JThreads/C++

"Java-like Threads for C++"

Version 1.0.14

Capitolo1

Introduzione

1.1 Che cosa è JThreads/C++?

JTHREADS/C++ è la forma abbreviata di “Java-like Threads for C++”. JTHREADS/C++ è una libreria per i programmatori C++ per l'astrazione dei thread ad alto livello secondo il “look & feel” dei thread di Java.

Java supporta la programmazione multi-thread tramite le classi `java.lang.Thread` e `java.lang.ThreadGroup`, l'interfaccia `java.lang.Runnable`, la parola chiave synchronized insieme con i metodi `wait`, `notify` e `notifyAll` in `java.lang.Object`.

Vediamo ora come JTHREADS/C++ traduce tutto ciò verso C++:

- Le classi Java `java.lang.Thread` e `java.lang.ThreadGroup` sono direttamente tradotte nelle classi C++ `JTCThread` e `JTCThreadGroup`. La sola differenza è che le classi JTHREADS/C++ hanno `JTC` come prefisso al posto del package `java.lang`. L'interfaccia Java `java.lang.Runnable` è implementata come la classe C++ astratta `JTCRunnable`, che contiene il puro metodo virtuale `run`.
- Il supporto per la parola chiave `synchronized` è leggermente più difficoltoso, in quanto non è possibile aggiungere parole chiavi nuove in C++. JTHREADS/C++ risolve ciò usando la classe `JTCMonitor` e `JTCSynchronized`. Istanze di `JTCSynchronized` possono essere usate al posto della parola chiave `synchronized`, purchè una istanza di `JTCMonitor` sia stata creata per l'oggetto per essere sincronizzato. `JTCMonitor` fornisce anche i metodi `wait`, `notify` e `notifyAll`.

Ci sono alcune funzioni del modello thread di Java che non vengono implementate in JTHREADS/C++. Queste sono:

- Le API sulla sicurezza. Ciò perchè alcune parti delle API semplicemente non possono essere implementate in C++. In generale questo argomento non è così importante come in Java, in quanto C++ non viene usato per le applicazioni Internet (“applets”) allo stesso modo di Java.
- Il controllo dei thread primitivi `java.lang.Thread.stop`, `java.lang.Thread.suspend`, e `java.lang.Thread.resume` non può essere implementato con la stessa semantica sulla portabilità come il modello dei thread Java. Il thread WIN32 API supporta le primitive per queste operazioni, ma il thread POSIX API no. In generale, questa non è una buona idea per usare le API primitive, in quanto possono condurre facilmente a situazioni di deadlock. Queste primitive sono deprecate nella JDK 1.2 [4], e perciò non vengono supportate nelle versioni aggiornate di Java.

1.2 Licenza?

JTHREADS/C++ è *free* per usi non commerciali.

1.3 Help

Per assistenza su JTHREADS/C++, contattate `support@ooc.com`.
`http://www.ooc.com/jtc/mailing-list.html`.

1.4 Informazioni su questo Manuale

Questo manuale non è un sostituto per una libro sulla programmazione dei thread, ma solo descrive come i thread Java sono tradotti in JTHREADS/C++.
Ci sono ottimi libri sulla programmazione dei thread in Java, per esempio [2] e [3].
Si raccomanda l'uso di questi libri mentre imparate JTHREADS/C++.
Con l'aiuto di questo manuale sarà facile tradurre gli esempi che troverete in programmi JTHREADS/C++.

Capitolo 2

"Hello World"

Cominciamo con il primo programma più famoso: Un programma che visualizza il testo "Hello World". Comunque, il nostro esempio è differente dal tipico programma "Hello World" in quanto esso è multi-thread. Infatti, la nostra versione lancia un nuovo thread che ha il solo scopo di stampare "Hello World" sullo schermo.

2.1 "Hello World" in Java

In Java, questo programma può essere scritto come:

```
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 }
```

1 Viene definita una classe HelloWorld, che estende la classe `java.lang.Thread`.

3-6 E' definito un metodo `run`, che visualizza "Hello World" sull'output standard.

8-12 E' definito un metodo statico `main` che crea un oggetto di tipo `HelloWorld`. Il metodo start è chiamato e un nuovo thread è lanciato. Questo thread allora invoca il metodo `run` dell'oggetto `HelloWorld`.

2.2 *"Hello World" in C++*

Andiamo a convertire il programma Java in un programma JTHREADS/C++:

```
 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 }
```

1 Tutti i programmi JTHREADS/C++ devono includere l'intestazione del file `JTC/JTC.h`, che contiene (fra altre cose utili) tutti le definizioni di classi necessarie al JTHREADS/C++.

3 Proprio come nell'esempio Java, una classe `HelloWorld` è definita. Questa classe è derivata da `JTCThread` invece dell'equivalente Java `java.lang.Thread`

6-9 Un metodo `run` è definito, il quale stampa "Hello World" sullo standard output. `System.out` è sostituito dal consueto cout (oggetto iostream C++).

12-19 E' definito un metodo principale main, non come una classe statica come in Java, ma come una funzione globale standard C++. `main` crea un oggetto di tipo `HelloWorld` e chiama il metodo `start` che lancia in esecuzione un nuovo thread.

L'altra sola differenza è che il thread della libreria JTHREADS/C++ deve essere inizializzato nel `main`. Ciò è fatto creando un'istanza della classe `JTCInitialize`.

A prima vista questa applicazione sembra presentare un problema. Può l'applicazione terminare a causa di un return dal `main` prima che il thread venga avviato?
La risposta è No, perchè il distruttore di `JTCInitialize` non parte finchè tutti i thread non sono terminati.
Il distruttore `JTCInitialize` permette alle applicazioni JTHREADS/C++ di avere lo stesso comportamento come le applicazioni multi-threaded Java.

2.3 *"Hello World" con Runnable*

Java fornisce l'interfaccia `Runnable`, così che una applicazione potrebbe usare i thread senza usare l'ereditarietà. L'equivalente in JTHREADS/C++ dell'interfaccia `Runnable` è la classe `JTCRunnable`.

L'esempio "Hello World" che usa l'interfaccia `Runnable` in Java diventa come:

```
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 }
```

1 E' dichiarato che la classe HelloWorld implementa l'interfaccia `Runnable`.

10 Un nuovo thread è creato con un oggetto `Runnable` come parametro, che in questo caso è una istanza della classe `HelloWorld`.

11 Il thread è lanciato. Poichè l'oggetto `Thread` è stato creato con un oggetto `Runnable` come parametro, il metodo `run` di questo `Runnable` è invocato.

La versione Java può essere tradotta direttamente in una applicazione JTHREADS/C++:

```
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 }
```

3 Come nell'esempio Java, la classe `HelloWorld` eredita dalla classe JTHREADS/C++ `JTCRunnable`.

16 Crea un nuovo thread, usando una nuova istanza della classe `HelloWorld` come parametro `JTCRunnable`.

17 Lancia il nuovo thread, che invoca il metodo `run`.

Capitolo 3

Lavorando con i Thread

3.1 Sincronizzazione

Andiamo a scrivere una classe C++, che può essere usata come buffer di caratteri. Questa classe definisce i metodi `addChar` e `writeBuffer`. `addChar` aggiunge un carattere ad un buffer interno e `writeBuffer` stampa il contenuto del buffer sullo standard output:

```
1  class CharacterBuffer
2  {
3      char* data_;
4      int max_;
5      int len_;
6
7  public:
8
9      CharacterBuffer()
10         : data_(0), len_(0), max_(0)
11     {
12     }
13
14     ~CharacterBuffer()
15     {
16         delete[] data_;
17     }
18
```

```
19      void addChar(char c)
20      {
21          if(len_ == max_)
22          {
23              char* newData = new char[len_ + 128];
24              memcpy(newData, data_, len_);
25              delete[] data_;
26              data_ = newData;
27              max_ += 128;
28          }
29          data_[len_++] = c;
30      }
31
32      void writeBuffer()
33      {
34          cout.write(data_, len_) << flush;
35          len_ = 0;
36      }
37  };
```

3-5 Molti dati sono definiti:

- `data_` è un puntatore di caratteri per bufferizzare i caratteri.
- `max_` è la massima lunghezza del buffer puntato da `data_`.
- `len_` è la corrente lunghezza del buffer, per esempio, il numero dei caratteri validi nel buffer puntato da `data_`. `len_` deve essere minore o uguale a `max_`.

10 Il costruttore inizializza i dati `data_`, `max_` e `len_`.

14 Il distruttore cancella `data_`, liberando la memoria per il buffer.

21-28 Se il buffer è pieno (per esempio, se `len_` è uguale a `max_`), si alloca più memoria. Ciò è fatto allocando un nuovo, buffer di caratteri più grande, copiando il contenuto del buffer esistente nel nuovo buffer, cancellando il vecchio buffer e assegnando il puntatore `data_` al nuovo buffer. Infine `max_` deve essere aggiornato secondo la nuova grandezza del buffer.

29 Un carattere è aggiunto al buffer e `len_` è incrementato di uno.

32-36 Il metodo `writeBuffer` stampa `len_` caratteri dal buffer sullo standard output e resetta a zero `len_`.

Questa classe lavora bene fintanto che c'è solo un thread in esecuzione, ma in ambiente multi-threaded non lavorerà correttamente. Per esempio, se due thread eseguono simultaneamente `addChar`, le cose possono facilmente andare non come vorremmo. Assumiamo che il primo thread è in esecuzione fino a quanto non viene eseguita l'espressione `delete[] data_`. A questo punto il

sistema operativo passa dall'esecuzione del primo thread al secondo thread. Così `max_` non è stato ancora incrementato dal primo thread, la condizione viene ancora soddisfatta e il secondo thread allora accede alla variabile `data_`, che punta ad una parte della memoria già cancellata dal primo thread. Ciò porterà probabilmente al crash del programma.

3.1.1 Thread Sicuri(Safe): Versione in Java

Per risolvere questo problema, Java usa il concetto conosciuto come i *monitor*.
Una versione thread-safe di questo codice in Java può essere scritto come segue:

```
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     }
```

3-4 Sono definiti due campi dati:

- `data_` è un array di caratteri, che mantiene i caratteri bufferizzati.
- `len_` è la lunghezzza corrente del buffer, per esempio, il numero di caratteri validi nel buffer puntati da `data_`.

A differenza della versione del programma in C++, in questo programma non è necessario avere un campo `max_`, in quanto al suo posto può essere usato `data_.length`.

6-14 Se nessun buffer è stato ancora creato o se il buffer è pieno (per esempio, se `len_`è uguale a `data_.length)` un nuovo, buffer più grande è allocato. Ciò è simile alla versione in C++.

15 Un carattere è aggiunto al buffer e `len_` è incrementato di uno.

18-22 Come nell'esempio C++, il metodo `writeBuffer` stampa `len_` caratteri dal buffer sullo standard output e quindi resetta a zero `len_`.

Il solo cambiamento concettuale per fare il programma thread-safe è stato quello di aggiungere la parola chiave `synchronized` per la definizione di `addChar` è `writeBuffer`. In Java ogni oggetto implicitamente ha un monitor associato. All'ingresso di un metodo `synchronized`, il monitor appartenente all'oggetto è bloccato, impedendo l'ingresso ad altri thread a qualsiasi altro metodo sincronizzato dell'oggetto. All'uscita, il monitor è sbloccato, permettendo allora l'accesso ad altri thread. Ciò assicura che lo scenario descritto sopra non si presenterà mai, infatti sarà impossibile per due thread entrare nel metodo `addChar` simultaneamente.

3.1.2 Thread Sicuri(Safe): Versione in C++

JTHREADS/C++ supporta i monitor con due classi: `JTCMonitor` e `JTCSynchronized`.
La classe JTCSynchronized usa il concetto di "inizializzazione è acquisizione" per entrare in possesso del lucchetto del monitor. Il relativo lucchetto del monitor è ottenuto sulla costruzione e viene rilasciato sulla distruzione dell'oggetto.

Qui è presentata la versione C++ dell'esempio thread-safe:

```
1  class CharacterBuffer : public JTCMonitor
2  {
3     char* data_;
4     int len_;
5     int max_;
6
7  public:
8
9     CharacterBuffer()
10       : data_(0), len_(0), max_(0)
11    {
12    }
13
14    ~CharacterBuffer()
15    {
16       delete[] data_;
17    }
18
19    void addChar(char c)
```

```
20      {
21          JTCSynchronized synchronized(*this);
22          if (len_ >= max_)
23          {
24              char* newData = new char[len_+128];
25              memcpy(newData, data_, len_);
26              delete[] data_;
27              data_ = newData;
28              max_ += 128;
29          }
30          data_[len_++] = c;
31      }
32
33      void writeBuffer()
34      {
35          JTCSynchronized synchronized(*this);
36          cout.write(data_, len_) << flush;
37          len_ = 0;
38      }
39  };
```

1 La classe `CharacterBuffer` è ora derivata da `JTCMonitor`. In Java ciò non è necessario, in quanto tutti gli oggetti Java ereditano implicitamente da `java.lang.Object`, che fornisce le funzionalità del monitor.

21,35 I metodi `addChar` e `writeBuffer` sono ora thread-safe. Invece di dichiarare le operazioni come `synchronized` (così come avviene in Java), la funzione prima crea un'istanza di `JTCSynchronized` con l'oggetto monitor di `CharacterBuffer` come argomento.

Tutto ciò che viene fatto per tradurre una classe Java thread-safe (per esempio, sincronizzata) in una classe thread-safe JTHREADS/C++ è:

- Derivare la classe da `JTCMonitor`.
- Rimpiazzare il metodo `synchronized` con il metodo che contiene `JTCSynchronized synchronized(*this)` come prima espressione nel corpo della funzione.

3.2 Blocco di Sincronizzazione

Java non solo supporta la sincronizzazione dei metodi, ma anche la sincronizzazione di blocchi di codice. Per esempio, assumiamo di voler scrivere una classe thread il cui metodo `run` mette una stringa in un oggetto `CharacterBuffer` usando `addChar`.
In Java, ciò potrebbe essere scritto come:

```
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 };
```

1 E' definita una classe `Writer`, che eredita da `Thread.`

6-10 Il costruttore inizializza i campi `buffer_` e `str_`.

12-16 Il metodo run del thread mette la stringa `str_` dentro il buffer, carattere per carattere, usando il metodo `addChar` del buffer.

Comunque questa classe non lavora come noi vorremmo. Supponiamo di lanciare due nuovi, thread, uno per aggiungere "123" al buffer, e uno per aggiungere “abc”:

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

1 Un `CharacterBuffer` è creato.

2,3 Sono creati due thread Writer, uno con argomento "123", e l'altro con argomento “abc”. Entrambi i thread usano lo stesso oggetto `CharacterBuffer`. Il template `JTCHandleT` è spiegato in “Il Template JTCHandleT” alla pagina 33. E' errato assumere che qui si stia usando un semplice puntatore Writer* C++, ma per ora assumiamo che `JTCHandleT<Writer>` e `Writer*` siano la stessa cosa.

4,5 I due thread `Writer` sono lanciati.

Ora consideriamo il seguente scenario: `w1` parte per prima, ma dopo aver scritto "12" nel buffer il sistema operativo passa all'esecuzione del thread `w2`, che scrive "abc". Dopo quindi `w1` continua a scrivere "3". Il risultato è che il buffer ora contiene la sequenza di caratteri "12abc3" invece che "123abc".

Possiamo facilmente evitare ciò, riscrivendo il metodo `run` in modo tale che il lock del monitor `di CharacterBuffer` venga preso prima di cominciare a scrivere nel 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  }
```

3-7 Il ciclo `for` è ora posto in un blocco di codice sincronizzato con il lucchetto del monitor CharacterBuffer. Ciò ci assicurerà che i caratteri sono messi nel buffer nella giusta sequenza.

Questo è chiamato un "blocco di codice sincronizzato," in contrasto al metodo sincronizzato. L'esempio tradotto in JTHREADS/C++ è:

```
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  };
```

16-21 Invece di usare `*this`, `*buffer_` è usato per la sincronizzazione.

3.3 Monitor Statici

In Java è possibile avere metodi statici sincronizzati. Segue un esempio:

```
1  public class StaticCounter
2  {
3      static long counter_;
4
5      public static synchronized void increment()
6      {
7          ++counter_;
8      }
9
10     public static synchronized void decrement()
11     {
12         --counter_;
13     }
14
15     public static synchronized long value()
16     {
17         return counter_;
18     }
19 };
```

Questa classe permette un accesso globale ad un contatore protetto. Questa classe deve essere sincronizzata poichè l'accesso a un valore `long` in Java non è atomico.

Non è possibile ereditare da `JTCMonitor` se funzioni statiche hanno bisogno di essere sincronizzate, così la classe `JTCSynchronized`s richiede `*this` come argomento per il suo costruttore (che non è disponibile con funzioni statiche).

Per risolvere questo problema, un campo statico del tipo `JTCMonitor` è usato per la sincronizzazione delle funzioni statiche.

Questo è l'esempio Java convertito in 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_;
```

4 E' dichiarata una istanza statica di `JTCMonitor`. Questa permette alla classe di essere sincronizzata.

10,16, 22 I metodi sono sincronizzati. Al posto di `*this`, viene usata la variabile statica `mon_`.

Nota che ci sono delle restrizioni sull'uso dei monitor statici. Non è corretto usare un monitor statico prima che l'istanza di `JTCInitialize` sia stata creata. Qualsiasi uso prima della inizializzazione di JTHREADS/C++ avrà un comportamento indefinito. Inoltre, la classe monitor non deve essere usata dopo che l'istanza finale dell'oggetto `JTCInitialize` viene distrutta.

Nota che le classi JTHREADS/C++ che possono essere usate come tipi statici sono solo le classi `JTCMutex`, `JTCRecursiveMutex` e `JTCMonitor`. Tutte le altre non possono essere usate come tipi statici.

3.4 I metodi Wait, Notify e NotifyAll

Come in Java, JTHREADS/C++ offre i metodi `wait`, `notify` e `notifyAll` per la comunicazione fra thread. Come esempio, estendiamo il nostro precedente esempio sulla classe `CharacterBuffer`. Questa volta, noi vogliamo che l'operazione `writeBuffer` si comporti in modo leggermente differente: `writeBuffer` dovrebbe solo stampare il contenuto del buffer se ci sono almeno 80 caratteri nel buffer.

3.4.1 Uso di Wait/Notify con Java

Cominciamo con il riscrivere il metodo `writeBuffer` in Java, usando `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;
16 }
```

1 Il metodo `writeBuffer` deve essere dichiarato `synchronized`. Questo assicura che il lucchetto sul monitor è acquisito all'ingresso del metodo.

3 Il ciclo `while` è eseguito finchè non ci sono almeno 80 caratteri disponibili.

7 `wait` è invocato. Questo rilascia il lucchetto sul monitor (che era stato acquisito all'ingresso del metodo `writeBuffer`) e attende finchè un altro thread invoca o `notify` o `notifyAll` sul monitor.

5,9 E' possibile che `wait` lanci un `InterruptedException`. Perciò questa eccezione deve essere catturata.

Ora cambiamo il metodo `addChar` così che esso invochi `notify` ogni volta che ci sono almeno 80 caratteri nel 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 }
```

1 `addChar` è dichiarato `synchronized` così che il lucchetto del monitor è acquisito. Questo è un requisito per usare `wait`, `notify` o `notifyAll`.

11-12 Se, dopo l'aggiunta di un nuovo carattere, il numero di caratteri nel buffer è uguale o è maggiore di 80, `notify` è invocato. Questo sveglia esattamente un thread che è in attesa sulla `wait`. "Sveglia" in questo contesto significa che è ritornato un thread che attende sulla wait e implicitamente si blocca di nuovo il monitor, assicurandoci che solo un thread alla volta può essere eseguito da un metodo `synchronized`.

La differenza fra `notify` e `notifyAll` è che `notify` sveglia solo un thread, mentre `notifyAll` sveglia tutti i thread in attesa. Se c'è più di un thread in attesa, e viene usato `notify`, un thread casuale i svegliato. Se più di un thread è in attesa e viene usato `notifyAll`, allora tutti i thread sono svegliati, ma l'ordine con cui i thread in attesa sono svegliati dalla `wait` è casuale. Ricordatevi che solo un thread alla volta può essere ritornato dalla `wait`, in quanto il ritorno dalla `wait` richiede che il lucchetto del monitor sia acquisito.

In questo esempio, viene usato `notify` in quanto sappiamo che c'è solo un thread in attesa sulla wait. Comunque, non importa se viene usato `notify` o `notifyAll` poichè il numero dei caratteri è resettato a zero una volta che un thread è stato ritornato dalla `wait`.
Quando gli altri thread vengono ritornati successivamente dalla `wait`, loro andranno ancora in wait a causa del ciclo while.

3.4.2 Uso di Wait/Notify in C++

Vediamo ora come questo esempio viene tradotto in 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 }
```

3 Come nell'esempio Java il metodo `writeBuffer` deve essere sincronizzato. Invocando `wait`, `notify` o `notifyAll` senza avere il monitor bloccato si ha come risultato l'eccezione `JTCIllegalMonitorStateException`.

4 Come nell'esempio Java, il ciclo `while` è eseguito finchè ci sono almeno 80 caratteri disponibili.

8 `wait` è chiamato esattamente allo stesso modo dell'esempio in Java. Allora si entra in possesso del lucchetto del monitor (che è stato acquisito con la sincronizzazione attraverso la classe `JTCSynchronize`) e si attende fino alla notificazione.

5,9 L'equivalente per `java.lang.InterruptedException` è l'eccezione JTHREADS/C++ `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  }
```

3 `addChar` è reso sincronizzato.

12-13 `notify` è chiamato se sono disponibili 80 caratteri, e sveglia un thread in attesa sulla wait.

Come potete vedere, la semantica di `wait`, `notify` e `notifyAll` in JTHREADS/C++ è esattamente la stessa di quella in Java.

3.5 I metodi Stop e Suspend

Abbiamo già introdotto il metodo `start` della classe `JTCThread`. L'opposto di start è stop, che termina l'esecuzione di un thread. Oltre a stop c'è anche il metodo suspend che sospende l'esecuzione di un thread fino a quanto non viene chiamato `resume`.

3.5.1 Punti di controllo

`stop` e `suspend` non terminano o sospendono un thread immediatamente, poichè ciò non è supportato da ogni thread API a basso livello (per esempio, i thread POSIX). Invece, la libreria JTHREADS/C++ usa il concetto di *punto di controllo* per implementare i metodi `suspend` e `stop`. Ciò è simile al punto di cancellazione, concetto usato nella libreria dei thread POSIX. Se `suspend` o `stop` viene chiamato dall'esterno del thread che deve essere sospeso o stoppato, il thread viene marcato come *control-pending*. Quando questo thread chiama un metodo che è un *punto di controllo* il thread viene rspettivamente stoppato o sospeso.

Una volta che un thread viene sospeso, l'esecuzione per questo thread è arrestata finchè esso non viene ripreso. Se un thread viene stoppato, viene sollevata l'eccezione `JTCThreadDeath`. Se questa eccezione viene catturata dal codice utente, essa deve essere risollevata per assicurare la terminazione corretta del thread.

I punti di controllo nella libreria JTHREADS/C++ sono:

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

3.5.2 Implementazione di un metodo per la terminazione di un Thread

La versione Java dei metodi stop, resume e suspend sono deprecai [4]. Il motivo è che sotto alcune circostanze, questi metodi possono portare al deadlock della Java Virtual Machine.

L'implementazione JTHREADS/C++ di `stop`, `resume` e `suspend` è completamente portabile e non soffre dei stessi difetti di Java. Comunque, allo scopo di mantenere il vostro codice sorgente combatibile con Java, si raccomanda di fornire le classe thread con i propri metodi per la terminazione. Come esempio andiamo a vedere ancora una volta la classe `CharacterBuffer`. Vogliamo ora avere thread separati, che attendono fino a quanto 80 caratteri diventano disponibili. Allora si stampano questi 80 caratteri sullo standard output, resettando il contenuto del buffer e ricominciando da capo. Il thread potrebbe stopparsi se un metodo di terminazione viene chiamato sulla classe `CharacterBuffer`. Possiamo scrivere questa classe come segue:

```
1  class CharacterBuffer : public JTCMonitor, public JTCThread
2  {
3     char* data_;
4     int max_;
5     int len_;
6     bool done_;
7
8  public:
9
10    CharacterBuffer()
11       : data_(0), len_(0), max_(0), done_(false)
12    {
13    }
14
15    ~CharacterBuffer()
16    {
17       delete[] data_;
18    }
19
20    void addChar(char c)
21    {
22       JTCSynchronized synchronized(*this);
23       if (len_ >= max_)
24       {
25          char* newData = new char[len_+128];
26          memcpy(newData, data_, len_);
27          delete[] data_;
28          data_ = newData;
```

```
29              max_ += 128;
30          }
31          data_[len_++] = c;
32          if(len_ >= 80)
33              notify();
34      }
35
36      virtual void run()
37      {
38          JTCSynchronized synchronized(*this);
39          while(true)
40          {
41              while(!done_ && len_ < 80)
42              {
43                  try
44                  {
45                      wait();
46                  }
47                  catch(const JTCInterruptedException&)
48                  {
49                  }
50              }
51              if(done_)
52                  break;
53              cout.write(data_, len_) << flush;
54              len_ = 0;
55          }
56      }
57
58      void terminate()
59      {
60          JTCSynchronized synchronized(*this);
61          done_ = true;
62          notify();
63      }
64  };
```

1 La classe `CharacterBuffer` è ora anche derivata da `JTCThread` allo scopo di fornire thread separati per stampare il contenuto del buffer.

6,11 Aggiungiamo un flag `done_`, inizialmete settato a `false` nel costruttore.

20-34 Nessuno modifica è stata fatta al metodo `addChar`. L'implementazione è la stessa della sezione 3.4.2.

36-56 Il metodo `writeBuffer` è obsoleto. Ora invece abbiamo un metodo `run`, che stampa il contenuto del buffer in un ciclo infinito.

41-50 Questa è simile all'implementazione mostrata nella sezione 3.4.2. Comunque, il ciclo `while` ora non solo controlla se 80 caratteri sono disponibili, ma anche se il flag `done_` è settato a `true`.

50-51 Se il ciclo `while` innestato termina perchè `done_` viene settato a `true`, viene invocato. break. Ciò causa l'uscita del thread dal ciclo while più esterno, per ritornare al metodo run e terminare.

53-54 Se il ciclo `while` termina per qualche altra ragione, ci sono ora 80 caratteri disponibili, che sono stampati sullo standard output.

58-63 Il metodo `terminate` serve come una alternativa per `stop`. Esso prima acquisisce il lucchetto del monitor con un istanza di `JTCSynchronize`, quindi setta il flag done_ a `true` e sveglia un thread in attesa sulla wait.

3.6 I metodi Join e IsAlive

In alcune applicazioni è necessario attendere per la terminazione dei thread. Per esempio, se un insieme di thread sta realizzando un complesso calcolo parallelo, l'applicazione potrebbe attendere per il calcolo, per essere completato prima di continuare.

Come esempio, assumiamo di volere che la funzione `main` del nostro programma “Hello World” del Capitolo 2 attenda per il thread `HelloWorld` prima di terminare. Un modo per far ciò è il seguente:

```
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 }
```

5 E' assolutamente necessario usare qui `JTCThreadHandle` al posto di `JTCThread*`. Vedete “Introduzione “Handles”” alla pagina 31 per ulteriori informazioni. Per ora, immaginiamo un `JTCThreadHandle` come se fosse un `typedef` per `JTCThread*`.

7,8 Il metodo `isAlive` è usato per attendere fino alla terminazione del thread. `isAlive` ritorna `true` se il thread è vivo (cioè, se è in esecuzione e non ha ancora terminato), o `false` in caso contrario.

Il codice sotto ha ovvi problemi di busy-looping, che dovrebbero essere evitati a tutti i costi. Fortunatamente esiste un approccio alternativo: `join` può essere usato per questo scopo. Questo metodo consente di attende fino a che il thread.

```
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 }
```

7 Il metodo `join` è usato per attendere fino a che il thread muore.

Comunque, questo esempio presenta un problema. Il metodo `join` può lanciare l'eccezione `JTCInterruptedException`. Perciò questo esempio dovrebbe essere riscritto come segue:

```
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 }
```

9-15 `join` è chiamato sul thread, il quale consente di attendere finchè il thread termina. Comunque, se è lanciata l'eccezione `JTCInterruptedException` viene ignorata.

17 Ciò assicura che il ciclo è terminato se nessuna `JTCInterruptedException` è stata lanciata, per esempio, se il thread non è più vivo.

Capitolo 4

Gestione della memoria

4.1 Contatore di riferimento

Si è potuto pensare che tutti gli esempi “Hello World” del capitolo 2 comportano perdita di memoria, in quanto gli oggetti thread soo creati con `new` ma non sono mai cancellati con `delete`. Comunque è sbagliato pensare ciò. Perchè? Il trucco sta nel “contatore di riferimento”.

Ogni oggetto `JTCThread` (e anche oggetti `JTCThreadGroup` e `JTCRunnable`) hanno un contatore di riferimento. Quando un nuovo thread è creato, questo contatore è settato a 1. Quando il thread termina (per esempio, si ritorna dal metodo `run`), il contatore è decrementato di 1. Ogni volta che il valore del contatore diventa 0, l'oggetto thread è cancellato con `delete`.

Allora nel nostro esempio “Hello World” il contatore di riferimento non è mai incrementato, il contatore di riferimento diventa 0 ogni volta che si ritorna dal metodo `run`, e ciò significa che l'oggetto thread è cancellato subito dopo la terminazione del thread - così non c'è perdita di memoria.

Uno svantaggio di usare il contatore di riferimento è che non è possibile allocare oggetti con contatori di riferimento sullo stack. Si può solo allocare loro con `new` (sull'heap), cosicchè saranno cancellati con `delete` non appena il contatore di riferimento diventa 0.

4.2 Introduzione agli “Handles”

Forse pensate che il contatore di riferimento è abbastanza complicato, perchè ora bisogna ricordare quando incrementare o decrementare il contatore di riferimento dell'oggetto. Comunque, non siamo in questo caso. JTHREADS/C++ fornisce delle classi “handle” (qualche

volta chiamate “puntatori veloci”) che si prendono cura di incrementare e decrementare il contatore di riferimento per te.

Vediamo l'esempio di pagina 28 da “I metodi Join a e IsAlive”. Qui abbiamo detto che è assolutamente necessario usare `JTCThreadHandle` al posto di `JTCThread*`. Ora scopriamo il segreto che sta dietro.

Consideriamo come abbiamo scritto l'esempio senza `JTCThreadHandle`:

```
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;
}
```

Questo esempio è sbagliato e il programma andrà sicuramente in errore. Quando il thread termina il suo contatore di riferimento è decrementato da 1 a 0 e così l'oggetto thread è cancellato con `delete`. Comunque, noi stiamo ancora attendendo con `join` il thread e verificando se è vivo usando `isAlive` anche se l'oggetto thread è già stato cancellato.

Allora si ha che il contatore di riferimento è incrementato di alla `new` e decrementato di 1 dopo il ciclo `while`. Ciò assicura che il contatore di riferimento va a zero *dopo* che i metodi `join` e `isAlive` sono chiamati.

Questo è quello che l'"handle" fa per te. Se assegnate un oggetto thread a un "handle", questo incrementa di 1 il contatore di riferimento dell'oggetto thread. Stessa cosa succede su si assegna un "handle" ad un altro"handle". Se un "handle" è distrutto, il distruttore dell'"handle" decrementa il contatore di riferimento dell'oggetto che esso punta a 1.

Quindi per l'esempio sopra, abbiamo che se si sostituisce `JTCThread*` con `JTCThreadHandle`, il contatore di riferimento dell'oggetto thread diventerà diventerà dopo la new 2 invece che 1, perchè l'"handle" incrementa il contatore di 1. Dopo che il thread ha terminato, il contatore è portato a 1, e così l'oggetto thread non viene cancellato ject is not deleted, cosicchè ciò è tanto sicuro come usare le operazioni `isAlive` o `join` sull'oggetto thread. Quando l'"handle" è distrutto alla fine della funzione `main`, il distruttore dell'"handle" decrementa il contatore dell'oggetto thread di 1, cosicchè l'oggetto thread è cancellato.

4.3 Il Template JTCHandleT

JTHREADS/C++ fornisce le seguenti classe "handle":

- `JTCThreadHandle` come sostituto per `JTCThread*`.
- `JTCRunnableHandle` come sostituto per `JTCRunnable*`.
- `JTCThreadGroupHandle` come sostituto per `JTCThreadGroup*`.

Questi classi sono tutte `typedef`s per un tipo "handle" più generale possibile scritto come un template C++:

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

Nel caso si vuole accedere ai metodi da una classe derivata da `JTCThread` (o da `JTCRunnable`) bisogna definire un proprio tipo "handle". Per esempio, vediamo quello di pagina 26 dove è definito un metodo `terminate`. Se si vuole chiamare questo metodo, non possiamo usare `JTCThreadHandle` così come mostrato sotto:

```
JTCThreadHandle t = new CharacterBuffer;
... // fa qualcosa con il CharacterBuffer
t -> terminate(); // Questo non funziona, il compilatore darà errore
```

Cosi come non si può usare `JTCThread*` per accedere ai metodi dalla classe derivata da `JTCThread`, allo stesso modo non si può usare nemmeno `JTCThreadHandle`. Bisogna usare la classe "handle" per `CharacterBuffer*`. Ciò può essere fatto usando `JTCHandleT`:

```
typedef JTCHandleT<CharacterBuffer> CharacterBufferHandle;
CharacterBufferHandle t = new CharacterBuffer;
...//fa qualcosa con ilCharacterBuffer
t -> terminate(); // Questo lavora
```

4.4 Regole pratiche

Quando si usa JTHREADS/C++, in genere si mantengono le seguenti regole:

- Usare sempre tipi "handle" invece che semplici puntatori C++. La sola eccezione può essere fatta se si è assolutamente certi che il puntatore all'oggetto thread dopo la chiamata `starte` non viene usato in altrove.
- Mai allocare oggetti thread, oggetti runnable o oggetti di "thread group" sullo stack. Usare sempre la `new`.
- Mai cancellare oggetto thread, oggetti runnable o oggetti "thread group" con `delete`. Loro devono essere cancellati automaticamente.
- Definite il tipo proprio di "handle" per l'uso del template `JTCHandleT` se si deve accedere ai metodi della classe derivata da `JTCThread`, `JTCThreadGroup` o `JTCRunnable`.

Finchè si seguono queste regole basi, la gestione della memoria in JTHREADS/C++ è potenzialmente automaticamente.

La cosa carina circa il contatore di riferimento e le classi "handle" è che esse rendono JTHREADS/C++ molto simile a Java. Il contatore di riferimento simula il garbage collector di Java, e gli "handle" simulano i riferimenti Java.

CAPITOLO 5

Classi di riferimento

5.1 JTCInitialize

Una istanza di questa classe deve essere istanziata prim di usare JTHREADS/C++. Se nessuna istanza di questa classe è stata creata, la libreria JTHREADS/C++ non lavorerà correttamente.

`JTCInitialize` può essere istanziata più volte. Comunque, solo la prima istanza ha effetto. Quando l'ultima istanza di `JTCInitialize` è distrutta, il distruttore aspetterà che tutti i thread lanciati terminino.

`JTCInitialize` interpreta argumenti che cominciano con `-JTC`. Tutti questi argomenti, passati attraverso i parametri `argc` e `argv`, sono rimossi automaticamente dalla lista degli argomenti.

5.1.1 JTCOptions

Le seguenti opzioni JTHREADS/C++ possono essere usate:

-JTCversion

Mostra la versione di JTHREADS/C++.

*-JTCss **stack-size***

Questa opzione setta la dimensione dello stack del thread a `stack-size` kilobyte.

5.1.2 Costruttori

JTCInitialize

`JTCInitialize()`

Inizializza la libreria JTHREADS/C++.

Throws

`JTCSystemCallException` - Indica il fallimento di una chiamata di sistema.

JTCInitialize

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

Inizializza la libreria JTHREADS/C++ e interpreta gli argomenti che cominciano con `-JTC`.

Throws

`JTCSystemCallException` - Indica il fallimento di una chiamata di sistema.

`JTCInitializeError` - Indica che è stata specificata una opzione o un argomento non valido.

5.1.3 Funzioni

waitTermination

`void waitTermination()`

Attende che tutti i thread terminino.

initialized

`static bool initialized()`

Determina se la libreria JTHREADS/C++ è stata inizializzata.

Return

`true` se JTHREADS/C++ è stata inizializzata e `false` altrimenti.

5.2 JTCAdoptCurrentThread

Quando si fanno integrazione con altre librerie, è necessario chiamare i metodi JTHREADS/C++ da thread che non sono stati creati usando JTHREADS/C++. In questa situazione, il thread deve creare una istanza di `JTCAdoptCurrentThread` prima di usare qualsiasi altra classe JTHREADS/C++ .Se non si istanzia `JTCAdoptCurrentThread` il comportamento risulterà essere indefinito.

5.2.1 Costruttori

JTCAdoptCurrentThread

```
JTCAdoptCurrentThread()
```

Informa la libreria JTHREADS/C++ sull'esistenza di questo thread.

Throws

`JTCSystemCallException` - Indica il fallimento di una chiamata di sistema.

5.3 JTCThread

Questa classe è usata per creare l'esecuzione di un nuovo thread. Il thread può essere o aggiunto da una classe derivata da `JTCThread` e fare l'"overriding" del metodo `run`, o essere un oggetto di una classe derivata da `JTCRunnable` e passato al costruttore`JTCThread`

5.3.1 Costruttori

JTCThread

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

Crea un nuovo oggetto thread con un oggetto di riferimento e un nome.

Parametri

`target` - L'oggetto il cui metodo `run` è invocato quando `start` è chiamato. Se nessun oggetto è specificato, il metodo `run` dell'oggetto thread deve essere ridefinito in una classe derivata.

`name` - Il nome del thread. Se nessun nome è specificato, viene usato un nome di default. Questo nome di default è la stringa "`thread-`" unita all'id del thread. L'id del thread è un

identificatore specifico generato dal sistema operativo quando un nuovo thread è creato. Gli sviluppatori di applicazioni sono incoraggiati all'uso della classe `JTCThreadId` per referenziare gli id dei thread.

Throws

`JTCSystemCallException` - Indica il fallimento di una chiamata di sistema.

JTCThread

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

Crea un nuovo oggetto thread con un nome.

Parametri

`name` - Il nome del thread.

Throws

`JTCSystemCallException` - Indica il fallimento di una chiamata di sistema.

JTCThread

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

Crea un nuovo oggetto thread appartenente ad un gruppo, con un oggetto di riferimento e un nome

Parametri

`group` - Il gruppo thread.

`target` - L'oggetto il cui metodo `run` è invocato quando viene chiamato `start`. Se non viene specificato nessun oggetto di riferimento, il metodo `run` dell'oggetto thread deve essere ridefinito in una classe derivata.

`name` - Il nome del thread. Se nessun nome è specificato, viene usato un nome di default. Questo nome di default è la stringa"thread-" unita all'id del thread. L'id del thread è un identificatore specifico generato dal sistema operativo quando un nuovo thread è creato.
Gli sviluppatori di applicazioni sono incoraggiati all'uso della classe JTCThreadId per referenziare gli id dei thread.

Throws

`JTCSystemCallException` - Indica il fallimento di una chiamata di sistema.

JTCThread

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

Crea un nuovo oggetto thread appartenente ad un gruppo, con un nome.

`group` - Il gruppo thread.

name- Il nome del thread. Se nessun nome è specificato, viene usato un nome di default. Questo nome di default è la stringa"thread-" unita all'id del thread. L'id del thread è un identificatore specifico generato dal sistema operativo quando un nuovo thread è creato. Gli sviluppatori di applicazioni sono incoraggiati all'uso della classe JTCThreadId per referenziare gli id dei thread.

Throws

`JTCSystemCallException` - Indica il fallimento di una chiamata di sistema.

5.3.2 Member Functions

getThreadGroup

```
JTCThreadGroupHandle getThreadGroup()
```

Ritorna un oggetto per la manipolazione del gruppo thread al quale questo oggetto thread appartiene.

Return

Un oggetto per la manipolazione del gruppo thread.

setName

```
void setName(const char* name)
```

Setta il nome dell'oggetto thread.

Parametri
name - Il nome del thread. Se nessun nome è specificato, viene usato un nome di default. Questo nome di default è la stringa"thread-" unita all'id del thread. L'id del thread è un

identificatore specifico generato dal sistema operativo quando un nuovo thread è creato. Gli sviluppatori di applicazioni sono incoraggiati all'uso della classe JTCThreadId per referenziare gli id dei thread.

getName

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

Ritorna il nome dell'oggetto thread.

Return

Il nome dell'oggetto thread.

suspend

```
void suspend()
```

Sospende l'esecuzione del thread. Se il chiamante è lo stesso oggetto thread, il thread è sospeso immediatamente. Se il chiamante è un altro thread, il thread è sospeso subito dopo l'ingresso ad un altro punto di controllo (vedere “I metodi Stop e Suspend” pagina 25).

Throws

`JTCSystemCallException` - Indica il fallimento di una chiamata di sistema.

resume

```
void resume()
```

Riprende l'esecuzione del thread. L'esecuzione del thread è ripresa immediatamente.

Throws

`JTCSystemCallException` - Indica il fallimento di una chiamata di sistema.

stop

```
virtual void stop()
```

Arresta l'esecuzione del thread. Se il chiamante è lo stesso oggetto thread, il thread è arrestato immediatamente. Se il chiamante è un altro thread, il thread è arrestato subito dopo l'ingresso dopo l'ingresso ad un altro punto di controllo (vedere“I metodi Stop e Suspend” pagina 25).

Throws

`JTCSystemCallException` - Indica il fallimento di una chiamata di sistema.

start

```
void start()
```

Avvia l'esecuzione di un thread. Se il thread è stato creato con un oggetto di riferimento, per esempio con un oggetto di una classe derivata da `JTCRunnable`, il metodo run dell'oggetto dell'oggetto di riferimento viene invocato. Se non c'è nessun oggetto di riferimento, il metodo `run` dello stesso oggetto thread è invocato. In questo caso, una classe derivata da `JTCThread` con un metodo `run` ridefinito dovrebbe essere usata.

Throws

`JTCSystemCallException` - Indica il fallimento di una chiamata di sistema.

`JTCIllegalStateException` - Viene lanciata se il thread è gia stato avviato.

run

```
virtual void run()
```

Questo metodo viene invocato quando è chiamato `start`. Se l'oggetto thread è stato costruito con un associato oggetto di riferimento `JTCRunnable`, il metodo run dell'oggetto di riferimento viene invocato. In altre circostanze, il metodo run potrebbe essere ridefinito in una classe derivata da `JTCThread`. Se run termina a causa di un'eccezione non catturata, allora il metodo `uncaughtException` del gruppo thread del thread è chiamato.

isAlive

```
bool isAlive() const
```

Questo metodo determina se il thread è vivo o meno.

Returns

`true` se il thread è vivo, `false` altrimeti.

join

```
void join()
```

Attende che il thread termini.

Throws

`JTCSystemCallException` - Indica il fallimento di una chiamata di sistema.

join

```
void join(long millis)
```

Attende la terminazione del thread al più per `millis` millisecondi.

Throws

`JTCSystemCallException` - Indica il fallimento di una chiamata di sistema.

`JTCIllegalArgumentException` - Lanciata se il valore di `millis` è negativo.

join

```
void join(long millis, int nanos)
```

Attende la terminazione del thread al più per millis millisecondi e nanos nanosecondi.

Throws

`JTCSystemCallException` - Indica il fallimento di una chiamata di sistema.

`JTCIllegalArgumentException` - Lanciata se il valore di `millis` è negativo, o se il valore di `nanos` non è nel range 0 - 999999.

setPriority

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

Setta la priorità del thread ad un nuovo valore.

Parametri

`newPri` - La nuovo priorità del thread.

Throws

`JTCSystemCallException` - Indica il fallimento di una chiamata di sistema.

getPriority

```
int getPriority() const
```

Ritorna la priorità del thread.

Return

La priorità del thread.

Throws

`JTCSystemCallException` - Indica il fallimento di una chiamata di sistema.

setAttrHook

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

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

Setta/ottiene un *aggancio(hook)* che sarà utilizzato per inizializzare gli attributi del thread POSIX. Nota: questo metodo è disponibile solo per sistemi che supportano i thread POSIX.

Parametri

`hook` - La funzione che sarà chiamata per ricercare gli attributi del thread POSIX prima della creazione di ogni thread.

oldHook - Parametro opzionale nel quale il precedente "hook" è ritornato. Applicazioni potrebbero chiamare questa funzione all'interno del nuovo hook. In sostanza, gli "hook"potrebbero essere concatenati.

setRunHook

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

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

Setta/ottiene un aggancio (hook) run che potrebbe essere usato per settare qualsiasi informazione specifica di una applicazione durante la creazione del thread. La funzione "hook" deve essere chiamata thread->run() per l'effettivo run del thread.

Parameters

hook - La funzione che sarà chiamata alla creazione del thread.

oldHook - Parametro opzionale nel quale il precedente "hook" è ritornato. Applicazioni potrebbero chiamare questa funzione all'interno del nuovo hook. In sostanza, gli "hook"potrebbero essere concatenati.

setStartHook

```
typedef void (*JTCStartHook)()
```

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

Setta/ottiene un aggancio da usare per settare un'informazione specifica di un thread.

Parametri

hook - La funzione che sarà chiamata alla creazione del thread.

oldHook - Parametro opzionale nel quale il precedente "hook" è ritornato. Applicazioni potrebbero chiamare questa funzione all'interno del nuovo hook. In sostanza, gli "hook" potrebbero essere concatenati.

enumerate

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

Copia ogni thread attivo nel gruppo e sottogruppo di thread di questo thread in un array `list.` Se più di len elementi sono presenti, la lista è troncata.

Parametri

`list` - L'array nel quale tutti i thread nel gruppo e sottogruppo di thread di questo thread sono copiati.

`len` - Il numero di elementi `JTCThreadHandle*` nella `list`.

Returns

Il numero di thread ritornati nella `list`.

getPriority

```
int getPriority() const
```

Ritorna la priorità del thread.

Return

La priorità del thread.

Throws

`JTCSystemCallException` - Indica il fallimento di una chiamata di sistema.

setAttrHook

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

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

Setta/ottiene a *aggancio(hook)* che sarà utilizzato per inizializzare gli attributi del thread POSIX. Nota: questo metodo è disponibile solo per sistemi che supportano i thread POSIX.

Parametri

`hook` - La funzione che sarà chiamata per ricercare gli attributi del thread POSIX prima della creazione di ogni thread.

oldHook - Parametro opzionale nel quale il precedente "hook" è ritornato. Applicazioni potrebbero chiamare questa funzione all'interno del nuovo hook. In sostanza, gli "hook" potrebbero essere concatenati.

setRunHook

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

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

Setta/ottiene un aggancio (hook) run che potrebbe essere usato per settare qualsiasi informazione specifica di una applicazione durande la creazione del thread.
La funzione "hook" deve essere chiamata `thread -> run()` per l'effettivo run del thread.

getId

```
JTCThreadId getId() const
```

Returna l'id del thread.

Returns

Il thread id del thread.

5.3.3 Campi DATI membri

JTC_MIN_PRIORITY

```
const int JTC_MIN_PRIORITY
```

Una costante per la priorità minima che il thread può avere.

JTC_NORM_PRIORITY

```
const int JTC_NORM_PRIORITY
```

Una costante per la priorità di default del thread.

JTC_MAX_PRIORITY

```
const int JTC_MAX_PRIORITY
```

Una costante per la massima priorità che il thread può avere.

5.3.4 Funzioni correlate

operator<<

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

Stampa l'id del thread sull'output stream `os`. Il formato dell'output del campo thread-id è specificamente piatto.

Parametri

`os` - Output stream nel quale si inserisce l'id del thread.

`thr` - Referimento al thread.

Returns

L'output stream `os`.

5.4 JTCRunnable

Questa classe è offerta come un metodo alternativo per fornire le funzionalità di thread. Per usare questa classe, bisogna scrivere una sottoclasse e fornire una definizione per il metodo run. Una istanza di questa classe dovrebbe essere fornita come un argomento al costruttore `JTCThread`. Quando il thread è avviato, il metodo `run` dell'istanza sarà invocato.

5.4.1 Funzioni

run

```
virtual void run()
```

Chiamata quando il metodo `start` è chiamato sull'associato oggetto thread.

5.5 JTCThreadGroup

Questa classe rappresenta una collezione di thread, e altri gruppi di thread. Il gruppo di thread forma un albero, con la radice sul gruppo thread di sistema. Nuovi thread di default appartengono al gruppo thread del suo thread padre. Un gruppo di thread può opzionalmente essere un gruppo thread demone, che si distrugge automaticamente dopo che tutti i thread sono terminati e tutti i sotto gruppi sono distrutti. Un gruppo di thread appena creato eredita lo stato del demone dal thread padre. Il gruppo thread radice non è un thread demone.

5.5.1 Constructors

JTCThreadGroup

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

Crea un nuovo gruppo di thread con il nome fornito.Il padre del nuovo gruppo di thread è il thread corrente.

Parametri

`name` - Il nome del gruppo di thread.

Throws

`JTCIllegalThreadStateException` - Lanciata se il padre del gruppo thread è stato distrutto.

JTCThreadGroup

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

Crea un nuovo gruppo di thread con un nome `name` e un thread padre `group`.

Parametri

`group` - Il padre del gruppo di thread.

`name` - Il nome del gruppo di thread.

Throws

`JTCIllegalThreadStateException` - Lanciata se il padre del gruppo di thread è stato distrutto.

5.5.2 Funzioni membro

getName()

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

Ritorna il nome del gruppo di thread.

Returns

Il nome del gruppo di thread.

getParent

```
JTCThreadGroupHandle getParent() const
```

Ritorna il padre del gruppo di thread. Se il gruppo di thread è il gruppo di thread radice, sarà ritornato un puntatore a null.

Return

Il padre del gruppo di thread.

isDaemon

```
bool isDaemon() const
```

Ritorna il flag del demone per questo gruppo di thread. Se il flag è true, il gruppo di thread è distrutto una volta che tutti i thread sono terminati e i sottogruppi sono vuoti.

Return

Il valore del flag del demone.

setDaemon

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

Setta il flag del demone per questo gruppo di thread. Se il flag demone è `true`, il gruppo di thread è distrutto una volta che tutti i thread sono terminati e i sottogruppi sono vuoti.

Parametri

`daemon` - Il nuovo valore per il flag del demone.

uncaughtException

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

Questo metodo è chiamato se un `JTCThread::run()` finisce a causa di una eccezione `JTCException` non catturata. Di default, se il gruppo di thread ha un padre, questo metodo invoca il metodo `uncaughtException` del padre, altrimenti esso visualizza l'eccezione per `stderr`.

Parametri

`t` - Il thread che lancia il `JTCException`.

`e` - L'eccezione non catturata.

uncaughtException

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

Questo metodo è chiamato se un JTCThread::run() finisce a causa di una eccezione JTCException non catturata. Di default, se il gruppo di thread ha un padre, questo metodo

invoca il metodo uncaughtException del padre, altrimenti esso visualizza l'eccezione per stderr.

Parametri

`t` - Il thread che lancia la `JTCException`.

getMaxPriority

```
int getMaxPriority() const
```

Ritorna la massima priorità permessa per i thread in questo gruppo di thread.

Returns

La massima priorità di questo gruppo di thread.

stop

```
void stop()
```

Arresta tutti i thread in questo gruppo di thread chiamando il metodo `stop` su tutti i thread del gruppo di thread e di tutti i suoi sottogruppi.

resume

```
void resume()
```

Riprende tutti i thread nel gruppo di thread chiamando il metodo `resume` su tutti i thread nel gruppo e di tutti i suoi sottogruppi.

suspend

```
void suspend()
```

Sospende tutti i thread nel gruppo di thread chiamando il metodo `suspend` su tutti i thread del gruppo e di tutti i suoi sottogruppi.

isDestroyed

```
bool isDestroyed() const
```

Determina se il gruppo di thread è stato distrutto. Un gruppo di thread è distrutto una volta che tutti i thread nel gruppo di thread e in tutti i sottogruppi sono terminati.

Returns

`true` se il gruppo di thread è stato distrutto, `false` altrimenti.

destroy

```
void destroy()
```

Distrugge questo gruppo di thread e tutti i suoi sotto gruppi. Il gruppo di thread non deve contenere nessun thread attivo.

Throws

`JTCIllegalThreadStateException` - Se il gruppo di thread ha thread attivi, o che sono già distrutti.

setMaxPriority

```
void setMaxPriority(int pri)
```

Setta la massima priorità che i thread nel gruppo di thread e nei suoi sotto gruppi potrebbero avere. I thread nel gruppo di thread che hanno una priorità più alta non sono influenzati. Cioè, la loro priorità non viene abbassata.

parentOf

```
bool parentOf(JTCThreadGroupHandle g)
```

Ritorna `true` se il gruppo di thread è il padre del gruppo di thread `g`.

Returns

`true` se questo thread è un padre di `g`, `false` altrimenti.

activeCount

```
int activeCount() const
```

Ritorna il numero di thread attivi in questo gruppo di thread, e di tutti i suoi sotto gruppi.

Returns

Il numero di thread attivi.

activeGroupCount

```
int activeGroupCount() const
```

Ritorna il numero di gruppi di thread attivi in questo gruppo.

Returns

Il numero di gruppi di thread attivi.

enumerate

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

Copia i puntatori a ogni thread attivo in questo gruppo di thread nell'array `list`. `activeCount` può essere usato per dare una stima di quanto grande l'array dovrebbe essere. Se sono presenti più di `len` thread attivi, i thread rimanenti sono ignorati in silenzio. La ragione che lo sviluppatore non può determinare precisamente il numero di thread è che i thread possono essere aggiunti e rimossi da gruppo di thread allo stesso tempo della loro enumerazione.

Parametri

`list` - L'array nel quale l'insieme dei thread attivi viene copiato.

`len` - La lunghezza dell'array `list`.

`recurse` - Se settato a true, sono enumerati anche i thread attivi dei sotto gruppi.

Returns

Il numero dei thread copiati nell'array.

enumerate

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

Copia gli "handle" per ogni sottogruppo attivo di questo gruppo di thread nell'array `list`. `activeGroupCount` può essere usato per determinare quanto grande l'array deve essere. Se sono presenti più di `len` sottogruppi attivi, i rimanenti sottogruppi sono ignorati in silenzio.

Parametri

`list` - L'array nel quale sono copiati gli "handle" dei gruppi di thread.

`len` - La lunghezza dell'array.

`recurse` - Se settato a true, sono enumerati anche i figli dei sottogruppi.

Returns

Il numero degli "handle" copiati nella `list`.

list

```
void list()
```

Produce sullo stream `cout` l'insieme dei thread e dei sotto gruppi.

list

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

Produce sullo stream `os` l'insieme dei thread e sotto gruppi. Usa `indent` spazi per la separazione.

5.5.3 Funzioni

operator<<

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

Stampa una stinga rappresentante il gruppo di thread sull'output stream `os`. Questo chiama `g.list(os, 4)`.

Parametri

`os` - L'outputstream nel quale inserire il thread id.

`g` - Riferimento al gruppo di thread.

Returns

L'output stream `os`.

5.6 JTCHandleT

La libreria JTHREADS/C++ non può conoscere quando cancellare una istanza di `JTCThread`, `JTCRunnable`, e `JTCThreadGroup`. Una soluzione a questo dilemma è forzare l'applicazione sviluppata a cancellare le istanze di queste classe quando ci si assicura che non vengono più usate. Comunque, questo è un errore prono. Fortunatamente c'è una soluzione ben conosciuta per questo problema - il contatore di riferimento (vedere [5] p. 782). Questo non è necessario in Java da quanto esiste il garbage collection. L'idea base è di contare il numero di riferimento agli oggetti, e cancellare gli oggetti quando il contatore di riferimento va a zero. Per semplificare il contatore di riferimento, una classe "handle" ciene usata per incrementare il contatore di riferimento quando si costruisce, e decrementare il contatore di riferimeto quando si dustrugge.

Le classi `JTCThreadGroupHandle`, `JTCThreadHandle`, e `JTCRunnableHandle` sono di utilità `typedef` di questa classe template.

5.6.1 Costruttori

JTCHandleT

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

Crea un "handle" che riferisce all'oggetto `tg`.

Parametri

`tg` - L'oggetto riferito.

JTCHandleT

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

Crea un "handle" che riferisce all'oggetto riferito da `rhs`.

Parametri

`rhs` - L"handle" dal quale ricercare l'oggetto.

Parametri

`list` - L'array nel quale sono copiati gli "handle" dei gruppi di thread.

`len` - La lunghezza dell'array.

`recurse` - Se settato a true, sono enumerati anche i figli dei sottogruppi.

Returns

Il numero degli "handle" copiati nella `list`.

list

```
void list()
```

Produce sullo stream `cout` l'insieme dei thread e dei sotto gruppi.

list

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

Produce sullo stream `os` l'insieme dei thread e sotto gruppi. Usa `indent` spazi per la separazione.

5.5.3 Funzioni correlate.

operator<<

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

Stampa una stringa rappresentante il gruppo di thread sull'output stream `os`. Questo chiama `g.list(os, 4)`.

Parametri

`os` - L'output stream nel quale inserire il thread id.

`g` - Riferimento al gruppo di thread.

Returns

L'output stream `os`.

operator!

```
bool operator!() const
```

Determina se l'oggetto riferito dai "handle" non è valido, per esempio, se esso è nullo.

Returns

`true` se l'oggetto non è valido, `false` altrimenti.

operator bool

```
operator bool () const
```

Determina se l'oggetto riferito dall'"handle" è valido, per esempio, se esso non è nullo.

Returns

`true` se l'oggetto è valido, `false` altrimenti.

operator->

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

Invoca un metodo sull'oggetto riferito.

Returns

Un puntatore all'oggetto riferito.

get

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

Restituisce un puntatore all'oggetto riferito.

Returns

Un puntatore all'oggetto riferito.

*operator**

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

Ricerca un riferimento C++ all'oggetto riferito.

Returns

Un riferimento C++ all'oggetto.

5.7 JTCMonitor

Questa classe fornisce la funzionalità dei monitor Java. Per implementare il metodo sincronizzato, il lucchetto del monitor deve essere acquisito, per esempio creado una istanza della classe `JTCSynchronized` all'inizio del metodo sincronizzato, con il monitor come argomento per il costruttore.

Il metodo wait del monitor può essere usato per rilasciare il lucchetto del monitor e di attendere che si venga svegliati. I metodi `notify` e `notifyAll` possono essere usati per svegliare rispettivamente uno o tutti quelli che aspettano sul monitor.

5.7.1 Funzioni

wait

```
void wait()
```

Attende fino al risveglio fatto da un altro thread. Il thread chiamante deve possedere il lucchetto del monitor. Il lucchetto del monitor è rilasciato e il thread attende che venga svegliato da un altro thread tramite `notify` o `notifyAll`. Il thread allora attende finchè esso può riacquisire padronanza del lucchetto del monitor e allora riprende l'esecuzione.

Throws

`JTCIllegalMonitorStateException` - Se il monitor non è bloccato dalla chiamata di un thread.

`JTCSystemCallException` - Indica il fallimento di una chiamata di sistema.

wait

```
wait(long timeout)
```

Attende che venga svegliato da un altro thread. Il thread chiamante deve possedere il lucchetto del monitor. Il lucchetto del monitor è rilasciato e il thread attende che venga svegliato da un altro thread tramite `notify` o `notifyAll`, o finchè non passano timeout millisecondi. Il thread allora attende finchè esso può riacquisire padronanza del lucchetto del monitor e allora riprende l'esecuzione.

Parameters

`timeout` - Il massimo numero di millisecondi per attendere che si venga svegliati.

Throws

`JTCIllegalMonitorStateException` - Se il monitor non è bloccato dalla chiamata di un thread.

`JTCSystemCallException` - Indica il fallimento di una chiamata di sistema.

notify

```
void notify()
```

Sveglia un singolo thread che attende sul monitor. Il thread chiamante deve possedere il lucchetto del monitor.

notifyAll()

```
void notifyAll()
```

Sveglia tutti i thread che attendono sul monitor. Il thread chiamante deve possedere il lucchetto del monitor.

5.8 JTCMonitorT

Questa è una classe template che permette la creazione di classi sincronizzate senza alterare la implementazione.

5.8.1 Costruttori

JTCMonitorT

```
template <class T> JTCMonitorT(T& obj)
```

Inizializza il monitor della classe template. Il parametro `obj` è l'oggetto data che deve essere sincronizzato con questo monitor.

Parametri

`obj` - L'oggetto da sincronizzare.

5.8.2 Funzioni

get

```
template <class T> T& get()
```

Ritorna l'oggetto sincronizzato.

Returns

L'oggetto sincronizzato.

*operator**

```
template <class T> T& operator*()
```

Ritorna l'oggetto sincronizzato.

Returns

L'oggetto sincronizzato.

operator->

```
template <class T> T* operator->()
```

Accesso di un membro dell'oggetto sincronizzato.

Returns

Un puntatore all'oggetto sincronizzato.

operator->

```
template <class T> const T* operator->() const
```

Accesso di un membro costante dell'oggetto sincronizzato.

Returns

Un puntatore `const` all'oggetto sincronizzato.

5.9 JTCRecursiveMutex

Questa classe può essere usata per fissare una sezione critica. Questa classe non ha un diretto equivalente in Java, ed è fornita solo per ragioni di performance. Una istanza di `JTCRecursiveMutex` può essere bloccata più volte dallo stesso thread, e perciò potrebbe non essere tanto efficiente quanto la classe `JTCMutex`.
Lo sviluppatore si deve assicurare che ogni mutex lock ha un corrispondente unlock.

5.9.1 Funzioni

lock

```
void lock() const
```

Blocca la mutex. Se la mutex è gia bloccata, il thread chiamante è bloccato finchè la mutex non viene rilasciata. Se il proprietario corrente della mutex tenta di ri-prendersi la mutex, non ci sarà deadlock.

unlock

```
void unlock() const
```

Questo metodo è chiamato da chi detiene la mutex per rilasciarla. La mutex deve essere bloccata e il thread chiamante deve essere l'ultimo che ha bloccato la mutex. Se queste condizioni non sono soddisfatte ci sarà un comportamento indefinito.

5.10 JTCMutex

Questa classe può essere usata per fissare una sezione critica. Questa classe non ha un diretto equivalente in Java, ed è fornita solo per ragioni di performance. A differenza di `JTCMonitor` o `JTCRecursiveMutex`, questa classe non garantisce una semantica ricorsiva di blocco. Se la mutex è bloccata più di una volta dallo stesso thread, ci potrebbe essere deadlock.

5.10.1 Funzioni

lock

```
void lock() const
```

Blocca la mutex. Se la mutex è gia bloccata, il thread chiamante è bloccato finchè la mutex non viene rilasciata. Se il proprietario corrente della mutex tenta di ri-prendersi la mutex, ci potrebbe essere deadlock.

unlock

```
void unlock() const
```

Questo metodo è chiamato da chi detiene la mutex per rilasciarla. La mutex deve essere bloccata e il thread chiamante deve essere l'ultimo che ha bloccato la mutex. Se queste condizioni non sono soddisfatte ci sarà un comportamento indefinito.

5.11 JTCSynchronized

Questa classe è usata per acquisire e rilasciare il lucchetto di un monitor. Per creare un metodo sincronizzato, un istanza di questa classe potrebbe essere creato con il monitor passato come argomento al costruttore. Il costruttore acquisisce il lucchetto e il distruttore rilascia il lucchetto. Questa classe potrebbe anche essere usata con le classi `JTCMutex` e `JTCRecursiveMutex`.

5.11.1 Constructor

JTCSynchronized

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

Acquisisce il lucchetto del monitor. Il distruttore rilascia il lucchetto del monitor.

Throws

`JTCSystemCallException` - Indica il fallimento di una chiamata di sistema.

JTCSynchronized

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

Acquisisce il lucchetto del monitor. Il distruttore rilascia il lucchetto del monitor.

Throws

`JTCSystemCallException` - Indica il fallimento di una chiamata di sistema.

JTCSynchronized

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

Acquisisce il lucchetto della mutex. Il distruttore rilascia il lucchetto della mutex.

Throws

`JTCSystemCallException` - Indica il fallimento di una chiamata di sistema.

5.12 JTCThreadId

Questa classe rappresenta l'id del thread. La sola operazione che può essere usata sono l'uguaglianza e la diversità. Due oggetti thread possono essere considerati uguali se i loro id sono uguali. Un utente può non costruire direttamente una istanza di questa classe.

5.12.1 Funzioni

operator==

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

Confronto per uguaglianza.

Parametri

`rhs` - L'id del thread con cui fare il confronto

Returns

`true` se gli id dei thread sono equivalenti, `false` altrimenti.

operator!=

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

Confronto per diseguaglianza.

Parametri

`rhs` - L'id del thread con cui fare il confronto.

Returns

`true` se gli id dei thread non sono equivalenti, `false` altrimenti.

5.13 JTCTSS

Questa classe è usata per gestire la memorizzazione di un thread specifico, la quale è uno strumento estremamente utile per la gestione dei dati che sono associati ad ogni thread. Ogni thread associa i dati con una chiave. Poichè ogni thread ha i suoi propri dati, non c'è conflitto fra i dati di thread multipli.

5.13.1 Funzioni

allocate

```
static JTCThreadKey allocate()
```

Crea un nuovo thread specifico per la memorizzazione delle chiavi.

Returns

Un nuovo thread specifico per la memorizzazione delle chiavi.

Throws

`JTCSystemCallException` - Indica il fallimento di una chiamata di sistema.

allocate

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

Crea un nuovo thread specifico per la memorizzazione delle chiavi con una associata funzione di cancellazione. Subito dopo la terminazione del thread, è chiamata la funzione di cancellazione registrata con un argomanto che contiene il valore associato con il thread thread specifico per la memorizzazione delle chiavi.

Returns

Un nuovo thread specifico per la memorizzazione delle chiavi.

Throws

`JTCSystemCallException` - Indica il fallimento di una chiamata di sistema.

release

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

Rilascia un thread specifico per la memorizzazione delle chiavi. Lo sviluppatore è responsabile di liberare qualsiasi memorizzazione associata prima di rilasciare la chiave. Qualsiasi funzione di cancellazione associata non viene chiamata.

Parametri

`key` - Il thread specifico per la memorizzazione delle chiavi da rilasciare.

Throws

`JTCSystemCallException` - Indica il fallimento di una chiamata di sistema.

get

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

Restituisce i dati associati con il thread specifico per la memorizzazione delle chiavi.

Parametri

`key` - Il thread specifico per la memorizzazione delle chiavi.

Returns

I dati associati con il thread specifico per la memorizzazione delle chiavi.

Throws

`JTCSystemCallException` - Indica il fallimento di una chiamata di sistema.

set

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

Associa i dati con un thread specifico per la memorizzazione delle chiavi.

Parametri

`key` - Il thread specifico per la memorizzazione delle chiavi.

`data` - I dati associati con la chiave.

Throws

`JTCSystemCallException` - Indica il fallimento di una chiamata di sistema.

5.14 JTCThreadDeath

Questa eccezione è lanciata quando un thread viene terminato da `JTCThread::stop`.
Se l'eccezione è catturata, essa deve essere ri-lanciata per la corretta terminazione del thread.

5.15 JTCException

Con l'eccezione `JTCThreadDeath`, `JTCException` è la classe base di tutte le classi eccezione JTHREADS/C++.

5.15.1 Costruttori

JTCException

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

Costruisce `JTCException` con il messaggio `note`, e errore con tipo specificato in `error`.

Parametri

`note` - Una descrizione dell'errore.

`error` - Un codice di errore di una eccezione specifica.

5.15.2 Funzioni

getError

```
long getError() const
```

Ritorna il codice di errore dell'eccezione specifica. Al momento solo `JTCSystemCallException` ha un codice di errore specifico.

Returns

Il codice di errore.

getType

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

Ritorna una stringa rappresentante il tipo di eccezione. Questo è il nome della classe eccezione. Questa funzione non è disponibile in Java.

Returns

Il nome della classe.

getMessage

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

Ritorna una descrizione dell'eccezione. Questa è il parametro `note` fornito nel costruttore.

Returns

Una descrizione dell'eccezione.

5.15.3 Funzioni correlate

operator<<

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

Inserisce una descrizione dell'errore sull'output stream `os`.

Parametri

`os` - L'output stream nel quale viene inserito l'id del thread.

`e` - Il riferimento all'eccezione.

Returns

L'output stream `os`.

5.16 JTCInterruptedException

Questa eccezione è lanciata se una chiamata di sistema viene interrotta. Al momento `JTCMonitor::wait()` e `JTCThread::sleep()` possono lanciare questa eccezione. La semantica differisce da Java. Una `InterruptedException` in Java è lanciata se un thread è interrotto da `java.lang.Thread.interrupt`. Sfortunatamente, è impossibile implementare questo metodo in un modo portabile usando i modelli POSIX e WIN32.

5.17 JTCIllegalThreadStateException

Questa eccezione è lanciata se una funzione è chiamata mentre l'oggetto è in uno stato illegal. Al momento `JTCThread::start(), i costruttori JTCThreadGroup::JTCThreadGroup() e JTCThreadGroup::destroy()` possono lanciare questa eccezione.

5.18 JTCIllegalMonitorStateException

Questa eccezione è lanciata da `JTCMonitor::wait()`, `JTCMonitor::notify()` o `JTCMonitor::notifyAll()` se il lucchetto del monitor non è stato acquisito dal thread chiamante.

5.19 JTCIllegalArgumentException

Questa eccezione è lanciata quando un argomento illegale è passato ad un metodo JTHREADS/C++. I metodi `JTCMonitor::wait()` (con un argomento con timeout), `JTCThread::setPriority()`, e `JTCThread::sleep()`possono lanciare questa eccezione.

5.20 JTCSystemCallException

Questa eccezione indica il fallimento di una chiamata di sistema. Più metodi JTHREADS/C++ possono generare questa eccezione. Il metodo `JTCException::getError()`ritorna il valore di errore. Sotto UNIX questo è il valore di `errno`, sotto WIN32 questo è il valore di `getLastError()`. Non c'è un metodo per determinare quale operazione ha causato l'errore. Comunque, il messaggio di eccezione contiene una descrizione dell'operazione, e tutti gli argomenti di assistenza per il debuggin.

5.21 JTCOutOfMemoryError

Questa eccezione è generata dai costruttori `JTCThread` su una condizione di out of memory.

5.22 JTCInitializeError

Questa eccezione è generata dal costruttore `JTCInitialize(int&, char**)` quando una opzione invalida o un argomento invalido è specificato.

Riferimenti

[1] *The JTHREADS/C++ Home Page*, `http://www.ooc.com/jtc/`, Object Oriented Concepts, Inc.

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

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

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

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

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