Comparison of POSIX and Java Threads Models

By Chary Tamirisa

AIX 4 supports multithreaded application development with a C language-based user level threads library called the Pthreads library, based on the POSIX.1c draft 7 standard. Typically, concurrent applications are written in C and C++ and use the Pthreads library. With the porting of the Java Development Kit (JDK) 1.0.2 to AIX 4, we have another threading model in the Java environment. This article compares the POSIX and the Java threads models. It is intended to explain the Java threads model to those familiar with the POSIX threads model. This article assumes the reader is familiar with the Pthreads library and the Java environment.

AIX 4 provides a threads library based on POSIX.1c draft 7 and supported by kernel threads. In fact, there is a one-to-one mapping of user-level threads to kernel threads. C and C++ programs typically use this threads package. POSIX threads (pthreads for short) is biased toward the C language and is not object-oriented.

Java threads and pthreads have several common features. For example, the two models have a similar synchronization mechanism. POSIX threads offers features such as thread-specific data and signal (UNIX® signals) support that are not found in Java threads. We will examine the two models briefly with focus on a few important pthreads APIs.

Thread Management

The following sections discuss various aspects of thread management comparing POSIX and Java.

Process versus threads. Before the advent of threads, the concept of a process referred to resources used by a program (such as the address space, file descriptors) as well as to the entity in an operating system that can be executed or scheduled. After threads were introduced, the term process now defines the resources and the thread defines a sequence of execution. Both POSIX and Java support the concepts of process and threads within process, with the separation maintained.

Thread creation. POSIX threads define a `pthread_create()` API for creating a thread. Figure 1 shows the syntax.

The `pthread_create()` Application Programming Interface (API) takes four arguments:

- Thread handle for the newly created thread is an output argument
- Optional attribute for the new thread
- Actual thread body to be executed in the newly created thread
- Optional parameter passed to the new thread

The thread handle and thread function are the two important arguments in the thread creation API. When `pthread_create()` returns, the new thread is created and ready to run; it is not necessary to start it explicitly. Unlike Java, all threads are created ready to run.

In Java, you implement the `java.lang.Runnable` interface, which involves implementing the thread body in the `run()` method. You then pass an instance of this class to the Thread constructor (the `java.lang.Thread` class). Finally, you start the thread running by invoking the `start()` method on the Thread object.

Chary Tamirisa

```
#include <pthread.h>
int pthread_create(pthread_t *thread, const pthread_attr_t *attr,
                   void *(*start_routine)(void *), void *arg);
```

Figure 1. POSIX thread creation API

Thread Termination. In pthreads, a thread is terminated by simply returning from the thread body or by invoking `pthread_exit()` explicitly. In Java, a thread is terminated by simply returning from the `run()` method.

In pthreads, if the main thread exits by simply returning from the main thread, the whole process is terminated. To terminate only the main thread and not the whole process, you need to invoke `pthread_exit()` explicitly if the main thread needs to be terminated. In a Java application (with `main()`) if the main thread returns, the process does not exit if non-daemon threads are present. The Java Virtual Machine terminates a process if any thread invokes the `exit()` method of the Runtime or System classes—if the security manager allows the exit operation.

Thread Attributes. In pthreads, every thread has a set of attributes that can specify a thread's stack size, the stack address, and the detach state. Java does not have corresponding APIs; however, the Thread class allows you to specify if a thread is a daemon thread. You also can associate a name with a thread for easy identification. This is often useful when you are debugging a program.

Thread IDs. Every pthread has an opaque handle associated with it; this handle is returned when a pthread is created. Similarly, every Java thread has a Thread object associated with it, which is used to manipulate the thread.

Waiting for Thread Termination. Pthreads provides the detach state to control what happens when a thread is terminated. You can set the detach state of a thread to indicate whether you are interested in waiting for the exit status after the thread terminates. If you are not interested in the exit status of a thread, you can set the thread's detach state to indicate that any system resources associated with it should be freed up after it terminates.

The `pthread_join()` API allows you to wait for a thread to obtain its exit status. Java does not have a detach state associated with a thread. However, similar to pthreads, you can join with a thread using the `Thread` class' `join()method`. Java does not return any exit status associated with a thread when `join()` returns. In pthreads, the `pthread_join()` API does not have a timeout associated with it; it just blocks until the specified thread terminates. Java, however, allows a timeout to be associated with the `join()` method.

Detaching a Thread. In pthreads, you can specify the detach state of a thread by invoking `pthread_detach()` on a running thread. If the detach state of a thread is set, the pthreads library releases all the library resources associated with a thread. Java does not have a similar API.

Dynamic Package Initialization. Pthreads allows you to invoke an initializer only once using the `pthread_once()` API. This is useful in libraries where the locks must be initialized before they are used, and initialized only once. Since Java objects are always associated with a lock, an application does not need to initialize the locks associated with an object. Java has no such mechanism.

Comparison of Thread IDs. You can compare two pthreads using the `pthread_equal()` API. In Java, you can compare two Thread objects using the `equals()` method of `java.lang.Object` class.

Thread Suspend/Resume. Pthreads does not provide any explicit APIs for suspending or resuming a thread. Java allows a thread to be suspended or resumed.

Thread Synchronization

Pthreads provides the pthread mutex APIs to ensure mutual exclusion of threads to protect critical sections. The Java language supports the keyword synchronized (Expression) to provide locking for critical sections. Pthreads and Java provide similar functionality.

In addition to mutexes, pthreads allows a thread to wait for an event through the condition variable APIs. The condition variable is generally associated with a mutex and a boolean variable. Pthreads provides two APIs for waiting: the non-timed wait API `pthread_cond_wait()` and the timed wait API `pthread_cond_timedwait()`.

The following discussion concerns `pthread_cond_wait()`, but it also applies to the timed wait API. A thread invokes the condition variable wait API with the mutex locked.

In a loop (typically `while()` loop), determine if the flag associated with the event of interest indicates whether the current thread should wait. When the thread invokes `pthread_cond_wait()`, it atomically releases the mutex lock and blocks.

Another thread changes the flag and signals this thread when the event occurs. This thread is awakened, obtains the mutex lock again, and returns from `pthread_cond_wait()`. The while loop forces the thread to evaluate the flag again, and this protects against spurious wake ups. Once the loop is successfully completed, the thread assumes that the condition for which it has been waiting is satisfied and it can proceed. The mutex lock protects the shared boolean flag.

Finally, the mutex is unlocked and the thread continues. Figure 2 shows how condition variables are used in pthreads.

The mechanism of condition variable waits is similar to the `wait()/notify()` family of methods provided in the Java class `java.lang.Object`. In fact, the usage of these methods involves similar steps as outlined above. The pthread's condition variable APIs expect that a non-recursive mutex as the mutex lock is unlocked only once before blocking in `pthread_cond_wait()/pthread_cond_timedwait()`. If a recursive mutex is used and it is locked more than once by the thread, it cannot be used in the condition variable APIs.

Java is different. The same thread can invoke the `synchronized()` method on the same object several times and still block on the object by invoking the `wait()` method on the object. The `wait()` method will unlock the object as many times as needed and make the object lockable by other threads.

Thread-Specific Data

Pthreads supports thread-specific data through a set of APIs. You can associate a unique key with a memory you allocate on a per-thread basis. First, create a thread-specific key. The pthreads library ensures that all the threads in the process have a null pointer associated with this key. Each thread can then allocate memory and associate that memory with this key. Once this is done, each thread can access this thread-specific memory by simply using the key associated with the memory. The pthreads library sets a limit on how many keys you can create. When you create a key, you can also specify that a destroy function be invoked to free up the thread-specific memory when the thread terminates. Java does not offer support for thread-specific data.

Execution Scheduling

Pthreads offers two fixed real-time priority-based scheduling policies called First-In-First-Out (FIFO) policy and Round Robin (RR) policy. In

```
/*Following is a code fragment to illustrate the pthread condition variable API */

#include <pthread.h>

pthread_mutex_t mutex;
pthread_cond_t condition_variable;
int flag;

pthread_mutex_lock(&mutex); /* lock the critical section */

while( !flag) {

     /* boolean variable indicating if the event has occurred */
     pthread_cond_wait(&condition_variable, &mutex);
     /* Wait for another thread to signal this thread on a change in state */

}

pthread_mutex_unlock(&mutex); /* release the lock */
```

Figure 2. POSIX condition variable API usage

FIFO policy, a thread with the highest priority runs until it blocks; there is no time-slicing. In the RR policy, a thread with highest priority runs; however, threads with equal priority are time-sliced. In addition to these polices, pthreads allows an implementation to provide any scheduling policy under the `SCHED_OTHER` policy.

Pthreads has priority inversion where a high-priority thread may be blocked for a lock held by a low-priority thread. This low-priority thread may be preempted from running by a higher priority thread that may run indefinitely. In Java, the scheduling policy is preemptive; that is, a thread with a low priority will be preempted from running by a higher priority ready to run a thread. However, Java does not specify any time-slicing of threads of equal priority. To share the execution among threads of the same priority, you can use the `Thread` class' `yield()` method.

Synchronization Scheduling

To prevent priority inversion, pthreads specifies APIs to contain the priority inversion. Java does not have any such mechanisms.

Thread Cancellation

Pthreads allows a thread to be terminated asynchronously in addition to the `pthread_exit()` API. This cancel mechanism is useful when a thread needs to be terminated from another thread. Pthreads provides a way to control whether a thread can be canceled. In addition, pthreads allows an application to clean up state when a thread is canceled.

Java also allows a thread to be canceled from another thread using the `stop()` method of the `Thread` class. When `stop()` is invoked on a thread, the target thread is forced to complete whatever it is doing abnormally. In addition, the target thread throws an exception. Java Version 1.1 specifies a new method `interrupt()` to post an interruption to a thread. The target thread does not necessarily react immediately to the interrupt. The target thread throws the exception: `InterruptedException`.

Summary

We have presented a brief comparative study of the POSIX and Java threads models describing their salient features. Although Java does not offer all of the POSIX features in the JDK 1.0.2, the Java threading model is powerful.

References

Gosling, James; Joy, Bill; Steele, Guy. *The Java Language Specification*. Addison-Wesley, 1996.

Flanagan, David. *Java in a Nutshell.* O'Reilly & Associates, Inc., 1996.

Lea, Doug. *Concurrent Programming in Java: Design Principles and Patterns.* Addison-Wesley, 1996.

Chary Tamirisa, *IBM Corporation, 11400 Burnet Road, Austin, TX 78758. Internet: chary@austin.ibm.com. Mr. Tamirisa was the team leader for the threads package on AIX and OS/2® DCE. He has also worked in the fields of communication protocols, system software, and National Language Support. Mr. Tamirisa has an MS in Computer Science from McGill University and a BTech in Electrical Engineering from the Indian Institute of Technology in Madras, India.*