTCThreadId` class to refer to thread ids.

getName

`const char* getName() const`

Returns the name of the thread object.

Returns:

The thread object name.

start

`void start()`

Starts execution of the thread. If the thread was created with a target object (that is, with an object of a class derived from `JTCRunnable)`, the `run` method of the target object is invoked. If there is no target object, the `run` method of the thread object itself is invoked. In this case, a class derived from `JTCThread` with an overridden `run` method should be used.

Throws:

`JTCSystemCallException` - Indicates a failed system call.

`JTCIllegalStateException` - Thrown if the thread has already been started.

run

`virtual void run()`

This method is called when `start` is invoked. If the thread object has been constructed with an associated target `JTCRunnable` object, the target's `run` method is invoked. Otherwise, the `run` method should be overridden in a class derived from `JTCThread`. If `run` terminates due to an uncaught exception, then the thread's thread group method `uncaughtException` is called.

isAlive

`bool isAlive() const`

This method determines whether the thread is alive.

Returns:

`true` if the thread is alive, `false` otherwise.

join

`void join()`

Waits for the thread to terminate.

Throws:

`JTCSystemCallException` - Indicates a failed system call.

join

`void join(long millis)`

Waits for the thread to terminate for at most `millis` milliseconds.

Throws:

`JTCSystemCallException` - Indicates a failed system call.

`JTCIllegalArgumentException` - Thrown if the value of `millis` is negative.

join

`void join(long millis, int nanos)`

Waits for the thread to terminate for at most `millis` milliseconds and `nanos` nanoseconds.

Throws:

`JTCSystemCallException` - Indicates a failed system call.

`JTCIllegalArgumentException` - Thrown if the value of `millis` is negative, or if the value of `nanos` is not in the range 0 - 999999.

setPriority

```
void setPriority(int newPri)
```

Sets the thread priority to a new value.

Parameters:

`newPri` - The new thread priority.

Throws:

`JTCSystemCallException` - Indicates a failed system call.

getPriority

```
int getPriority() const
```

Returns the priority of the thread.

Returns:

The thread priority.

Throws:

`JTCSystemCallException` - Indicates a failed system call.

enumerate

```
static int enumerate(JTCThreadHandle* list, int len)
```

Copies each active thread from this thread's thread group and subgroups into the array `list`. If more than `len` items are present, the list is truncated.

Parameters:

`list` - The array into which all threads from this thread's group and subgroups are copied.

`len` - The number of `JTCThreadHandle*` elements in `list`.

Returns:

The number of threads returned in `list`.

currentThread

```
static JTCThread* currentThread()
```

Returns a pointer to the currently executing thread object.

Returns:

The currently executing thread object.

sleep

```
static void sleep(long millis, int nanos = 0)
```

Suspends execution of this thread for `millis` milliseconds, and `nanos` nanoseconds.

Parameters:

`millis` - The number of milliseconds to sleep.

`nanos` - The number of nanoseconds to sleep.

Throws:

`JTCSystemCallException` - Indicates a failed system call.

`JTCIllegalArgumentException` - Thrown if the value of `millis` is negative, or if the value of `nanos` is not in the range 0 - 999999.

`JTCInterruptedException` - Thrown if the `sleep` call is interrupted.

yield

```
static void yield()
```

Gives up the thread's current timeslice. This can be called if you want to manually give other threads an opportunity to execute.

activeCount

```
static int activeCount()
```

Returns the number of active threads in this thread's thread group and subgroups.

Returns:

The number of active threads in this thread's group and subgroups.

getId

```
JTCThreadId getId() const
```

Returns the id of the thread.

Returns:

The thread id of the thread.

setAttrHook

```
typedef void (*JTCAttrHook)(pthread_attr_t*)
```

```
static void setAttrHook(JTCAttrHook hook, JTCAttrHook*
    oldHook = 0)
```

Sets/gets a *hook* that will be used to initialize custom POSIX thread attributes. Note: this method is only available for systems using POSIX threads.

Parameters:

`hook` - The function that will be called to retrieve the custom POSIX thread attributes before the creation of each thread.

`oldHook` - Optional parameter in which the previously set hook is returned. Applications should call this function within the new hook. In essence, hooks may be chained.

setRunHook

```
typedef void (*JTCRunHook)(JTCThread*)

static void setRunHook(JTCRunHook hook, JTCRunHook* oldHook =
    0)
```

Sets/gets a run *hook* which may be used to setup any application specific information during thread creation. The hook function must call `thread -> run()` to actually run the thread.

Parameters:

hook - The function that will be called on creation of the thread.

`oldHook` - Optional parameter in which the previously set hook is returned. Applications should call this function from within the new hook. In essence, hooks may be chained.

setStartHook

```
typedef void (*JTCStartHook)()

static void setStartHook(JTCStartHook hook, JTCStartHook*
    oldHook = 0)
```

Sets/gets a start *hook* which may be used to setup any thread specific information.

Parameters:

hook - The function that will be called on creation of the thread.

`oldHook` - Optional parameter in which the previously set hook is returned. Applications should call this function from within the new hook. In essence, hooks may be chained.

Data members

JTC_MIN_PRIORITY

`const int JTC_MIN_PRIORITY`

A constant for the minimum priority a thread can have.

JTC_NORM_PRIORITY

`const int JTC_NORM_PRIORITY`

A constant for the default priority of a thread.

JTC_MAX_PRIORITY

`const int JTC_MAX_PRIORITY`

A constant for the maximum priority a thread can have.

Related functions

operator<<

`ostream& operator<<(ostream& os, const JTCThread& thr)`

Print the thread id to the output stream `os`. The output format of the thread-id field is platform-specific.

Parameters:

`os` - Output stream in which to insert the thread id.

`thr` - Reference to the thread.

Returns:

The output stream `os`.

JTCRunnable

Overview

This class is provided as an alternative method of providing functionality in a thread. In order to use this class, you must write a subclass and provide a definition for the `run` method. An instance of this class should then be provided as an argument to the `JTCThread` constructor. When the thread is started, the `run` method of that instance will be invoked.

Member functions

run

```
virtual void run()
```

Called when the `start` method is called on the associated thread object.

JTCThreadGroup

Overview

This class represents a collection of threads, and other thread groups. The thread groups form a tree, rooted at the system thread group. New threads by default belong to the thread group of their parent thread. A thread group can optionally be a daemon thread group, which automatically destroys itself after all threads have terminated and all sub-groups are destroyed. A newly created thread group inherits its parent's daemon status. The root thread group is a non-daemon thread group.

Constructors

JTCThreadGroup

```
JTCThreadGroup(const char* name)
```

Creates a new thread group with the provided name. The new thread group's parent is that of the current thread.

Parameters:

`name` - The name of the thread group.

Throws:

`JTCIllegalThreadStateException` - Thrown if the parent thread group has been destroyed..

JTCThreadGroup

```
JTCThreadGroup(JTCThreadGroup* group, const char* name)
```

Creates a new thread group with the provided `name` and parent thread `group`.

Parameters:

`group` - The parent of the thread group.

`name` - The name of the thread group.

Throws:

`JTCIllegalThreadStateException` - Thrown if the parent thread group has been destroyed..

Member functions

getName()

```
const char* getName() const
```

Returns the name of the thread group.

Returns:

The name of the thread group.

getParent

```
JTCThreadGroupHandle getParent() const
```

Returns the parent of the thread group. If the thread group is the root thread group, the handle contains a null pointer.

Returns:

The parent of the thread group.

isDaemon

```
bool isDaemon() const
```

Returns the daemon flag for this thread group. If the daemon flag is true, the thread group is destroyed once all threads are terminated and sub-groups are empty.

Returns:

The value of the daemon flag.

setDaemon

```
void setDaemon(bool daemon)
```

Sets the daemon flag for this thread group. If the daemon flag is `true`, the thread group is destroyed once all threads are terminated and sub-groups are empty.

Parameters:

`daemon` - The new value for the daemon flag.

uncaughtException

```
virtual void uncaughtException(JTCThreadHandle t, const
   JTCException& e)
```

This method is called if a `JTCThread::run()` exits because of an uncaught `JTCException`. By default, if the thread group has a parent this method invokes the parent's `uncaughtException` method, otherwise it displays the exception to `stderr`.

Parameters:

`t` - The thread that threw the `JTCException`.

`e` - The uncaught exception.

uncaughtException

```
virtual void uncaughtException(JTCThreadHandle t)
```

This method is called if a `JTCThread::run()` exits because of an uncaught exception. By default, if the thread group has a parent this method invokes the parent's `uncaughtException` method, otherwise it displays the text "uncaught exception" to `stderr`.

Parameters:

`t` - The thread that threw the `JTCException`.

getMaxPriority

```
int getMaxPriority() const
```

Returns the maximum priority permitted for threads in this thread group.

Returns:

The maximum priority of this thread group.

isDestroyed

```
bool isDestroyed() const
```

Determines if the thread group has been destroyed. A thread group is destroyed once all threads have terminated in the thread group and all subgroups.

Returns:

`true` if the thread group has been destroyed, `false` otherwise.

destroy

```
void destroy()
```

Destroys this thread group and all of its subgroups. The thread group must not contain any active threads.

Throws:

`JTCIllegalThreadStateException` - If the thread group has active threads, or has already been destroyed.

setMaxPriority

`void setMaxPriority(int pri)`

Sets the maximum priority that threads in the thread group and its subgroups may have. Threads in the thread group that have higher priority are not affected. That is, their priorities are not lowered.

parentOf

`bool parentOf(JTCThreadGroupHandle g)`

Returns `true` if the thread group is the parent of thread group `g`.

Returns:

`true` if this thread is a parent of `g`, `false` otherwise.

activeCount

`int activeCount() const`

Returns the number of active threads in this thread group, and all of its subgroups.

Returns:

The number of active threads.

activeGroupCount

`int activeGroupCount() const`

Returns the number of active thread groups in this thread group.

Returns:

The number of active thread groups.

enumerate

```
int enumerate(JTCThreadHandle* list, int len, bool recurse =
   true) const
```

Copies pointers to each active thread in this thread group to the array `list`. `activeCount` can be used to get an estimate of how big the array should be. If more than `len` active threads are present, the remaining threads are silently ignored. The reason that the developer cannot determine precisely the number of threads is that threads can be added and removed from the thread group at the same time as they are enumerated.

Parameters:

`list` - The array into which the set of active threads should be copied.

`len` - The length of the array `list`.

`recurse` - If set to `true`, active threads from subgroups are also enumerated.

Returns:

The number of active threads copied to the array.

enumerate

```
int enumerate(JTCThreadGroupHandle* list, int len, bool
   recurse = true) const
```

Copies handles for every active subgroup of this thread group into the array `list`. `activeGroupCount` can be used to determine how big the array must be. If more than `len` active subgroups are present, the remaining subgroups are silently ignored.

Parameters:

`list` - The array in which to copy the thread group handles.

`len` - The length of the array.

`recurse` - If set to `true`, child subgroups are also enumerated.

Returns:

The number of handles copied to `list`.

list

```
void list()
```

Outputs to the stream `cout` the set of threads and subgroups.

list

```
void list(ostream& os, int indent)
```

Outputs to the stream `os` the set of threads and subgroups. Use `indent` spaces for indentation.

Related Functions

operator<<

```
ostream& operator<<(ostream& os, const JTCThreadGroup& g)
```

Prints a string representation of the thread group to the output stream `os`. This calls `g.list(os, 4)`.

Parameters:

`os` - Output stream in which to insert the thread id.

`g` - Reference to the thread group.

Returns:

The output stream `os`.

JTCHandleT

Overview

The JThreads/C++ library cannot know when to delete instances of `JTCThread`, `JTCRunnable`, and `JTCThreadGroup`. One solution to this dilemma is to force application developers to delete instances of these classes when they are sure the instances are no longer useful. However, this is error prone. Fortunately there is a well-known solution to this problem: reference counting (see [4], p. 782). This isn't necessary in Java since it provides garbage collection. The basic idea is to count the number of references to the object, and delete the object when the reference count drops to zero. To ease the counting of references, a handle class is used that increments the reference count when constructed, and decrements the reference count when destructed.

The `JTCHandleT` is a *smart pointer* to a reference-counted object. A regular pointer to instances of these classes should never be stored.

The classes `JTCThreadGroupHandle`, `JTCThreadHandle`, and `JTCRunnableHandle` are all convenience `typedef`s of this template class.

Constructors

JTCHandleT

```
JTCHandleT(T* tg = 0)
```

Creates a handle that refers to the object `tg`.

Parameters:

`tg` - The object to reference.

JTCHandleT

```
JTCHandleT(const JTCHandleT<T>& rhs)
```

Creates a handle that refers to the object referred to by `rhs`.

Parameters:

`rhs` - The handle from which to retrieve the object.

Member functions

operator=

`JTCHandle<T>& operator=(const JTCHandleT<T>& rhs)`

Creates a handle that refers to the object referenced by `rhs`.

Parameters:

`rhs` - The handle from which to retrieve the object.

Returns:

A reference to the handle.

operator==

`bool operator==(const JTCHandleT<T>& rhs) const`

Returns `true` if `rhs` references the same object as this object.

Parameters:

`rhs` - The handle to compare with.

Returns:

`true` if the objects are equivalent, `false` otherwise.

operator!=

`bool operator!=(const JTCHandleT<T>& rhs) const`

Returns `true` if `rhs` references a different object as this.

Parameters:

`rhs` - The handle to compare with.

Returns:

`true` if the objects are not equivalent, `false` otherwise.

operator!

`bool operator!() const`

Determines whether the object referenced by the handle is not valid (that it is nil).

Returns:

`true` if the object is not valid, `false` otherwise.

operator bool

`operator bool () const`

Determines whether the object referenced by the handle is valid (that it is not nil).

Returns:

`true` if the object is valid, `false` otherwise.

operator->

```
T* operator->() const
```

Invokes a method on the referenced object.

Returns:

A pointer to the referenced object.

get

```
T* get() const
```

Gets a pointer to the referenced object.

Returns:

A pointer to the referenced object.

operator*

```
T& operator*()
```

Retrieve a C++ reference to the referenced object.

Returns:

A C++ reference to the object.

JTCMonitor

Overview

This class provides the functionality of Java monitors. In order to implement synchronized methods, the monitor's lock must be acquired, for example by creating an instance of the `JTCSynchronized` class at the top of the synchronized method, with the monitor as the argument to the constructor.

The monitor's `wait` method can be used to release the monitor's lock and to wait for notifications. The `notify` and `notifyAll` methods can be used to wake one or all waiting monitors, respectively.

Methods

wait

`void wait()`

Waits for notification by another thread. The calling thread must own the monitor's lock. The monitor's lock is released and the thread waits for notification by another thread via a call to either `notify` or `notifyAll`. The thread then waits until it can regain ownership of the monitor's lock and then resumes execution.

Throws:

`JTCIllegalMonitorStateException` - If the monitor is not locked by the calling thread.

`JTCSystemCallException` - Indicates a failed system call.

wait

`wait(long timeout)`

Waits for notification by another thread. The calling thread must own the monitor's lock. The monitor's lock is released and the thread waits for notification by another thread via a call to either `notify` or `notifyAll`, or until `timeout` milliseconds have passed. The thread then waits until it can regain ownership of the monitor's lock and then resumes execution.

Parameters:

`timeout` - The maximum number of milliseconds to wait for notification.

Throws:

`JTCIllegalMonitorStateException` - If the monitor is not locked by the calling thread.

`JTCSystemCallException` - Indicates a failed system call.

notify

```
void notify()
```

Wakes a single thread waiting on the monitor. The calling thread must own the monitor’s lock.

notifyAll()

```
void notifyAll()
```

Wakes all threads waiting on the monitor. The calling thread must own the monitor’s lock.

JTCMonitorT

Overview

This is a template class that allows creation of synchronized classes without altering the implementation.

Member functions

wait

`void wait()`

Waits for notification by another thread. The calling thread must own the monitor's lock. The monitor's lock is released and the thread waits for notification by another thread via a call to either `notify` or `notifyAll`. The thread then waits until it can regain ownership of the monitor's lock and then resumes execution.

Throws:

`JTCIllegalMonitorStateException` - If the monitor is not locked by the calling thread.

`JTCSystemCallException` - Indicates a failed system call.

wait

`wait(long timeout)`

Waits for notification by another thread. The calling thread must own the monitor's lock. The monitor's lock is released and the thread waits for notification by another thread via a call to either `notify` or `notifyAll`, or until `timeout` milliseconds have passed. The thread then waits until it can regain ownership of the monitor's lock and then resumes execution.

Parameters:

`timeout` - The maximum number of milliseconds to wait for notification.

Throws:

`JTCIllegalMonitorStateException` - If the monitor is not locked by the calling thread.

`JTCSystemCallException` - Indicates a failed system call.

notify

```
void notify()
```

Wakes a single thread waiting on the monitor. The calling thread must own the monitor's lock.

notifyAll()

```
void notifyAll()
```

Wakes all threads waiting on the monitor. The calling thread must own the monitor's lock.

JTCRecursiveMutex

Overview

This class can be used to establish a critical section. This class has no direct equivalent in Java, and is provided for performance reasons only. An instance of `JTCRecursiveMutex` can be locked multiple times by the same thread, and therefore may not be as efficient as the `JTCMutex` class. The developer is responsible for ensuring that each mutex lock has a corresponding unlock.

Member functions

lock

```
bool lock() const
```

Lock the mutex. If the mutex is already locked, the calling thread blocks until the mutex is unlocked. If the current owner of the mutex attempts to re-lock the mutex, a deadlock will not result.

Returns:

`true`, if the mutex is locked for the first time, `false`, otherwise.

unlock

```
bool unlock() const
```

This method is called by the owner of the mutex to release it. The mutex must be locked and the calling thread must be the one that last locked the mutex. If these conditions are not met, undefined behavior will result.

Returns:

`true`, if the mutex is available for locking by some other thread, `false` otherwise.

trylock

```
bool trylock() const
```

This method is identical to `lock` except that if the mutex is already locked, then `false` is returned.

Returns:

`true`, if the mutex was locked, `false` otherwise.

get_owner

```
JTCThreadId get_owner() const
```

Return the thread id of the owning thread.

Returns:

The thread id of the owning thread.

JTCMutex

Overview

This class can be used to establish a critical section. This class has no direct equivalent in Java. It is provided for performance reasons only. Unlike `JTCMonitor` or `JTCRecursiveMutex`, this class not does guarantee recursive locking semantics. If the mutex is locked more than once by the same thread, a deadlock may result.[1]

Member functions

lock

`bool lock() const`

Lock the mutex. If the mutex is already locked, the calling thread blocks until the mutex is unlocked. If the current owner of the mutex attempts to re-lock the mutex, a deadlock may result.

Returns:

This method always returns `true`.

unlock

`bool unlock() const`

This method is called by the owner of the mutex to release it. The mutex must be locked and the calling thread must be the one that last locked the mutex. If these conditions are not met, undefined behavior will result.

Returns:

This method always returns `true`.

trylock

`bool trylock() const`

This method is identical to `lock` except that if the mutex is already locked, then `false` is returned.

Returns:

`true`, if the mutex was locked, `false` otherwise.

1. Under Windows, `JTCMutex` allows recursive locking, while a pthreads implementation (for example, under UNIX) does not.

get_owner

```
JTCThreadId get_owner() const
```

Return the thread id of the owning thread.

Returns:

The thread id of the owning thread.

JTCRWMutex

Overview

This class can be used to create read-write locks. This class has no direct equivalent in Java. It is provided for performance reasons only. Like `JTCMutex`, this class not does guarantee recursive locking semantics. If the mutex is locked more than once by the same thread, a deadlock may result.

Member functions

read_lock

```
void read_lock() const
```

Lock the mutex for reading. If the mutex is locked for writing or writers are waiting for a write lock, the calling thread blocks until the mutex is unlocked. If the current owner of the mutex attempts to re-lock the mutex, a deadlock may result.

write_lock

```
void write_lock() const
```

Lock the mutex for writing. If the mutex is locked for reading or writing, the calling thread blocks until the mutex is unlocked. If the current owner of the mutex attempts to re-lock the mutex, a deadlock may result.

unlock

```
void unlock() const
```

This method is called by the owner of the mutex to release it. The mutex must be locked and the calling thread must be the one that last locked the mutex. If these conditions are not met, undefined behavior will result.

JTCSynchronized

Overview

This class is used to acquire and release a monitor's lock. To create a synchronized method, an instance of this class should be created with the monitor as the constructor argument. The constructor acquires the lock and the destructor releases the lock. This class may also be used with the classes `JTCMutex`, `JTCRecursiveMutex` and `JTCRWMutex`.

Constructor

JTCSynchronized

```
JTCSynchronized(const JTCMonitor& mon)
```

Acquires the monitor's lock. The destructor releases the monitor's lock.

Throws:

`JTCSystemCallException` - Indicates a failed system call.

JTCSynchronized

```
JTCSynchronized(const JTCMutex& mon)
```

Acquires the mutex's lock. The destructor releases the mutex's lock.

Throws:

`JTCSystemCallException` - Indicates a failed system call.

JTCSynchronized

```
JTCSynchronized(const JTCRecursiveMutex& mon)
```

Acquires the mutex's lock. The destructor releases the mutex's lock.

Throws:

`JTCSystemCallException` - Indicates a failed system call.

JTCSynchronized

```
enum ReadWriteLockType
        {
            read_lock,
            write_lock
        };

JTCSynchronized(const JTCRWMutex& mon, ReadWriteLockType
    type)
```

Acquires the mutex's lock. The destructor releases the mutex's lock.

Throws:

`JTCSystemCallException` - Indicates a failed system call.

JTCSyncT

Overview

This class is a template version of the `JTCSynchronized` class. The `JTCSyncT` template is more efficient than the `JTCSynchronized` class, however it is more difficult to use. The template's constructor invokes the `lock` method on the parameter class, and the destructor invokes the `unlock` method. This template may also be instantiated with the classes `JTCMonitor`, `JTCMutex` and `JTCRecursiveMutex`.

Constructor

JTCSyncT

```
JTCSyncT(const T& mon)
```

Acquires the monitor's lock. The destructor releases the monitor's lock.

Throws:

`JTCSystemCallException` - Indicates a failed system call.

JTCReadLock

Overview

This class is used to acquire and release a read lock. To create a synchronized method, an instance of this class should be created with the `JTCRWMutex` as the constructor argument. The constructor acquires the lock and the destructor releases the lock.

Constructor

JTCReadLock

```
JTCReadLock(const JTCRWMutex& mon)
```

Acquires the mutex's lock for reading. The destructor releases the mutex's lock.

Throws:

`JTCSystemCallException` - Indicates a failed system call.

JTCWriteLock

Overview

This class is used to acquire and release a write lock. To create a synchronized method, an instance of this class should be created with the `JTCRWMutex` as the constructor argument. The constructor acquires the lock and the destructor releases the lock.

Constructor

JTCWriteLock

```
JTCWriteLock(const JTCRWMutex& mon)
```

Acquires the mutex's lock for writing. The destructor releases the mutex's lock.

Throws:

`JTCSystemCallException` - Indicates a failed system call.

JTCThreadId

This class represents a thread id. The only operations that should be used are equality and inequality. Two thread objects may be considered to be equal if their thread ids are equivalent. A user should not directly construct instances of this class.

Member functions

operator==

```
bool operator==(const JTCThreadId& rhs)
```

Compares for equality.

Parameters:

`rhs` - The thread id with which to compare.

Returns:

`true` if the thread ids are equivalent, `false` otherwise.

operator!=

```
bool operator!=(const JTCThreadId& rhs) const
```

Compares for inequality.

Parameters:

`rhs` - The thread id with which to compare.

Returns:

`true` if the thread ids are not equivalent, `false` otherwise.

JTCThreadKey

Overview

This type represents a thread specific storage key. `JTCThreadKey` should be used as an opaque type.

JTCTSS

Overview

This class is used to manage thread-specific storage, which is an extremely useful method of managing data that is associated with each thread, while avoiding the overhead of a mutex. Using thread-specific storage, each thread associates data with a key. Because each thread has its own data, there is no contention for the data among multiple threads.

Member functions

allocate

```
static JTCThreadKey allocate()
```

Creates a new thread-specific storage key.

Returns:

A new thread-specific storage key.

Throws:

`JTCSystemCallException` - Indicates a failed system call.

allocate

```
static JTCThreadKey allocate(void (*)(void*))
```

Creates a new thread-specific storage key with an associated cleanup function. Upon thread termination, the registered cleanup function is called with an argument that contains the value associated with the thread-specific storage key.

Returns:

A new thread-specific storage key.

Throws:

`JTCSystemCallException` - Indicates a failed system call.

release

```
static void release(JTCThreadKey key)
```

Releases a thread-specific storage key. The developer is responsible for freeing any associated storage before releasing the key. Any associated cleanup function is not called.

Parameters:

`key` – The thread-specific storage key to release.

Throws:

`JTCSystemCallException` - Indicates a failed system call.

get

```
static void* get(JTCThreadKey key)
```

Gets the data associated with a thread-specific storage key.

Parameters:

`key` – The thread-specific storage key.

Returns:

The data associated with the thread-specific storage key.

Throws:

`JTCSystemCallException` - Indicates a failed system call.

set

```
static void set(JTCThreadKey key, void* data)
```

Associates data with a thread-specific storage key.

Parameters:

`key` – The thread-specific storage key.

`data` - The data to associate with the key.

Throws:

`JTCSystemCallException` - Indicates a failed system call.

JTCThreadDeath

Overview

This exception is thrown when a thread is terminated by `JTCThread::stop`. If this exception is caught, it must be re-thrown to ensure correct termination of the thread.

JTCException

Overview

With the exception of `JTCThreadDeath`, `JTCException` is the base class of all JThreads/C++ exception classes.

Constructors

JTCException

```
JTCException(const char* note = "", long error = 0)
```

Constructs a `JTCException` with the message in `note`, and the error type in `error`.

Parameters:

`note` - A description of the error.

`error` - An exception-specific error code.

Member functions

getError

```
long getError() const
```

Returns the exception-specific error code. Currently only `JTCSystemCallException` has a specific error code.

Returns:

The error code.

getType

```
virtual const char* getType() const
```

Returns a string representation of the exception type. This is the name of the exception class. This member is not available in Java.

Returns:

The class name.

getMessage

```
const char* getMessage() const
```

Returns a description of the exception. This is the `note` parameter provided in the constructor.

Returns:

A description of the exception.

Related functions

operator<<

```
ostream& operator<<(ostream& os, const JTCException& e)
```

Inserts a description of the error to the output stream `os`.

Parameters:

`os` - The output stream in which to insert the thread id.

`e` - The reference to the exception.

Returns:

The output stream `os`.

JTCInterruptedException

Overview

This exception is thrown if a system call is interrupted. Currently `JTCMonitor::wait()` and `JTCThread::sleep()` can throw this exception. The semantics differ from Java in this respect. An `InterruptedException` in Java is thrown if a thread is interrupted by `java.lang.Thread.interrupt`. Unfortunately, it is impossible to implement this method in a portable fashion using the POSIX and WIN32 threading models.

JTCIllegalThreadStateException

Overview

This exception is thrown if a member function is called while the object is in an illegal state. Currently `JTCThread::start()`, the `JTCThreadGroup::JTCThreadGroup()` constructors and `JTCThreadGroup::destroy()` can throw this exception.

JTCIllegalMonitorStateException

Overview

This exception is thrown by `JTCMonitor::wait()`, `JTCMonitor::notify()` or `JTCMonitor::notifyAll()` if the monitor's lock has not been acquired by the calling thread.

JTCIllegalArgumentException

Overview

This exception is thrown when an illegal argument is passed to a JThreads/C++ method. The methods `JTCMonitor::wait()` (with a timeout argument), `JTCThread::setPriority()`, and `JTCThread::sleep()` can throw this exception.

JTCSystemCallException

Overview

This exception indicates a failed system call. Most JThreads/C++ methods can generate this exception. The `JTCException::getError()` method returns the error value. Under UNIX this is the value of `errno`, under WIN32 this is the value of `getLastError()`. There is no application method of determining which operation caused the error. However, the exception message contains a description of the operation, and all arguments to assist in debugging.

JTCUnknownThreadException

Overview

This exception is generated from the `JTCThread::currentThread` method when the current thread is not known.

JTCOutOfMemoryError

Overview

This exception is generated from the `JTCThread` constructors on an out of memory condition.

JTCInitializeError

Overview

This exception is generated from the `JTCInitialize(int&, char**)` constructor when an invalid option or option argument is specified.

JThreads Bibliography

[1] Scott Oaks & Henry Wong, *Java Threads*, O'Reilly & Associates, Inc., 1997.

[2] Doug Lea, *Concurrent Programming in Java*, Addison-Wesley Longman, Inc., 1997.

[3] *Why JavaSoft is Deprecating Thread.stop, Thread.suspend and Thread.resume*, Sun Microsystems, Inc.[1]

[4] Bjarne Stroupstrup, *The C++ Programming Language*, Third Edition, Addison-Wesley Longman, Inc., 1997.

1. Available from http://java.sun.com/products/jdk/1.2/docs/guide/misc/threadPrimitiveDeprecation .html.