C H A P T E R 21

Multi-Threaded Tcl Scripts

This chapter describes the `Thread` extension for creating multi-threaded Tcl scripts.

Thread support, a key feature of many languages, is a recent addition to Tcl. That's because the Tcl event loop supports features implemented by threads in most other languages, such as graphical user interface management, multi-client servers, asynchronous communication, and scheduling and timing operations. However, although Tcl's event loop can replace the need for threads in many circumstances, there are still some instances where threads can be a better solution:

- Long-running calculations or other processing, which can "starve" the event loop
- Interaction with external libraries or processes that don't support asynchronous communication
- Parallel processing that doesn't adapt well to an event-driven model
- Embedding Tcl into an existing multi-threaded application

What are Threads?

Traditionally, processes have been limited in that they can do only one thing at a time. If your application needed to perform multiple tasks in parallel, you designed the application to create multiple processes. However, this approach has its drawbacks. One is that processes are relatively "heavy" in terms of the resources they consume and the time it takes to create them. For applications that frequently create new processes — for example, servers that create a new process to handle each client connection — this can lead to decreased response time. And widely parallel applications that create many processes can consume so many system resources as to slow down the entire system. Another drawback is that passing information between processes can be slow because most interprocess communication mechanisms — such as files, pipes, and sockets — involve intermediaries such as the file system or operating system, as well as requiring a context switch from one running process to another.

Threads were designed as a light-weight alternative. Threads are multiple flows of execution within the same process. All threads within a process share the same memory and other resources. As a result, creating a thread requires far fewer resources than creating a separate process. Furthermore, sharing information between threads is much faster and easier than sharing information between processes.

The operating system handles the details of thread creation and coordination. On a single-processor system, the operating system allocates processor time to each of an application's threads, so a single thread doesn't block the rest of the application. On multi-processor systems, the operating system can even run threads on separate processors, so that threads truly can run simultaneously.

The drawback to traditional multi-threaded programming is that it can be difficult to design a *thread-safe* application — that is, an application in which one thread doesn't corrupt the resources being used by another thread. Because all resources are shared in a multi-threaded application, you need to use various locking and scheduling mechanisms to guard against multiple threads modifying resources concurrently.

Thread Support in Tcl

Tcl added support for multi-threaded programming in version 8.1. The Tcl core was made thread-safe. Furthermore, new C functions exposed "platform-neutral" thread functionality. However, no official support was provided for multithreaded scripting. Since then, the `Thread` extension — originally written by Brent Welch and currently maintained by Zoran Vasiljevic — has become the accepted mechanism for creating multi-threaded Tcl scripts. The most recent version of the `Thread` extension as this was being written was 2.5. In general, this version requires Tcl 8.3 or later, and several of the commands provided require Tcl 8.4 or later.

At the C programming level, Tcl's threading model requires that a Tcl interpreter be managed by only one thread. However, each thread can create as many Tcl interpreters as needed running under its control. As is the case in even a single-threaded application, each Tcl interpreter has its own set of variables and procedures. A thread can execute commands in another thread's Tcl interpreter only by sending special messages to that interpreter's event queue. Those messages are handled in the order received along with all other types of events.

Obtaining a Thread-Enabled Tcl Interpreter

Most binary distributions of Tcl are not thread-enabled, because the default options for building the Tcl interpreters and libraries do not enable thread support. Thread safety adds overhead, slowing down single-threaded Tcl applications, which constitute the vast majority of Tcl applications. Also, many Tcl extensions aren't thread safe, and naively trying to use them in a multi-threaded application can cause errors or crashes.

Unless you can obtain a thread-enabled binary distribution of Tcl, you must compile your own from the Tcl source distribution. This requires running the `configure` command with the `--enable-threads` option during the build process. (See Chapter 48, "Compiling Tcl and Extensions" for more information.)

You can test whether a particular Tcl interpreter is thread-enabled by checking for the existence of the `tcl_platform(threaded)` element. This element exists and contains a Boolean true value in thread-enabled interpreters, whereas it doesn't exist in interpreters without thread support.

Using Extensions in Multi-Threaded Scripts

Because each interpreter has its own set of variables and procedures, you must explicitly load an extension into each thread that wants to use it. Only the `Thread` extension itself is automatically loaded into each interpreter.

You must be careful when using extensions in multi-threaded scripts. Many Tcl extensions aren't thread-safe. Attempting to use them in multi-threaded scripts often results in crashes or corrupted data.

Tcl-only extensions are generally thread-safe. Of course, they must make no use of other commands or extensions that aren't thread-safe. But otherwise, multi-threaded operation doesn't add any new issues that don't already affect single-threaded scripts.

You should always assume that a binary extension is not thread-safe unless its documentation explicitly says that it is. And even thread-safe binary extensions must be compiled with thread support enabled for you to use them in multi-threaded applications. (The default compilation options for most binary extensions don't include thread support.)

Tk isn't truly thread-safe.

Most underlying display libraries (such as X Windows) aren't thread safe — or at least aren't typically compiled with thread-safety enabled. However, significant work has gone into making the Tk core thread-safe. The result is that you can safely use Tk in a multi-threaded Tcl application as long as only one thread uses Tk commands to manage the interface. Any other thread that needs to update the interface should send messages to the thread controlling the interface.

Getting Started with the Thread Extension

You start a thread-enabled `tclsh` or `wish` the same as you would a non-threaded `tclsh` or `wish`. When started, there is only one thread executing, often referred to as the *main thread*, which contains a single Tcl interpreter. If you don't create any more threads, your application runs like any other single-threaded application.

Make sure that the main thread is the last one to terminate.

The main thread has a unique position in a multi-threaded Tcl script. If it exits, then the entire application terminates. Also, if the main thread terminates while other threads still exist, Tcl can sometimes crash rather than exiting cleanly. Therefore, you should always design your multi-threaded applications so that your main thread waits for all other threads to terminate before it exits.

Before accessing any threading features from your application, you must load the `Thread` extension:

```
package require Thread
```

The `Thread` extension automatically loads itself into any new threads your application creates with `thread::create`. All other extensions must be loaded explicitly into each thread that needs to use them. The `Thread` extension creates commands in three separate namespaces:

- The `thread` namespace contains all of the commands for creating and managing threads, including inter-thread messaging, mutexes, and condition variables.
- The `tsv` namespace contains all of the commands for creating and managing thread shared variables.
- The `tpool` namespace contains all of the commands for creating and managing thread pools.

Creating Threads

The `thread::create` command creates a new thread containing a new Tcl interpreter. Any thread can create another thread at will; you aren't limited to starting threads from only the main thread. The `thread::create` command returns immediately, and its return value is the ID of the thread created. The ID is a unique token that you use to interact with and manipulate the thread, in much the same way as you use a channel identifier returned by `open` to interact with and manipulate that channel. There are several commands available for introspection on thread IDs: `thread::id` returns the ID of the current thread; `thread::names` returns a list of threads currently in existence; and `thread::exists` tests for the existence of a given thread.

The `thread::create` command accepts a Tcl script as an argument. If you provide a script, the interpreter in the newly created thread executes it and then terminates the thread. Example 21–1 demonstrates this by creating a thread to perform a recursive search for files in a directory. For a large directory structure, this could take considerable time. By performing the search in a separate thread, the main thread is free to perform other operations in parallel. Also note how the "worker" thread loads an extension and opens a file, completely independent of any extensions loaded or files opened in other threads.

Example 21–1 Creating a separate thread to perform a lengthy operation.

```
package require Thread

# Create a separate thread to search the current directory
# and all its subdirectories, recursively, for all files
# ending in the extension ".tcl". Store the results in the
# file "files.txt".

thread::create {
    # Load the Tcllib fileutil package to use its
    # findByPattern procedure.

    package require fileutil

    set files [fileutil::findByPattern [pwd] *.tcl]

    set fid [open files.txt w]
    puts $fid [join $files \n]
    close $fid
}

# The main thread can perform other tasks in parallel...
```

If you don't provide a script argument to `thread::create`, the thread's interpreter enters its event loop. You then can use the `thread::send` command, described on page 328, to send it scripts to evaluate. Often though, you'd like to perform some initialization of the thread before having it enter its event loop. To do so, use the `thread::wait` command to explicitly enter the event loop after performing any desired initialization, as shown in Example 21–2. You should always use `thread::wait` to cause a thread to enter its event loop, rather than `vwait` or `tkwait`, for reasons discussed in "Preserving and Releasing Threads" on page 330.

Example 21–2 Initializing a thread before entering its event loop.

```
set httpThread [thread::create {
    package require http
    thread::wait
}]
```

After creating a thread, never assume that it has started executing.

There is a distinction between creating a thread and starting execution of a thread. When you create a thread, the operating system allocates resources for the thread and prepares it to run. But after creation, the thread might not start execution immediately. It all depends on when the operating system allocates execution time to the thread. Be aware that the `thread::create` command returns when the thread is *created*, not necessarily when it has *started*. If your application has any inter-thread timing dependencies, always use one of the thread synchronization techniques discussed in this chapter.

Creating Joinable Threads

Remember that the main thread must be the last to terminate. Therefore you often need some mechanism for determining when it's safe for the main thread to exit. Example 21–3 shows one possible approach: periodically checking `thread::names` to see if the main thread is the only remaining thread.

Example 21–3 Creating several threads in an application.

```
package require Thread

puts "*** I'm thread [thread::id]"

# Create 3 threads

for {set thread 1} {$thread <= 3} {incr thread} {
   set id [thread::create {

      # Print a hello message 3 times, waiting
      # a random amount of time between messages

      for {set i 1} {$i <= 3} {incr i} {
         after [expr { int(500*rand()) }]
         puts "Thread [thread::id] says hello"
      }

   }] ;# thread::create

   puts "*** Started thread $id"
} ;# for

puts "*** Existing threads: [thread::names]"

# Wait until all other threads are finished

while {[llength [thread::names]] > 1} {
   after 500
}

puts "*** That's all, folks!"
```

A better approach in this situation is to use *joinable* threads, which are supported in Tcl 8.4 or later. A joinable thread allows another thread to wait upon its termination with the `thread::join` command. You can use thread::join only with joinable threads, which are created by including the thread::create -joinable option. Attempting to join a thread not created with -joinable results in an error. Failing to join a joinable thread causes memory and other resource leaks in your application. Example 21–4 revises the program from Example 21–3 to use joinable threads.

Example 21–4 Using joinable threads to detect thread termination.

```
package require Thread

puts "*** I'm thread [thread::id]"

# Create 3 threads

for {set thread 1} {$thread <= 3} {incr thread} {
    set id [thread::create -joinable {

        # Print a hello message 3 times, waiting
        # a random amount of time between messages

        for {set i 1} {$i <= 3} {incr i} {
            after [expr { int(500*rand()) }]
            puts "Thread [thread::id] says hello"
        }

    }] ;# thread::create

    puts "*** Started thread $id"

    lappend threadIds $id

} ;# for

puts "*** Existing threads: [thread::names]"

# Wait until all other threads are finished

foreach id $threadIds {
    thread::join $id
}

puts "*** That's all, folks!"
```

The thread::join *command blocks.*

Be aware that thread::join blocks. While the thread is waiting for thread::join to return, it can't perform any other operations, including servicing its event loop. Therefore, make sure that you don't use thread::join in situations where a thread must be responsive to incoming events.

Sending Messages to Threads

The `thread::send` command sends a script to another thread to execute. The target thread's main interpreter receives the script as a special type of event added to the end of its event queue. A thread evaluates its messages in the order received along with all other types of events. Obviously, a thread must be in its event loop for it to detect and respond to messages. As discussed on page 324, a thread enters its event loop if you don't provide a script argument to `thread::create`, or if you include the `thread::wait` command in the thread's initialization script.

Synchronous Message Sending

By default, `thread::send` blocks until the target thread finishes executing the script. The return value of `thread::send` is the return value of the last command executed in the script. If an error occurs while evaluating the script, the error condition is "reflected" into the sending thread; `thread::send` generates the same error code, and the target thread's stack trace is included in the value of the `errorInfo` variable of the sending thread:

Example 21–5 Examples of synchronous message sending.

```
set t [thread::create]  ;# Create a thread
=> 1572
set myX 42  ;# Create a variable in the main thread
=> 42
# Copy the value to a variable in the worker thread
thread::send $t [list set yourX $myX]
=> 42
# Perform a calculation in the worker thread
thread::send $t {expr { $yourX / 2 } }
=> 21
thread::send $t {expr { $yourX / 0 } }
=> divide by zero
catch {thread::send $t {expr { $yourX / 0 } } } ret
=> 1
puts $ret
=> divide by zero
puts $errorInfo
=> divide by zero
       while executing
   "expr { $yourX / 0 } "
       invoked from within
   "thread::send $t {expr { $yourX / 0 } } "
```

If you also provide the name of a variable to a synchronous `thread::send`, then it behaves analogously to a `catch` command; `thread::send` returns the return code of the script, and the return value of the last command executed in the script — or the error message — is stored in the variable. Tcl stores the target thread's stack trace in the sending thread's `errorInfo` variable.

Example 21–6 Using a return variable with synchronous message sending.

```
thread::send $t {incr yourX 2} myY
=> 0
puts $myY
=> 44
thread::send $t {expr { acos($yourX) } } ret
=> 1
puts $ret
=> domain error: argument not in valid range
puts $errorInfo
=> domain error: argument not in valid range
        while executing
    "expr { acos($yourX) } "
```

While the sending thread is waiting for a synchronous `thread::send` to return, it can't perform any other operations, including servicing its event loop. Therefore, synchronous sending is appropriate only in cases where:

- you want a simple way of getting a value back from another thread;
- you don't mind blocking your thread if the other thread takes a while to respond; or
- you need a response from the other thread before proceeding.

Watch out for deadlock conditions with synchronous message sending.

If Thread A performs a synchronous `thread::send` to Thread B, and while evaluating the script Thread B performs a synchronous `thread::send` to Thread A, then your application is deadlocked. Because Thread A is blocked in its `thread::send`, it is not servicing its event loop, and so can't detect Thread B's message.

This situation arises most often when the script you send calls procedures in the target thread, and those procedures contain `thread::send` commands. Under these circumstances, it might not be obvious that the script sent will trigger a deadlock condition. For this reason, you should be cautious about using synchronous `thread::send` commands for complex actions. Sending in asynchronous mode, described in the next section, avoids potential deadlock situations like this.

Asynchronous Message Sending

With the `-async` option, `thread::send` sends the script to the target thread in asynchronous mode. In this case, `thread::send` returns immediately.

By default, an asynchronous `thread::send` discards any return value of the script. However, if you provide the name of a variable as an additional argument to `thread::send`, the return value of the last command executed in the script is stored as the value of the variable. You can then either `vwait` on the variable or create a write trace on the variable to detect when the target thread responds. For example:

```
thread::send -async $t [list ProcessValues $vals] result
vwait result
```

In this example, the `thread::send` command returns immediately; the sending thread could then continue with any other operations it needed to perform. In this case, it executes a `vwait` on the return variable to wait until the target thread finishes executing the script. However, while waiting for the response, it can detect and process incoming events. In contrast, the following synchronous `thread::send` blocks, preventing the sending thread from processing events until it receives a response from the target thread:

```
thread::send $t [list ProcessValues $vals] result
```

Preserving and Releasing Threads

A thread created with a script not containing a `thread::wait` command terminates as soon as the script finishes executing. But if a thread enters its event loop, it continues to run until its event loop terminates. So how do you terminate a thread's event loop?

Each thread maintains an internal reference count. The reference count is set initially to 0, or to 1 if you create the thread with the `thread::create -preserved` option. Any thread can increment the reference count afterwards by executing `thread::preserve`, and decrement the reference count by executing `thread::release`. These commands affect the reference count of the current thread unless you specify the ID of another thread. If a call to `thread::release` results in a reference count of 0 or less, the thread is marked for termination.

The use of thread reference counts allows multiple threads to preserve the existence of a worker thread until all of the threads release the worker thread. But the majority of multi-threaded Tcl applications don't require that degree of thread management. In most cases, you can simply create a thread and then later use `thread::release` to terminate it:

```
set worker [thread::create]
thread::send -async $worker $script
# Later in the program, terminate the worker thread
thread::release $worker
```

A thread marked for termination accepts no further messages and discards any pending events. It finishes processing any message it might be executing currently, then exits its event loop. If the thread entered its event loop through a call to `thread::wait`, any other commands following `thread::wait` are executed before thread termination, as shown in Example 21–7. This can be useful for performing "clean up" tasks before terminating a thread.

Example 21–7 Executing commands after `thread::wait` returns.

```
set t [thread::create {
    puts "Starting worker thread"
    thread::wait
    # This is executed after the thread is released
    puts "Exiting worker thread"
}]
```

Note that if a thread is executing a message script when `thread::release` is called (either by itself or another thread), the thread finishes executing its message script before terminating. So, if a thread is stuck in an endless loop, calling `thread::release` has no effect on the thread. In fact, there is no way to kill such a “runaway thread.”

Always use `thread::wait` *to enter a thread’s event loop.*

This system for preserving and releasing threads works only if you use the `thread::wait` command to enter the thread’s event loop (or if you did not provide a creation script when creating the thread). If you use `vwait` or `tkwait` to enter the event loop, `thread::release` cannot terminate the thread.

Error Handling

If an error occurs while a thread is executing its creation script (provided by `thread::create`), the thread dies. In contrast, if an error occurs while processing a message script (provided by `thread::send`), the default behavior is for the thread to stop execution of the message script, but to return to its event loop and continue running. To cause a thread to die when it encounters an uncaught error, use the `thread::configure` command to set the thread’s `-unwindonerror` option to true:

```
thread::configure $t -unwindonerror 1
```

Error handling is determined by the thread creating the thread or sending the message. If an error occurs in a script sent by a synchronous `thread::send`, then the error condition is “reflected” to the sending thread, as described in “Synchronous Message Sending” on page 328. If an error occurs during thread creation or an asynchronous `thread::send`, the default behavior is for Tcl to send a stack trace to the standard error channel. Alternatively, you can specify the name of your own custom error handling procedure with `thread::errorproc`. Tcl automatically calls your procedure whenever an “asynchronous” error occurs, passing it two arguments: the ID of the thread generating the error, and the stack trace. (This is similar to defining your own `bgerror` procedure, as described in “The `bgerror` Command” on page 202.) For example, the following code logs all uncaught errors to the file `errors.txt`:

Example 21–8 Creating a custom thread error handler.

```
set errorFile [open errors.txt a]

proc logError {id error} {
   global errorFile
   puts $errorFile "Error in thread $id"
   puts $errorFile $error
   puts $errorFile ""
}

thread::errorproc logError
```

Shared Resources

The present working directory is a resource shared by all interpreters in all threads. If one thread changes the present working directory, then that change affects all interpreters and all threads. This can pose a significant problem, as some library routines temporarily change the present working directory during execution, and then restore it before returning. But in a multi-threaded application, another thread could attempt to access the present working directory during this period and get incorrect results. Therefore, the safest approach if your application needs to access the present working directory is to store this value in a global or thread-shared variable before creating any other threads. The following example uses `tsv::set` to store the current directory in the `pwd` element of the `application` shared variable:

```
package require Thread
# Save the pwd in a thread-shared variable
tsv::set application pwd [pwd]
set t [thread::create {#...}]
```

Environment variables are another shared resource. If one thread makes a change to an environment variable, then that change affects all threads in your application. This might make it tempting to use the global `env` array as a method for sharing information between threads. However, you should not do so, because it is far less efficient than thread-shared variables, and there are subtle differences in the way environment variables are handled on different platforms. If you need to share information between threads, you should instead use thread-shared variables, as discussed in “Shared Variables” on page 337.

The `exit` *command kills the entire application.*

Although technically not a shared resource, it’s important to recognize that the `exit` command kills the entire application, no matter which thread executes it. Therefore, you should never call `exit` from a thread when your intention is to terminate only that thread.

Managing I/O Channels

Channels are shared resources in most programming languages. But in Tcl, channels are implemented as a per-interpreter resource. Only the standard I/O channels (`stdin`, `stdout`, and `stderr`) are shared.

Be careful with standard I/O channel on Windows and Macintosh.

When running `wish` on Windows and Macintosh prior to OS X, you don't have real standard I/O channels, but simulated `stdout` and `stderr` channels direct output to the special console window. As of Thread 2.5, these simulated channels appear in the main thread's channel list, but not in any other thread's channel list. Therefore, you'll cause an error if you attempt to access these channels from any thread other than the main thread.

Accessing Files from Multiple Threads

In a multi-threaded application, avoid having the same file open in multiple threads. Having the same file open for read access in multiple threads is safe, but it is more efficient to have only one thread read the file and then share the information with other threads as needed. Opening the same file in multiple threads for write or append access is likely to fail. Operating systems typically buffer information written to a disk on a per-channel basis. With multiple channels open to the same file, it's likely that one thread will end up overwriting data written by another thread. If you need multiple threads to have write access to a single file, it's far safer to have one thread responsible for all file access, and let other threads send messages to the thread to write the data. Example 21–9 shows the skeleton implementation of a logging thread. Once the log file is open, other threads can call the logger's `AddLog` procedure to write to the log file.

Example 21–9 A basic implementation of a logging thread.

```
set logger [thread::create {
   proc OpenLog {file} {
      global fid
      set fid [open $file a]
   }
   proc CloseLog {} {
      global fid
      close $fid
   }
   proc AddLog {msg} {
      global fid
      puts $fid $msg
   }
   thread::wait
}]
```

Transferring Channels between Threads

As long as you're working with Tcl 8.4 or later, the `Thread` extension gives you the ability to transfer a channel from one thread to another with the `thread::transfer` command. After the transfer, the initial thread has no further access to the channel. The symbolic channel ID remains the same in the target thread, but you need some method of informing the target thread of the ID, such as a thread-shared variable. The `thread::transfer` command blocks until the target thread has incorporated the channel. The following shows an example of transferring a channel, and simply duplicating the value of the channel ID in the target thread rather than using a thread-shared variable:

```
set fid [open myfile.txt r]
# ...
set t [thread::create]
thread::transfer $t $fid
# Duplicate the channel ID in the target thread
thread::send $t [list set fid $fid]
```

Another option for transferring channels introduced in `Thread 2.5` is `thread::detach`, which detaches a channel from a thread, and `thread::attach`, which attaches a previously detached channel to a thread. The advantage to this approach is that the thread relinquishing the channel doesn't need to know which thread will be acquiring it. This is useful when your application uses thread pools, which are described on page 342.

The ability to transfer channels between threads is a key feature in implementing a multi-thread server, in which a separate thread is created to service each client connected. One thread services the listening socket. When it receives a client connection, it creates a new thread to service the client, then transfers the client's communication socket to that thread.

Transferring socket channels requires special handling.

A complication arises in that you can't perform the transfer of the communication socket directly from the connection handler, like this:

```
socket -server ClientConnect 9001
proc ClientConnect {sock host port} {
    set t [thread::create { ... }]
    # The following command fails
    thread::transfer $t $sock
}
```

The reason is that Tcl maintains an internal reference to the communication socket during the connection callback. The `thread::transfer` command (and the `thread::detach` command) cannot transfer the channel while this additional reference is in place. Therefore, we must use the `after` command to defer the transfer until after the connection callback returns, as shown in Example 21–10.

Example 21–10 Deferring socket transfer until after the connection callback.

```
proc _ClientConnect {sock host port} {
    after 0 [list ClientConnect $sock $host $port]
}

proc ClientConnect {sock host port} {
    # Create the client thread and transfer the channel
}
```

One issue in early versions of Tcl 8.4 was a bug that failed to initialize Tcl's socket support when a socket channel was transferred into a thread. The work-around for this bug is to explicitly create a socket in the thread (which can then be immediately closed) to initialize the socket support, and then transfer the desired socket. This bug has been fixed, but Example 21–11 illustrates how you can perform extra initialization in a newly created thread before it enters its event loop:

Example 21–11 Working around Tcl's socket transfer bug by initializing socket support.

```
set t [thread::create {
   # Initialize socket support by opening and closing
   # a server socket.

   close [socket -server {} 0]

   # Now sockets can be transferred safely into this thread.

   thread::wait
}]
```

Example 21–12 integrates all of these techniques to create a simple multi-threaded echo server. Note that the server still uses event-driven interaction in each client thread. Technically, this isn't necessary for such a simple server, because once a client thread starts it doesn't expect to receive messages from any other thread. If a thread needs to respond to messages from other threads, it must be in its event loop to detect and service such messages. Because this requirement is common, this application demonstrates the event-driven approach.

Example 21–12 A multi-threaded echo server.

```
package require Tcl 8.4
package require Thread 2.5

if {$argc > 0} {
   set port [lindex $argv 0]
} else {
   set port 9001
}
```

```
socket -server _ClientConnect $port

proc _ClientConnect {sock host port} {

   # Tcl holds a reference to the client socket during
   # this callback, so we can't transfer the channel to our
   # worker thread immediately. Instead, we'll schedule an
   # after event to create the worker thread and transfer
   # the channel once we've re-entered the event loop.

   after 0 [list ClientConnect $sock $host $port]
}

proc ClientConnect {sock host port} {

   # Create a separate thread to manage this client. The
   # thread initialization script defines all of the client
   # communication procedures and puts the thread in its
   # event loop.

   set thread [thread::create {
      proc ReadLine {sock} {
         if {[catch {gets $sock line} len] || [eof $sock]} {
            catch {close $sock}
            thread::release
         } elseif {$len >= 0} {
            EchoLine $sock $line
         }
      }

      proc EchoLine {sock line} {
         if {[string equal -nocase $line quit]} {
            SendMessage $sock \
               "Closing connection to Echo server"
            catch {close $sock}
            thread::release
         } else {
            SendMessage $sock $line
         }
      }

      proc SendMessage {sock msg} {
         if {[catch {puts $sock $msg} error]} {
            puts stderr "Error writing to socket: $error"
            catch {close $sock}
            thread::release
         }
      }


      # Enter the event loop

      thread::wait

   }]
```

```
        # Release the channel from the main thread. We use
        # thread::detach/thread::attach in this case to prevent
        # blocking thread::transfer and synchronous thread::send
        # commands from blocking our listening socket thread.


        thread::detach $sock

        # Copy the value of the socket ID into the
        # client's thread

        thread::send -async $thread [list set sock $sock]

        # Attach the communication socket to the client-servicing
        # thread, and finish the socket setup.

        thread::send -async $thread {
            thread::attach $sock
            fconfigure $sock -buffering line -blocking 0
            fileevent $sock readable [list ReadLine $sock]
            SendMessage $sock "Connected to Echo server"
        }
    }

    vwait forever
```

Shared Variables

Standard Tcl variables are a per-interpreter resource; an interpreter has no access to variables in another interpreter. For the simple exchange of information between threads, you can substitute the *values* of variables into a script that you send to another thread, and obtain the return value of a script evaluated by another thread. But this technique is inadequate for sharing information among multiple threads, and inefficient when transferring large amounts of information.

The `Thread` extension supports the creation of *thread-shared variables*, which are accessible by all threads in an application. Thread-shared variables are stored independent of any interpreter, so if the thread that originally created a shared variable terminates, the shared variable continues to exist. Shared variables are stored in collections called *arrays*. The term is somewhat unfortunate, because while shared variable arrays are similar to standard Tcl arrays, they do not use the same syntax. Your application can contain as many shared variable arrays as you like.

Because of the special nature of shared variables, you cannot use the standard Tcl commands to create or manipulate shared variables, or use standard variable substitution syntax to retrieve their values. (This also means that you cannot use shared variables as a widget's `-textvariable` or `-listvariable`, with `vwait` or `tkwait`, or with variable traces.) All commands for interacting with shared variables are provided by the `Thread` extension in the `tsv` namespace. Most of the `tsv` commands are analogous to Tcl commands for creating and manipulating standard Tcl variables. Table 21–3 on page 346 describes all of the `tsv` commands.

You create a shared variable with `tsv::set`, specifying the array name, the variable name (sometimes also referred to as the shared array *element*), and the value to assign to it. For example:

```
tsv::set application timeout 10
```

To retrieve the value of a shared variable, either use `tsv::set` without a value or call `tsv::get`. The two commands shown below are equivalent:

```
tsv::set application timeout
tsv::get application timeout
```

All shared variable commands are guaranteed to be *atomic*. A thread locks the variable during the entire command. No other thread can access the variable until the command is complete; if a thread attempts to do so, it blocks until the variable is unlocked. This simplifies the use of shared variables in comparison to most other languages, which require explicit locking and unlocking of variables to prevent possible corruption from concurrent access by multiple threads.

This locking feature is particularly useful in the class of `tsv` commands that manipulate lists. Standard Tcl commands like `linsert` and `lreplace` take a list value as input, and then return a new list as output. Modifying the value of a list stored in a standard Tcl variable requires a sequence like this:

```
set states [linsert $states 1 California Nevada]
```

Doing the same with shared variables is problematic:

```
tsv::set common cities \
    [linsert [tsv::get common cities] 1 Yreka Winnemucca]
```

After reading the shared variable with `tsv::get`, another thread could modify the value of the variable before the `tsv::set` command executes, resulting in data corruption. For this reason, the `tsv` commands that manipulate list values actually modify the value of the shared variable. Data corruption by another thread won't occur because the shared variable is locked during the entire execution of the command:

```
tsv::linsert common cities 1 Yreka Winnemucca
```

Mutexes and Condition Variables

Mutexes and condition variables are thread synchronization mechanisms. Although they are used frequently in other languages, they aren't needed as often in Tcl because of Tcl's threading model and the atomic nature of all shared variable commands. All mutex and condition variable commands are provided by the `Thread` extension in the `thread` namespace.

Mutexes

A *mutex*, which is short for *mutual exclusion*, is a locking mechanism. You use a mutex to protect shared resources — such as shared variables, serial ports, databases, etc. — from concurrent access by multiple threads. Before accessing the shared resource, the thread attempts to *lock* the mutex. If no other thread currently holds the mutex, the thread successfully locks the mutex and can access the resource. If another thread already holds the mutex, then the attempt to lock the mutex blocks until the other thread releases the mutex.

This sequence is illustrated in Example 21–13. The first step is creating a mutex with the `thread::mutex create` operation, which returns a unique token representing the mutex. The same token is used in all threads, and so you must make this token available (for example, through a shared variable) to all threads that access the shared resource.

Example 21–13 Using a mutex to protect a shared resource.

```
# Create the mutex, storing the mutex token in a shared
# variable for other threads to access.

tsv::set db mutex [thread::mutex create]

# ...

# Lock the mutex before accessing the shared resource.

thread::mutex lock [tsv::get db mutex]

# Use the shared resource, and then unlock the mutex.

thread::mutex unlock [tsv::get db mutex]

# Lather, rinse, repeat as needed...

thread::mutex destroy [tsv::get db mutex]
```

Mutexes rely on threads being "good citizens."

Mutexes work only if all threads in an application use them properly. A "rogue" thread can ignore using a mutex and access the shared resource directly. Therefore, you should be very careful to use your mutexes consistently when designing and implementing your application.

Condition Variables

A *condition variable* is a synchronization mechanism that allows one or more threads to sleep until they receive notification from another thread. A condition variable is associated with a mutex and a boolean condition known as a *predicate*. A thread uses the condition variable to wait until the boolean predicate is true. A different thread changes the state of the predicate to true, and then notifies the condition variable. The mutex synchronizes thread access to the data used to compute the predicate value. The general usage pattern for the signalling thread is:

- Lock the mutex
- Change the state so the predicate is true
- Notify the condition variable
- Unlock the mutex

The pattern for a waiting thread is:

- Lock the mutex
- Check the predicate
- If the predicate is false, wait on the condition variable until notified
- Do the work
- Unlock the mutex

In practice, a waiting thread should always check the predicate inside a `while` loop, because multiple threads might be waiting on the same condition variable. A waiting thread automatically releases the mutex when it waits on the condition variable. When the signalling thread notifies the condition variable, all threads waiting on that condition variable compete for a lock on the mutex. Then when the signalling thread releases the mutex, one of the waiting threads gets the lock. It is quite possible for that thread then to change the state so that the predicate is no longer true when it releases the lock. For example, several worker threads forming a *thread pool* might wait until there is some type of job to process. Upon notification, the first worker thread takes the job, leaving nothing for the other worker threads to process.

This sequence for using a condition variable sounds complex, but is relatively easy to code. Example 21–14 shows the sequence for the signalling thread. The first step is creating a condition variable with the `thread::cond create` operation, which returns a unique token representing the condition variable. As with mutexes, the same token is used in all threads, and so you must make this token available (for example, through a shared variable) to all threads that access the condition variable. When the thread is ready to update the predicate, it first locks the associated mutex. Then it notifies the condition variable with `thread::cond notify` and finally unlocks the mutex.

Example 21–14 Standard condition variable use for a signalling thread.

```
# Create the condition variable and accompanying mutex.
# Use shared variables to share these tokens with all other
# threads that need to access them.

set cond [tsv::set tasks cond [thread::cond create]]
set mutex [tsv::set tasks mutex [thread::mutex create]]

# When we're ready to update the state of the predicate, we
# must first obtain the mutex protecting it.

thread::mutex lock $mutex

# Now update the predicate. In this example, we'll just set a
# shared variable to true. In practice, the predicate can be
# more complex, such as the length of a list stored in a
# shared variable being greater than 0.

tsv::set tasks predicate 1

# Notify the condition variable, waking all waiting threads.
# Each thread will block until it can lock the mutex.

thread::cond notify $cond

# Unlock the mutex.

thread::mutex unlock $mutex
```

Example 21–15 shows the sequence for a waiting thread. When a thread is ready to test the predicate, it must first lock the mutex protecting it. If the predicate is true, the thread can continue processing, unlocking the mutex when appropriate. If the predicate is false, the thread executes `thread::cond wait` to wait for notification. The `thread::cond wait` command atomically unlocks the mutex and puts the thread into a wait state. Upon notification, the thread atomically locks the mutex (blocking until it can obtain it) and returns from the `thread::cond wait` command. It then tests the predicate, and repeats the process until the predicate is true.

Example 21–15 Standard condition variable use for a waiting thread.

```
set mutex [tsv::get tasks mutex]
set cond [tsv::get tasks cond]

# Lock the mutex before testing the predicate.

thread::mutex lock $mutex

# Test the predicate, if necessary waiting until it is true.
```

```
while {![tsv::get tasks predicate]} {
   # Wait for notification on the condition variable.
   # thread::cond wait internally unlocks the mutex,
   # blocks until it receives notification, then locks
   # the mutex again before returning.

   thread::cond wait $cond $mutex
}

# We now hold the mutex and know the predicate is true. Do
# whatever processing is desired, and unlock the mutex when
# it is no longer needed.

thread::mutex unlock $mutex
```

Tcl's threading model greatly reduces the need for condition variables. It's usually much simpler to place a thread in its event loop with `thread::wait`, and then send it messages with `thread::send`. And for applications where you want a thread pool to handle jobs on demand, the `Thread` extension's built-in thread pool implementation is far easier than creating your own with condition variables.

Thread Pools

A *thread pool* is a common multi-threaded design pattern. A thread pool consists of several worker threads that wait for jobs to perform. When a job is sent to the thread pool, one of the available worker threads processes it. If all worker threads are busy, either additional worker threads are created to handle the incoming jobs, or the jobs are queued until worker threads are available.

The `tpool` namespace of the `Thread` extension provides several commands for creating and managing thread pools. Using these commands is much easier than trying to build your own thread pools from scratch using mutexes, condition variables, etc. Thread pool support was added to the `Thread` extension in version 2.5.

The `tpool::create` command creates a thread pool, returning the ID of the new thread pool. There are several options to `tpool::create` that allow you to configure the behavior of the thread pool. The `-minthreads` option specifies the minimum number of threads in the pool. This number of threads is created when the thread pool is created, and as worker threads in the pool terminate, new worker threads are created to bring the number up to this minimum. The `-maxthreads` option specifies the maximum number of worker threads allowed. If a job is posted to the thread pool and there are no idle worker threads available, a new worker thread is created to handle the job only if the number of worker threads won't exceed the maximum number. If the maximum has been reached, the job is queued until a worker thread is available. The `-idletime` option specifies the number of seconds that a worker thread waits for a new job before terminating itself to preserve system resources. And the `-initcmd` and `-exitcmd` options provide scripts to respectively initialize newly created worker threads and clean up exiting worker threads.

Once you have created a thread pool, you send *jobs* to it with the `tpool::post` command. A job consists of an arbitrary Tcl script to execute. The job is executed by the first available worker thread in the pool. If there are no idle worker threads, a new worker thread is created, as long as the number of worker threads doesn't exceed the thread pool maximum. If a new worker thread can't be created, the `tpool::post` command blocks until a worker thread can handle the job, but while blocked the posting thread still services its event loop.

The return value of `tpool::post` is a job ID. To receive notification that a job is complete, your thread must call `tpool::wait`. The `tpool::wait` command blocks, but continues to service the thread's event loop while blocked. Additionally, the `tpool::wait` command can wait for several jobs simultaneously, returning when any of the jobs are complete. The return value of `tpool::wait` is a list of completed job IDs.

After `tpool::wait` reports that a job is complete, you can call `tpool::get` to retrieve the result of the job, which is the return value of the last command executed in the job script. If the job execution resulted in an error, the error is "reflected" to the posting thread: `tpool::get` raises an error and the values of `errorInfo` and `errorCode` are updated accordingly.

Finally, a thread pool can be preserved and released in much the same way as an individual thread. Each thread pool maintains an internal reference count, which is initially set to 0 upon creation. Any thread can increment the reference count afterwards by executing `tpool::preserve`, and decrement the reference count by executing `tpool::release`. If a call to `tpool::release` results in a reference count of 0 or less, the thread pool is marked for termination. Any further reference to a thread pool once it is marked for termination results in an error.

The `Thread` Package Commands

The commands of the `Thread` extension are grouped into three separate namespaces, based on their functionality. This section summarizes the commands found in each namespace.

The `thread` Namespace

The `thread` namespace contains all of the commands for creating and managing threads, including inter-thread messaging, mutexes, and condition variables. Table 21–1 describes all of the commands contained in the `thread` namespace.

Table 21–1 The commands of the `thread` namespace.

`thread::attach` *`channel`*	Attaches the previously detached *`channel`* into current interpreter of the current thread.
`thread::cond create`	Returns a token for a newly created condition variable.
`thread::cond destroy` *`cond`*	Destroys the specified condition variable.
`thread::cond notify` *`cond`*	Wakes up all threads waiting on the specified condition variable.
`thread::cond wait` *`cond mutex`* `?`*`ms`*`?`	Blocks until the specified condition variable is signaled by another thread with `thread::cond notify`, or until the optional timeout in milliseconds specified by *`ms`* expires. The *`mutex`* must be locked by the calling thread before calling `thread::cond wait`. While waiting on the *`cond`*, the command releases *`mutex`*. Before returning to the calling thread, the command re-acquires *`mutex`* again.
`thread::configure` *`id`* `?`*`option`* `?`*`value`*`? ?`*`option value...`*`?`	Queries or sets thread configuration options, as described in Table 21–2.
`thread::create ?-joinable? ?-preserved? ?`*`script`*`?`	Creates a thread, returning the thread's ID. The `-joinable` flag allows another thread to wait for termination of this thread with `thread::join`. The `-preserved` flag sets the thread's initial reference count to 1, rather than the default of 0. (See `thread::preserve` and `thread::release`.) If provided, the thread executes the *`script`*, then exits; otherwise, it enters an events loop to wait for messages.
`thread::detach` *`channel`*	Detaches the specified *`channel`* from the current thread so that it no longer has access to it. Any single thread can then `thread::attach` the channel to gain access to it.
`thread::errorproc ?`*`proc`*`?`	Registers a procedure to handle errors that occur when performing asynchronous `thread::send` commands. When called, *`proc`* receives two argument: the ID of the thread that generated the error, and the value of that thread's `errorInfo` variable.
`thread::eval ?-lock` *`mutex`*`?` *`arg`* `?`*`arg...`*`?`	Concatenates the arguments and evaluates the resulting script under the *`mutex`* protection. If no *`mutex`* is specified, an internal static one is used for the duration of the evaluation.
`thread::exists` *`id`*	Returns boolean indicating whether or not the specified thread exists.
`thread::id`	Returns the current thread's ID.

Table 21–1 The commands of the `thread` namespace. (Continued)

`thread::join` *`id`*	Blocks until the target thread terminates. (Available only with Tcl 8.4 or later.)
`thread::mutex create`	Returns a token for a newly created mutex.
`thread::mutex destroy` *`mutex`*	Destroys the *`mutex`*.
`thread::mutex lock` *`mutex`*	Locks the *`mutex`*, blocking until it can gain exclusive access.
`thread::mutex unlock` *`mutex`*	Unlocks the *`mutex`*.
`thread::names`	Returns a list of the IDs of all running threads.
`thread::preserve` *`?id?`*	Increments the reference count of the indicated thread, or the current thread if no *`id`* is given.
`thread::release ?-wait? ?`*`id`*`?`	Decrements the reference count of the indicated thread, or the current thread if no *`id`* is given. If the reference count is 0 or less, mark the thread for termination. If `-wait` is specified, the command blocks until the target thread terminates.
`thread::send ?-async?` *`id script`* `?`*`varname`*`?`	Sends the *`script`*, to thread *`id`*. If `-async` is specified, do not wait for *`script`* to complete. Stores the result of *`script`* in *`varname`*, if provided.
`thread::transfer` *`id channel`*	Transfers the open *`channel`* from the current thread to the main interpreter of the target thread. This command blocks until the target thread incorporates the channel. (Available only with Tcl 8.4 or later.)
`thread::unwind`	Terminates a prior `thread::wait` to cause a thread to exit. Deprecated in favor of `thread::release`.
`thread::wait`	Enters the event loop.

The `thread::configure` command allows an application to query and set thread configuration options, in much the same way as the `fconfigure` command configures channels. Table 21–2 lists the available thread configuration options.

Table 21–2 Thread configuration options.

`-eventmark` *`int`*	Specifies the maximum number of pending scripts sent with `thread::send` that the thread accepts. Once the maximum is reached, subsequent `thread::send` messages to this script block until the number of pending scripts drops below the maximum. A value of 0 (default) allows an unlimited number of pending scripts.
`-unwindonerror` *`boolean`*	If true, the thread “unwinds” (terminates its event loop) on uncaught errors. Default is false.

The `tsv` Namespace

The `tsv` namespace contains all of the commands for creating and managing thread shared variables. Table 21–3 describes all of the commands contained in the `tsv` namespace.

Table 21–3 The commands of the `tsv` namespace.

`tsv::append` *`array element value`* `?`*`value`* `...?`	Appends to the shared variable like `append`.
`tsv::exists` *`array`* `?`*`element`*`?`	Returns boolean indicating whether the given *`element`* exists, or if no *`element`* is given, whether the shared *`array`* exists.
`tsv::get` *`array element`* `?`*`varname`*`?`	Returns the value of the shared variable. If *`varname`* is provided, the value is stored in the variable, and the command returns 1 if the *`element`* existed, 0 otherwise.
`tsv::incr` *`array element`* `?`*`increment`*`?`	Increments the shared variable like `incr`.
`tsv::lappend` *`array element value`* `?`*`value`* `...?`	Appends elements to the shared variable like `lappend`.
`tsv::lindex` *`array element index`*	Returns the indicated element from the shared variable, similar to `lindex`.
`tsv::linsert` *`array element index value`* `?`*`value`* `...?`	Atomically inserts elements into the shared variable, similar to `linsert`, but actually modifying the variable.
`tsv::llength` *`array element`*	Returns the number of elements in the shared variable, similar to `llength`.
`tsv::lock` *`array arg`* `?`*`arg`* `...?`	Concatenates the *`args`* and evaluates the resulting script. During script execution, the command locks the specified shared *`array`* with an internal mutex.
`tsv::lpop` *`array element`* `?`*`index`*`?`	Atomically deletes the value at the *`index`* list position from the shared variable and returns the value deleted. The default *`index`* is 0.
`tsv::lpush` *`array element value`* `?`*`index`*`?`	Atomically inserts the *`value`* at the *`index`* list position in the shared variable. The default *`index`* is 0.
`tsv::lrange` *`array element first last`*	Returns the indicated range of elements from the shared variable, similar to `lrange`.
`tsv::lreplace` *`array element value`* `?`*`value`* `...?`	Atomically replaces elements in the shared variable, similar to `lreplace`, but actually modifying the variable.
`tsv::lsearch` *`array element`* `?`*`mode`*`?` *`pattern`*	Returns the index of the first element in the shared variable matching the *`pattern`*, similar to `lsearch`. Supported modes are: `-exact`, `-glob` (default), and `-regexp`.

Table 21–3 The commands of the `tsv` namespace. (Continued)

`tsv::move` *`array old new`*	Atomically renames the shared variable from *`old`* to *`new`*.
`tsv::names ?`*`pattern`*`?`	Returns a list of all shared variable arrays, or those whose names match the optional glob pattern.
`tsv::object` *`array element`*	Creates and returns the name of an accessor command for the shared variable. Other `tsv` commands are available as subcommands of the accessor to manipulate the shared variable.
`tsv::pop` *`array element`*	Atomically returns the value of the shared variable and deletes the *`element`*.
`tsv::set` *`array element`* `?`*`value`*`?`	Sets the *`value`* of the shared variable, creating it if necessary. If *`value`* is omitted, the current value is returned.
`tsv::unset` *`array`* `?`*`element`*`?`	Deletes the shared variable, or the entire *`array`* if no *`element`* is specified.

The `tpool` Namespace

The `tpool` namespace contains all of the commands for creating and managing thread pools. Table 21–4 describes all of the commands contained in the `tpool` namespace.

Table 21–4 The commands of the `tpool` namespace.

`tpool::create ?`*`options`*`?`	Creates a thread pool, returning the thread pool's ID. Table 21–5 describes supported configuration options.
`tpool::post` *`tpoolId script`*	Sends a Tcl *`script`* to the specified thread pool for execution, returning the ID of the posted job. This command blocks (entering the event loop to service events) until a worker thread can service the job
`tpool::wait` *`tpoolId jobList`* `?`*`varName`*`?`	Blocks (entering the event loop to service events) until one or more of the jobs whose IDs are given by the *`jobList`* argument are completed. Returns a list of completed jobs from *`jobList`*. If provided, *`varName`* is set to a list of jobs from *`jobList`* that are still pending.
`tpool::get` *`tpoolId jobId`*	Returns the result of the specified *`jobId`*. `tpool::wait` must have reported previously that the job is complete. If no error occurred in the job, the result is the return value of the last command executed in the job script. Any error encountered in job execution is in turn thrown by `tpool::get`, with the `errorCode` and `errorInfo` variables set appropriately.

Table 21–4 The commands of the `tpool` namespace. (Continued)

`tpool::names`	Returns a list of existing thread pool IDs.
`tpool::preserve` *`tpoolId`*	Increments the reference count of the indicated thread pool.
`tpool::release` *`tpoolId`*	Decrements the reference count of the indicated thread pool. If the reference count is 0 or less, mark the thread pool for termination.

The `tpool::create` command supports several options for configuring thread pools. Table 21–5 lists the available thread pool configuration options.

Table 21–5 Thread pool configuration options.

`-minthreads` *`number`*	The minimum number of threads. If the number of live threads in the thread pool is less than this number (including when the thread pool is created initially), new threads are created to bring the number up to the minimum. Default is 0.
`-maxthreads` *`number`*	The maximum number of threads.When a job is posted to the thread pool, if there are no idle threads and the number of existing worker threads is at the maximum, the thread posting the job blocks (in its event loop) until a worker thread is free to handle the job. Default is 4.
`-idletime` *`seconds`*	The maximum idle time, in seconds, before a worker thread exits (as long as the number of threads doesn't drop below the `-minthreads` limit). Default value is 0, meaning idle threads wait forever.
`-initcmd` *`script`*	A script that newly created worker threads execute.
`-exitcmd` *`script`*	A script that worker threads execute before exiting.