22

Threads

In the previous chapter, we described the use of coroutines for concurrent computation. We now describe a more heavyweight, but also more powerful, alternative that uses operating system threads. We will contrast the two later in Section 22.15.

By default, Tcl applications run within a single operating system thread. Generally speaking, threads are used in software for one of several reasons:

- In languages with no or limited support for event-driven I/O, threads are used for performing I/O without blocking other computational tasks.
- Since each operating system thread runs on its own stack, threads maintain their own implicit context and state. This allows simpler coding of some programs where multiple tasks have to run concurrently and share results.
- Some application programming interfaces, database drivers for instance, may only provide a blocking mode of operation. In this case as well, threads may be used for these operations while allowing other computational tasks to proceed.
- Threads are also used for performance reasons in compute-intensive applications as each thread may run on its own processor in multiprocessor configurations.

When it comes to Tcl, the first two motivations above are not very compelling. The use of threads specifically for I/O is obviated by Tcl's first-class native support for asynchronous and event-driven I/O. Similarly, as discussed in Chapter 21, Tcl's coroutines also run on their own private stacks making the use of operating system threads purely for this purpose unnecessary.

Thus the use of threads in Tcl is driven primarily by the last two factors — a desire to utilize multiple processors within a program and for non-blocking operations with programming interfaces that only support a synchronous mode.

22.1. Enabling thread support: the `Thread` package

Use of threads within a Tcl interpreter requires the Tcl libraries to have been built with thread support. This has been the default on Windows platforms for many years. On other platforms, thread support requires the `--enable-threads` build option to have been specified to the `configure` script at the time of building the Tcl libraries. This is the default only in very recent Tcl releases for non-Windows platforms. Even then, some Tcl applications explicitly disable thread support for one of two reasons — non-threaded builds are slightly faster, and threaded builds do not work correctly in some cases involving binary extensions that invoke the `fork` system call.

You can check if the Tcl interpreter has thread support enabled by either inspecting the `threaded` element of the `tcl_platform` array or with the `tcl::pkgconfig` call.

```
puts $tcl_platform(threaded) → 1
tcl::pkgconfig get threaded  → 1
```

In addition to the core Tcl libraries being multithread-enabled, working with threads at the script level also requires the `Thread` package to be loaded.

```
package require Thread → 2.8.0
```

The commands implemented by the package are placed in four namespaces shown in Table 22.1 based on their functionality.

Table 22.1. Thread package namespaces

Namespace	Description
`thread`	Base commands for working with threads
`tpool`	Implements a pool of worker threads
`tsv`	Commands for sharing data between threads
`ttrace`	Enables sharing of interpreter state across threads

In addition to the above, other third-party packages are available that supplement or enhance the functionality of the `Thread` package. We however do not describe them in this book.

22.2. Threading model

Before going into the specifics of individual commands, a few words regarding the threading model used by Tcl are in order.

Unless otherwise noted, the term “thread” will henceforth refer to any thread running a Tcl interpreter. A process may contain other threads as well, even those created in the background by the Tcl libraries for specific tasks like socket I/O on Windows. These are not germane to our discussion as they are not visible at the script level.

When a Tcl application starts up, there is only one thread executing Tcl code. We refer to this as the *main thread* and it has a special role as we will see in a bit. This thread executes Tcl code in a Tcl interpreter which may create threads each of which runs a Tcl interpreter. These may in turn create additional threads ad infinitum.

Threads and interpreters

We discussed the use of multiple Tcl interpreters at length in Chapter 20. The difference here is that the interpreter in a new thread is **not** a slave of the interpreter that created the thread. Each thread runs an independent “top-level” interpreter. Naturally, these interpreters may in turn create slave interpreters running in the same thread. As always, each interpreter, whether a top-level interpreter in a thread or a slave to one, is responsible for its own initialization such as loading any packages it needs, creating the proper namespaces and so on.

The Tcl threading model thus allows for multiple threads where each thread is (potentially) running multiple interpreters. **However each interpreter runs within the thread that created it.** There is no sharing of interpreter contexts between threads. Normal Tcl code therefore does not need to worry about having to synchronize access to its variables and data structures.

Thread run modes

A Tcl thread generally runs in one of two modes. It may execute a script and terminate, or it may sit in the event loop waiting for commands from other threads and executing them until it is asked to exit. Threads may be started and exit at any time with one important condition: **when the main thread terminates, the entire process exits**. It is therefore important for the main thread to monitor the status of the additional threads before it exits.

Thread reference counting

Some threads may act as worker threads wherein they run scripts sent to them by other threads. These threads have no defined exit point and have no way of knowing when their services are no longer required. Moreover, their “client” threads do not necessarily know who else might be requiring the services of that worker thread

either. To help with this situation, a reference count is maintained for each thread. The worker thread can choose to exit when this reference count reaches 0. Conversely, threads that have an interest in that thread being available can increment its reference count and only decrement it when the worker thread is no longer of interest.

Thread interaction

The design center for threads in Tcl is one where thread interaction is not expected to be fine grained. They are expected to run independently for the most part with occasional interaction and exchange of data. Thus unlike many programming languages, Tcl does not permit **normal** variables to be shared across threads (and in fact even across interpreters within a single thread). Threads interact by sending each other messages in the form of scripts which are executed by a target thread with the result being optionally returned to the source thread. This model simplifies multithread programming as the potential for race conditions, deadlocks and the like is greatly reduced.

One point to keep in mind is that the inter-thread communication "messages" are always received by the top-level interpreter of each thread. It can then be programmed to dispatch to any slave interpreters if so desired.

Shared properties

While threads run independently, there are some properties that shared. This is actually not specific to Tcl but is true for processes in general on most modern operating systems.

- The current working directory, as returned by the pwd command, is common to all threads. Changing it with the `cd` command in any thread changes it for the entire process.
- The environment variables, available in the `env` global array of the interpreter, is also shared amongst all threads. Changing a value in one thread is reflected across all threads in the process.

22.3. Creating threads: `thread::create`

Having summarized the threading model used by Tcl, we are now ready to describe the actual nitty-gritty of thread commands. Threads are created with the `thread::create` command.

```
thread::create ?-joinable? ?-preserved? ?SCRIPT?
```

The `-joinable` and `-preserved` options are related to managing thread lifetimes and are described in later sections. If the optional *`SCRIPT`* argument is not specified, the created thread will wait in a loop for messages to arrive. Otherwise, it will execute *`SCRIPT`* in its top-level interpreter and terminate when the script completes. Of course, as we will see *`SCRIPT`* itself may contain commands that activate the message communication loop. Note that the result of the script execution is not made available unless the script itself takes some action to return it as a message.

The command returns a thread id for the created thread that can be used for various purposes such as sending messages and synchronization.

Here is a short example that creates a thread to retrieve a URL in the background and save it to a file.

```
thread::create {
    package require http
    package require fileutil
    set tok [http::geturl http://www.example.com]
    fileutil::writeFile example.html [http::data $tok]
    http::cleanup $tok
}
→ tid00000000000014B4
```

The script loads any packages that it needs as it starts in a new interpreter in the created thread. It uses the blocking form of the `http::geturl` command but because it is running in a separate thread, our current thread can proceed with other computations. Once the script completes, the created thread will terminate.

Our simple example is missing many pieces. Since the threads run independently and asynchronously, the main thread has no way to know when the created thread has completed its task. There is also no mechanism for reporting errors and exceptions. We will come to these topics in a little bit.

22.4. Interthread communication

Threads communicate with each other through messages that are actually scripts executed in the receiving thread.

22.4.1. Waiting for messages: `thread::wait`

For a thread to receive a message, it must be running its event loop. A thread that is created without the *`SCRIPT`* argument specified to the `thread::create` command runs its event loop by default and is ready to receive messages. Alternatively, the script passed to the thread may use the `thread::wait` command to enter its event loop and process messages. The following commands are thus equivalent.

```
thread::create
thread::create {
    thread::wait
}
```

The latter form is generally used when there is some initialization required before messages are received. The pattern looks like

```
thread::create {
    ...load packages...
    ...other initialization...
    thread::wait
    ...cleanup...
}
```

The `thread::wait` command is similar to the `vwait` command in that it enters the thread's event loop dispatching events. The difference is that while `vwait` stays in the loop until a specified variable is set, `thread::wait` stays in the loop until the thread's reference count drops to 0 signalling it is time for the thread to exit. We discuss this in further detail in Section 22.2.

Do **not** use `vwait` in a thread in lieu of `thread::wait` for the purposes of entering the event loop at the top level. The former is ignorant with respect to thread reference counts and therefore does not return when the count drops to 0. The thread will then not exit as expected.

Once the thread's event loop is running, events are handled just as we described in Chapter 15. Messages sent from other threads are just a special form of event. The associated event handler is the message itself which is a script to be executed in the context of the receiving thread's top-level interpreter.

22.4.1.1. Limiting message queue size: `thread::configure -eventmark`

The number of outstanding messages permitted on a thread's receive queue can be controlled through the configuration option `-eventmark` set with the `thread::configure` command.

```
thread::configure TID -eventmark ?QLIMIT?
```

If the *`QLIMIT`* argument is not specified, the command returns the current value of the option. If *`QLIMIT`* is 0, no limit is placed on the number of messages in the queue. Otherwise, it specifies the maximum number of messages allowed in the queue. When this limit is reached, even the asynchronous form of the `thread::send` command will block until the receiving thread processes enough messages for the number of queued messages to drop below the limit.

22.4.2. Sending messages: thread::send

On the sender's side, the thread::send command is used to send messages to a target thread.

```
thread::send ?-async? ?-head? TID SCRIPT ?VARNAME?
```

Here *TID* is the thread id of the target thread as returned by the thread::create command. The command appends, or prepends if the -head option is specified, the supplied *SCRIPT* argument to the target's event queue.

The command may operate either synchronously or asynchronously. If the -async option is not present, the command works in synchronous mode and waits for *SCRIPT* to be executed. If the *VARNAME* argument is not provided, the command returns the result of evaluation of *SCRIPT* in the target thread. If *VARNAME* is provided, the result of the command is the the **return code** from the script evaluation. For example, the command will return 0 on a successful evaluation of the script, 1 if an exception was raised and so on. The **result** of the script is stored in the variable *VARNAME*.

If the -async option is specified, the command returns immediately (with one exception described below) with an empty result. If the *VARNAME* argument is provided, the result will be placed in the variable of that name when the script completes. The sending thread can track completion of the script by either placing a trace on the variable or waiting on it with vwait. In case of errors or other exceptions this variable is still updated with the result, for example the error message. However, as the return code is not available, there is no way to distinguish this from a normal completion without somehow encoding this information in the result value.

Even with the -async option specified, there is one situation where the thread::send command may block. If there is an upper limit on the message queue size for the receiving thread as described in Section 22.4.1.1, the thread::send will block until the queue shrinks sufficiently for the new message to be placed on it.

Handling of erors during evaluation of the script is discussed in detail in Section 22.7.

You need to be careful to avoid deadlocks when using the synchronous form of thread::send. Thread A may send a script to Thread B. If as part of the evaluation of the script in Thread B, it executes a send back to Thread A, deadlock will result. Thread A cannot respond to this send because it is still blocked awaiting the result of the first send. That never completes because Thread B is still waiting for the result of the second send. This situation is not specific to Tcl or the Thread package. The same will apply to all synchronous communication mechanisms in any language or platform.

22.4.3. Broadcasting messages: thread::broadcast

Whereas the thread::send command sends a message to a specific thread, the thread::broadcast message sends a message to all existing threads that have been created with the thread::create command.

```
thread::broadcast SCRIPT
```

The command always works asynchronously and has no means of returning the response from each thread unless explicitly done via a thread::send from within the supplied script. Notice that the sending thread is not included amongst the recipients.

```
% thread::create
→ tid00000000000004F0
% thread::create
→ tid0000000000000660
% thread::broadcast [format {
    thread::send -async %s "puts {Ping from [thread::id]}"
} [thread::id]]
→ Ping from tid00000000000004F0
  Ping from tid0000000000000660
```

As an aside, the above example illustrates use of `format` to create the code fragment to be sent to the remote threads. This makes it a little less awkward to construct the script where part of the substitution happens in the current thread and part in the receiving thread. See Section 10.7.1.2.

22.5. Thread lifetime management

The issue of tracking and managing thread lifetimes comes up in a couple of scenarios:

- It is sometimes important to know when a thread has completed execution. For example, the main thread may fire off several threads to execute various tasks. It must make sure it does not exit before those threads as completion of the main thread causes the entire process to exit.
- Conversely, a thread may need to know when it itself may exit. For example, a server thread may service requests from multiple independent client threads. It needs some way of knowing when its services are no longer required so it can exit and not take up unnecessary resources. Moreover, the client threads may not be even aware of each other so any solution must not presume any coordination between them.

Although both these issues can be solved using explicit low-level synchronization primitives we discuss later, the scenarios are common enough that the package provides simpler built-in solutions for both.

22.5.1. Waiting for thread completion: `thread::join`

The `thread::join` command can be used to wait for the completion of a thread.

```
thread::join TID
```

Here *`TID`* must be the thread id of a *joinable* thread, one that is created using the `-join` option of the `thread::create` command. An attempt to join a thread that is not joinable will generate an error.

The command blocks and returns only when the specified thread has exited. The result of the command is the exit code for the thread. No events are processed in the thread calling `thread::join` until the command returns. At that point the thread being waited on has exited and the thread id *`TID`* no longer valid.

Any thread, not necessarily the parent thread, may use `thread::join` to wait for the completion of another thread. However, only one thread may issue a `thread::join` for a particular thread. Further attempts to wait on that same thread will raise an error.

Every joinable thread must be waited on by some `thread::join` command. Otherwise, when it completes it will stay around as a “zombie” taking up system resources.

To wait for the completion of multiple threads, you need to serially wait on each in turn. To expand a little on an earlier example, we can start two threads to download URL’s and wait for their completion as below.

```
set fetch_script {
    package require http
    package require fileutil
    set tok [http::geturl http://www.example.com/%1$s]
    fileutil::writeFile %1$s.html [http::data $tok]
    http::cleanup $tok
}
set tid1 [thread::create -joinable [format $fetch_script page1]]
set tid2 [thread::create -joinable [format $fetch_script page2]]
thread::join $tid1
thread::join $tid2
puts "[file exists page1.html] [file exists page2.html]"
→ 1 1
```

An alternative to using thread::join to ensure a thread has exited is to use the thread::names command discussed in Section 22.9. This returns a list of threads created with thread::create that are still running. The command can be invoked periodically and the returned list checked for the thread(s) of interest. This method may be more suitable when you do not want the calling thread to block until the target thread exits.

22.5.2. Thread reference counting: thread::preserve, thread::release

We now move on to the next topic related to thread lifetimes — how a thread knows when to exit its event loop and terminate. More specifically, how does the thread::wait command which sits in the event loop processing messages know that it should stop looking for more events and return to the caller.

The answer lies in an internal reference count that is maintained for each thread. When the value of this reference drops to 0 or a negative number, the thread::wait loop is terminated regardless of any additional messages that may be pending in the queue. Note that the thread itself is not terminated. What happens next will depend on the commands following the thread::wait call. In the normal case, the thread should do clean up and exit.

The thread reference count can be manipulated with the thread::preserve and thread::release commands which increment and decrement the reference count respectively.

```
thread::preserve ?TID?
thread::release ?TID?
```

The commands operate on the reference count of the thread identified by *TID* which defaults to the current thread if unspecified. Both commands return the reference count value after it has been modified.

The reference documentation advises that the thread should not be referenced if the return value from thread::release is 0 or negative. This is misleading. You should not reference the released thread after a thread::release from the thread doing the release irrespective of the return value. This is because other threads holding a reference to the released thread might have released it in the meantime.

The simplest demonstration of thread management is creation of a single thread which will implicitly run its thread::wait loop.

```
set tid [thread::create] → tid0000000000001804
```

The new thread is created with a reference count of 0 by default. We can send this thread some compute-intensive tasks.

```
thread::send $tid {expr 1+1} → 2
thread::send $tid {expr 2*2} → 4
```

Once we are done with the thread, we call thread::release on it. This decrements its reference count causing the thread::wait loop to return allowing the thread to exit.

```
thread::release $tid → 0
```

In a slightly more complex scenario, multiple threads might be using this “server” thread we created in which case the application has to ensure that the thread does not disappear until all client threads are done with it. To ensure this, each client thread should call thread::preserve on the server thread and match that with a thread::release once they are done with it. This is true of the creating thread as well if it will also make use of the server thread. Alternatively, it can use the -preserved option on thread creation. As we noted earlier, new threads are created with a default reference count of 0. However, if the -preserved option is provided to thread::create, the new thread is created with a reference count of 1. The creating thread must then execute a corresponding thread::release at the appropriate time.

22.6. Canceling script execution: thread::cancel

The thread::cancel command cancels the evaluation of a script in a thread.

```
thread::cancel ?-unwind? TID ?RESULT?
```

The command cancels the evaluation of the script running in the target thread identified by *TID* by raising an exception. If the *RESULT* argument is provided, it is used as the result or error message of the exception. Otherwise a default error message is used. If the -unwind option is specified, the exception cannot be caught by the target thread and propagates all the way to the top level of the thread. Note that the thread is **not** terminated.

The following console sample session should clarify the operation. It creates a thread and defines a procedure demo within it that prints two lines separated by a delay. The delay is simply to permit us to type a cancel command in time.

```
% set tid [thread::create]
→ tid0000000000002EA0
% thread::send $tid {proc demo {} {puts foo; after 5000; puts bar}}
```

We then ask the thread to execute the procedure and immediately issue a cancel command.

```
% thread::send -async $tid demo
→ foo

% thread::cancel $tid
% Error from thread tid0000000000002EA0
→ eval canceled
      while executing
  "after 5000"
      (procedure "demo" line 1)
      invoked from within
  ...Additional lines omitted...
```

As seen, bar is never printed because the evaluation of the script was aborted in the meantime. However, the thread is still active and can process further messages. We can try it again but this time without cancellation.

```
% thread::send -async $tid demo
→ foo
  bar
```

22.7. Error handling in threads: thread::errorproc

The manner in which errors and exceptions are handled in threads depends on whether the thread is executing a script passed as an argument to thread::create or is running scripts via the thread::wait loop. Furthermore, the latter case is further distinguished by whether the script execution is synchronous or not.

The simpler case is that of a thread executing a script passed through the thread::create command. Any errors or exceptions will result in the thread being terminated.

The above does not apply if the script invokes the thread::wait command and the exception is thrown in a script executed through its event loop. The behaviour for that case discussed later in this section.

By default, Tcl writes the error stack from the exception to the standard error channel. This behaviour can be changed by calling the thread::errorproc command.

```
thread::errorproc ?ERRORPROC?
```

If provided an argument, this command registers it as a procedure to be called to report the error. Tcl will append two additional arguments to the call: the id of the thread that generated the exception, and the error stack. An example procedure would be:

```
proc thread_error_handler {tid error_stack} {
    puts -nonewline stdout "[clock format [clock seconds]]: "
    puts "Thread $tid died. Error stack:\n$error_stack"
}
```

We can then register it and check out the effects.

```
% thread::errorproc thread_error_handler
% thread::create {error "Something horrible happened"}
→ tid0000000000000C60
  Sun Feb 19 22:56:34 IST 2017: Thread tid0000000000000C60 died. Error stack:
  Something horrible happened
      while executing
  "error "Something horrible happened""
```

The registered error handler is run in the interpreter that registered it irrespective of the thread creator. Moreover, that interpreter must have an active event loop running.

If thread::errorproc is called without any arguments, it returns the currently registered error handler.

```
% thread::errorproc
→ thread_error_handler
```

To reset the error handler to the default, call thread::errorproc with an empty string as the argument.

The other case to consider is that of exceptions thrown by scripts run by the thread::wait event loop, i.e. one sent by the thread::send command. Unlike for scripts passed as arguments to the thread::create command, exceptions in scripts run within thread::wait do not cause the thread to die by default. These exceptions are reported as described below but the thread continues to run the the event loop processing messages. The thread::configure command's -unwindonerror option may be used to change this behaviour.

```
thread::configure TID -unwindonerror ?UNWIND?
```

If *UNWIND* is 1, exceptions during execution of scripts from thread::wait will cause the thread to exit. The default value is 0. If the *UNWIND* argument is not given, the command returns the current value of the option.

Reporting of exceptions in scripts run with thread::send depends on whether the -async option was specified with the command. If present, exceptions are reported as described earlier for scripts passed to thread::create.

If the -async handler was not specified, the thread::send command executes synchronously. If the *VARNAME* argument is provided, the command behaves similar to the catch command. The result of the command is the return code and the script result is placed in the variable *VARNAME*.

```
% set tid [thread::create]
→ tid0000000000000FF4
% thread::send $tid {expr 1+1} result ❶
→ 0
% set result
```

```
→ 2
% thread::send $tid {error "An error!"} result ❷
→ 1
% set result
→ An error!
```

❶ Normal completion
❷ Error exception

If *VARNAME* was not provided, then a synchronous send results in exceptions being directly reflected as for any other command.

```
% thread::send $tid {error "An error!"}
Ø An error!
% set errorInfo
→ An error!
      while executing
  "error "An error!""
      invoked from within
  "thread::send $tid {error "An error!"}"
```

22.8. Threads and I/O channels: thread::transfer

As we described in Chapter 20, I/O channels are specific to an interpreter and not shared. The same then naturally applies to interpreters running in different threads. With the exception of standard I/O channels, a channel created in one thread is not available in other threads and cannot be shared. It can however be transferred from one thread to another with the thread::transfer command.

```
thread::transfer TID CHAN
```

The channel *CHAN* is made accessible to the **main** interpreter of the thread identified by *TID* and no longer available to the current interpreter. The channel name remains the same in the thread to which it is transferred. However, this name has to be explicitly provided to the target thread as otherwise it has no means to know how to reference the channel. The following trivial example illustrates the sequence.

We open a channel to a temporary file in our current interpreter and write to it.

```
set chan [file tempfile tempname]
puts $chan "File created by [thread::id]"
chan names
→ file413a920 stdin stdout rc28 stderr
```

Notice that chan names lists the channel amongst the open channels in this interpreter.

We then create a new thread and transfer the channel to it. We see that chan names no longer shows the channel as being available in the current interpreter. It does show up in the list of channels in the new thread.

```
set tid [thread::create]
thread::transfer $tid $chan
chan names
thread::send $tid {chan names}
→ file413a920 stderr stdout stdin
```

Although the channel is now available in the new thread, its name is not known to the interpreter in the thread. We have to inform it through some means. In our example, we simply set the global chan in the new thread to the channel name.

```
thread::send $tid [list set chan $chan]
→ file413a920
```

Finally we have the new thread write to the channel and close it.

```
thread::send $tid {
    puts $chan "Message from [thread::id]"
    close $chan
}
thread::release $tid
→ 0
```

As confirmation, we can read back the temporary file.

```
print_file $tempname
→ File created by tid0000000000002FE8
  Message from tid000000000000081C
```

There is one issue you have to be wary of when transferring channels. The `thread::transfer` command will block until the target thread has internally completed acceptance of the channel. This leads to the potential for deadlocks as illustrated by the following fragment.

```
thread::send $tid { set chan [file tempfile] }
thread::send $tid [list thread::transfer [thread::id] \$chan]
```

The thread executing the above code has the target thread open a temporary file. It then **synchronously** asks it to transfer the open channel back. However, because the `thread::send` command is used in synchronous mode, the first thread is blocked and cannot complete acceptance of the transfer. This results in the second thread being blocked as well since the `thread::transfer` command will not return until the channel transfer is complete. Deadlock results.

The general lesson in the above is to always be careful when using synchronous calls between threads. Again, this is nothing specific to Tcl or its `Thread` package.

An alternative to `thread::transfer` for moving channels between threads is the pair of commands `thread::detach` and `thread::attach`. The former removes the channel from the calling interpreter. The channel then stays in an “unowned” state until an interpreter (generally, not necessarily) in another thread calls `thread::attach` to claim ownership of the channel. This splitting of `thread::transfer` functionality is useful when the thread to which the channel should be transferred is not known a priori. This situation is common when using a “worker thread” model of the kind we will describe in Section 22.12.

Note that the same care for avoiding deadlocks must be taken with `thread::detach` / `thread::attach` as was described above for `thread::transfer`.

Transferring socket channels

There is one minor quirk to keep in mind regarding transfer of server sockets. As described in Chapter 18, a server socket is created through an accept callback on a listening socket. One common design in multithreaded servers is to hand off the socket to a separate thread for the actual data transfer[1]. For internal implementation reasons however, a server socket cannot be transferred to another thread from within the accept callback. In other words, the following accept callback will not work.

[1] This design is far less common in Tcl applications thanks to its strong support for async event-driven I/O

```
proc accept {chan remote_addr port} {
    set tid [thread::create]
    thread::transfer $tid $chan
}
```

The workaround is to reschedule the transfer of the channel through the event loop as shown below.

```
proc accept {chan remote_addr port} {
    after 0 [list transfer_socket $chan $remote_addr $port]
}
proc transfer_socket {chan remote_addr port} {
    set tid [thread::create]
    thread::transfer $tid $chan
}
```

This quirk originates from additional references held internally on the channel during the processing of the accept callback. These references prevent the channel from being transferred. Once the callback returns, these references are released allowing the transfer to take place in the rescheduled procedure.

22.9. Introspecting threads: thread::id, names, exists

All commands in the Thread package for introspecting threads only deal with threads created with the thread::create command. Threads that have been created by other means, such as at C level, are not considered by these commands.

The thread::id command returns the id of the current thread.

```
thread::id → tid0000000000002FE8
```

The thread::names command returns the list of currently executing threads.

```
% thread::names
→ tid0000000000000FF4 tid0000000000002FE8
```

You can check for the existence of a thread with the thread::exists command.

```
set tid [thread::create -joinable] → tid00000000000020E8
thread::exists $tid                → 1
thread::release $tid               → 0
thread::join $tid                  → 0
thread::exists $tid                → 0
```

22.10. Thread-shared variables

Variables in a Tcl interpreter are always contained within that interpreter. They are not directly accessible from other threads or even other interpreters within the same thread. Data is shared between threads by passing **values** through messages. There are times however where it is beneficial to have a shared **location** where data is stored and accessed from multiple threads:

- Situations where the same exact values for data items must be seen from multiple threads is greatly simplified if the data is stored in a single location and not passed around.
- When large amounts of data is being shared, passing it around by value entails memory copies with a significant cost in performance.

For these situations, the Thread package provides the *thread-shared variables*, or simply shared variables, and a set of commands for manipulating them under the tsv namespace.

Thread-shared variables have the following characteristics:

- They are not “real” Tcl variables in that they cannot be referenced using `$` or the `set` command, they are not traceable and so on. They can only be manipulated through the `tsv` commands implemented by the `Thread` package.
- Shared variables exist outside of any interpreter or thread in that they continue to exist irrespective of the creating thread or any other thread terminating. When no longer required they have to be explicitly unset so as to free up memory resources.
- A shared variable is not a scalar but a collection of values indexed by a key similar to Tcl arrays and dictionaries.
- Access to these variables is always protected under a lock so threads do not need to explicitly synchronize when manipulating these variables. Note however that each variable has an independent lock so if a transaction involves multiple shared variables, additional locking may be required.

Commands for manipulating shared variables can be categorized as follows:

- Scalar operations such as setting, incrementing etc. on an individual element of a shared variable
- List operations similar to `lappend`, `lsearch` etc. on an individual element of a shared variable
- Array commands, similar to the standard Tcl `array` command, that operate on the entire shared variable, not just individual elements.
- Commands that treat the shared variable as a set of *keyed lists*, similar in utility to Tcl dictionaries
- Miscellaneous commands for introspection and utility functions

22.10.1. Scalar operations on shared variables

The commands `tsv::set`, `tsv::unset`, `tsv::incr`, `tsv::append` and `tsv::exists` work very similar to their standard Tcl counterparts `set`, `unset`, `incr`, `append` and `info exists` except that they operate on individual elements of a thread-shared variable rather than a Tcl variable.

```
% set tid [thread::create]
→ tid0000000000001A18
% tsv::exists mytsv myelem ❶
→ 0
% tsv::set mytsv myelem "A value" ❷
→ A value
% tsv::exists mytsv myelem
→ 1
% thread::send $tid {
    tsv::append mytsv myelem " in" " shared storage" ❸
}
→ A value in shared storage
% tsv::set mytsv myelem ❹
→ A value in shared storage
% tsv::set mytsv myinteger 0
→ 0
% tsv::incr mytsv myinteger 5 ❺
→ 5
```

❶ Check existence of `myelem` in the `mytsv` shared variable
❷ Set the value of the `myelem` element in the shared variable `mytsv`
❸ Append a string to the element from another thread
❹ Retrieve its value
❺ Increment the value of the `myinteger` element of the shared variable `mytsv`

There are additional commands that have no direct counterparts in the core Tcl command set. These are required because of the potential for race conditions between threads, something which is not an issue for normal variables. For example, consider the following code fragment:

```
if {[tsv::exists mytsv myelem]} {
    puts "Value is [tsv::set mytsv myelem]"
}
→ Value is A value in shared storage
```

The above seems to work but there is a hidden race condition. Although individual `tsv` commands operate under a lock protecting access to the shared variable, other threads may still change the variable between command invocations. For example, in the above code fragment, some other thread could unset the shared variable at some point between the `tsv::exists` check and the `tsv::set` invocation resulting in an exception.

The commands, `tsv::get`, `tsv::move` and `tsv::pop`, are provided to deal with common cases of this type where multiple base operations have to be performed atomically. The `tsv::get` command is meant for exactly the case described above.

```
tsv::get TSVAR ELEM ?VARNAME?
```

The command **atomically** checks for the existence of the element *ELEM* in the shared variable *TSVAR* and returns its value if it exists. If the *VARNAME* argument is provided, the command returns 1 if the element exists and 0 otherwise. The element value is stored in the variable *VARNAME*. Our previous (incorrect) code fragment can then be written as

```
if {[tsv::get mytsv myelem val]} {
    puts "Value is $val"
}
→ Value is A value in shared storage
```

If the *VARNAME* argument is not specified, the command returns the value of the shared variable element if it exists and raises an error otherwise.

The `tsv::move` command renames an element.

```
tsv::move mytsv myinteger yourinteger → (empty)
tsv::exists mytsv myinteger           → 0
tsv::set mytsv yourinteger            → 5
```

The `tsv::pop` command retrieves the value of an element while simultaneously removing it from the shared variable.

```
tsv::pop mytsv yourinteger    → 5
tsv::exists mytsv yourinteger → 0
```

22.10.2. List operations

The next set of commands also operate on individual elements of a shared variable but in this case the value stored in the element is treated as a list. Again, the majority of these commands parallel the standard Tcl commands of the same name for operating on lists. These are `tsv::lappend`, `tsv::linsert`, `tsv::lreplace`, `tsv::llength`, `tsv::lindex`, `tsv::lrange`, `tsv::lset` and `tsv::lsearch`. While some list modification commands operate on **variables** and others on list **values**, all `tsv` list modification commands operate on shared variable elements, not values (the latter would not make sense).

```
tsv::lappend mytsv mylist C D E                      → C D E ❶
thread::send $tid {
    tsv::linsert mytsv mylist 0 A B                  ❷
    tsv::lreplace mytsv mylist end-1 end X [list Y Z] ❸
    tsv::lset mytsv mylist end 1 P                   ❹
}                                                    → A B C X {Y P}
tsv::lindex mytsv mylist 2                           → C ❺
tsv::lrange mytsv mylist 1 3                         → B C X ❻
tsv::lsearch mytsv mylist -exact Y                   → -1 ❼
```

❶ Create a list
❷ Insert list elements from another thread
❸ Replace elements
❹ Set nested list element
❺ Retrieve a single list element
❻ Retrieve a range of list elements
❼ Search for exact match. Other options are `-glob` and `-regexp`

Just as in the case of scalar operations, shared variable lists also have commands for common multi-step sequences that need to be atomic. The `tsv::lpop` command is similar to `tsv::lindex` but in addition to returning the value at the specified index position, it also removes it from the shared variable element.

```
tsv::get mytsv mylist    → A B C X {Y P}
tsv::lpop mytsv mylist 2 → C
tsv::get mytsv mylist    → A B X {Y P}
tsv::lpop mytsv mylist   → A ❶
tsv::get mytsv mylist    → B X {Y P}
```

❶ If index is not specified, it defaults to 0.

The `tsv::lpush` command does the reverse operation.

```
tsv::lpush mytsv mylist J 3 → (empty)
tsv::get mytsv mylist       → B X {Y P} J
tsv::lpush mytsv mylist K   → (empty) ❶
tsv::get mytsv mylist       → K B X {Y P} J
```

❶ If index is not specified, it defaults to 0.

The `tsv::lpush` command always returns the empty string as its result.

22.10.3. Array operations

We have seen `tsv` commands that parallel the standard Tcl scalar and list operations. As you might expect, a similar set of commands exists that treat shared variables in a fashion similar to Tcl's array variables. Unlike the `tsv` commands previously discussed which operated on individual elements of a shared variable, these commands treat the shared variable itself as an array.

The commands related to `tsv` arrays are all implemented as the `tsv::array` ensemble. The ensemble subcommands `tsv::array set`, `tsv::array get`, `tsv::array names` and `tsv::array size` have the same function as their array command counterparts.

```
tsv::array set myarr {AB 1 BC 2 AC 3} → (empty) ❶
tsv::array get myarr                  → AC 3 BC 2 AB 1 ❷
tsv::array get myarr a*               → (empty) ❸
tsv::array size myarr                 → 3 ❹
```

❶ Create, set or modify elements
❷ Get all elements of an array
❸ Get all elements matching a pattern
❹ Get number of elements in the shared variable

The above commands operate on the shared variable as an array but it is still a shared variable and therefore can be accessed with the previously discussed commands. For example,

```
tsv::get myarr AB       → 1 ❶
tsv::set myarr XY 4     → 4 ❷
tsv::lappend myarr BC 5 → 2 5 ❸
tsv::array names myarr  → AC XY BC AB ❹
```

❶ Get the value of array key AB
❷ Creating a new element in the shared variable is equivalent to creating an element of the array
❸ List commands on an array element
❹ All element names. Notice it includes XY

Notice the symmetry between shared variables and Tcl arrays. A shared variable is analogous to a Tcl array where elements are indexed by a key. Just as elements of a Tcl array can be operated on by list and string commands, so can elements of a shared variable with the equivalent `tsv` commands.

There is one feature of thread shared variables that we do not discuss here. Shared variable arrays can be bound to persistent disk storage so that their contents are stored in a database. As of this writing the package supports the `gdbm` and `ldbm` databases. Depending on the platform, this feature may or may not be included. You can use the `tsv::handlers` command to list the available database backends. To use the feature, refer to `tsv::bind` in the `Thread` package documentation.

22.10.4. Keyed lists

One final basic Tcl structured data type whose analogue we have not described are dictionaries. Shared variables do not have any commands that work on dictionaries. However, they do have *keyed lists* which serve a similar purpose.

A keyed list is a list each element of which is itself a list with two elements, the first being the "key" and the second being the corresponding "value". For example, the keyed list

```
{Name {{First Sherlock} {Last Holmes}}} {Address {221B Baker Street}}
```

has two keys `Name` and `Address`. Moreover, the value associated with `Name` is itself a nested key list with keys `First` and `Last`.

The `tsv` commands related to keyed lists operate on the assumption that the element of the shared variable is a keyed list. The `tsv::keylset` command sets the values of elements in a keyed list.

```
tsv::keylset detectives holmes Name {{First Sherlock} {Last Holmes}}
tsv::keylset detectives holmes Address {221B Baker Street}
tsv::keylset detectives poirot Name {{First Hercule} {Last Poirot}} \
    Address {Whitehaven Mansions}
```

This creates elements holmes and poirot in the thread shared variable detectives.

We can retrieve elements as usual with tsv::get.

```
% tsv::get detectives holmes
→ {Name {{First Sherlock} {Last Holmes}}} {Address {221B Baker Street}}
```

To operate on individual fields, we use the tsv keyed list commands. The tsv::keylset command we used for creation can also be used to modify or add new keys. Field values are retrieved with tsv::keylget.

```
tsv::keylget TSVAR ELEM KEY ?RETVAR?
```

If the *RETVAR* argument is not specified, the command returns the value associated with *KEY* in the *ELEM* element of the shared variable *TSVAR*. Nested keys are accessed using special syntax where each key level is separated by a period.

```
% tsv::keylget detectives holmes Address
→ 221B Baker Street
% tsv::keylget detectives holmes Name
→ {First Sherlock} {Last Holmes}
% tsv::keylget detectives holmes Name.First ❶
→ Sherlock
```

❶ Nested field

The command will raise an error if the key does not exist.

When the *RETVAR* argument is provided, the command returns 1 if the key exists in the keyed list and 0 otherwise. In the former case the corresponding value is placed in the variable *RETVAR*.

```
tsv::keylget detectives mason Address addr  → 0
tsv::keylget detectives poirot Address addr → 1
set addr                                    → Whitehaven Mansions
```

Deleting a key is accomplished with the tsv::keyldel command.

```
tsv::keyldel detectives poirot Address → (empty)
tsv::get detectives poirot             → {Name {{First Hercule} {Last Poirot}}}
```

The tsv::keylkeys command retrieves the list of keys.

```
tsv::keylkeys detectives holmes      → Name Address
tsv::keylkeys detectives holmes Name → First Last ❶
```

❶ Retrieve nested keys

22.10.5. Introspection and utility commands: thread::names, object, lock

The list of thread shared variables can be enumerated with the tsv::names command.

```
tsv::names      → myarr detectives mytsv ❶
tsv::names det* → detectives ❷
```

❶ All shared variables
❷ Shared variables matching a pattern

All shared variables should be deleted when no longer needed. The `tsv::names` command can be useful for cleaning up at an appropriate time.

```
foreach tsv [tsv::names my*] {
    tsv::unset $tsv
}
```

The `tsv::object` command is a convenience for working with a specific element in a shared variable. It creates an object command bound to a shared variable element with its methods redirected to the standard `tsv` commands. For example,

```
% set sherlock [tsv::object detectives holmes]
→ ::0000000004573C78
% $sherlock keylset Sidekick Watson
% $sherlock get
→ {Name {{First Sherlock} {Last Holmes}}} {Address {221B Baker Street}} {Sidekick Watson}
```

The created command is automatically deleted when the associated element is unset.

The `tsv::lock` command evaluates multiple commands while holding the internal lock associated with a shared variable.

```
tsv::lock TSVAR ARG ?ARG …?
```

The command obtains the internal lock to the shared variable *TSV* creating the variable if necessary. It then evaluates the script formed through the concatenation of the *ARG* arguments and then releases the internal lock.

The `tsv::pop` command described earlier could be implemented as the following procedure.

```
proc pop {tsv elem} {
    tsv::lock $tsv {
        set val [tsv::get $tsv $elem]
        tsv::unset $tsv $elem
    }
    return $val
}
```

The lock ensures retrieval of value followed by unsetting of the element as an atomic operation.

22.11. Synchronization and locking

Threading models in many languages entail sharing of data between threads making the use of synchronization primitives commonplace. On the other hand, threads in Tcl are generally written as message-passing constructs with limited need for these primitives. Nevertheless, there are situations where synchronized access is required for commonly shared resources which may be internal, such as the shared variables we described in Section 22.10, or external, such as a set of files. The `Thread` package therefore provides mutexes and condition variables for such synchronization purposes.

This section assumes you are familiar with locking and synchronization in multithreaded programs. We will not explain what mutexes and condition variables are and how they might be used. Our discussion is restricted to a description of the synchronization facilities commands available in Tcl.

22.11.1. Mutexes: thread::mutex, thread::rwmutex

The Thread package implements three types of mutexes that can be used to ensure mutual exclusion when accessing shared resources.

- An exclusive mutex can be locked at most once. A second attempt to lock it will block the thread attempting the lock, even if it is the same thread that is holding the lock. The second thread will be unblocked when the thread holding the lock releases it. Note that a second attempt to lock by the thread already holding the lock will result in a deadlock. Operations on exclusive mutexes are implemented by the `thread::mutex` command ensemble.
- A recursive mutex only allows at most one thread to lock it but that thread can recursively lock it any number of times. Attempts by another thread to lock the mutex will result in that thread being blocked until the first thread releases **all** the locks it is holding on the mutex. Operations on recursive mutexes are also implemented by the `thread::mutex` command ensemble.
- A reader-writer mutex differs in that it supports two modes of locking — reader and writer. Multiple threads may simultaneously hold reader locks on the mutex. However at most one thread may hold a writer lock. Morever, a reader lock cannot be obtained while a writer lock is held and vice versa. Threads holding reader locks are allowed by contract to read the shared resource being protected but operations that modify the resource require a writer lock to be obtained. Operations on read-write mutexes are implemented by the `thread::rwmutex` command ensemble.

The `create` subcommand of both ensembles creates a mutex of the appropriate type. A recursive mutex requires the `-recursive` option to be specified to the `thread::mutex` command.

```
set exclusive_mutex [thread::mutex create]            → mid0
set recursive_mutex [thread::mutex create -recursive] → rid1
set rw_mutex [thread::rwmutex create]                 → wid2
```

Exclusive and recursive mutexes are locked with the `thread::mutex lock` command.

```
thread::mutex lock $exclusive_mutex → (empty)
thread::mutex lock $recursive_mutex → (empty)
```

The locks are released with `thread::mutex unlock`.

```
thread::mutex unlock $exclusive_mutex → (empty)
thread::mutex unlock $recursive_mutex → (empty)
```

However for reader-writer mutexes, two separate commands, `thread::rwmutex rlock` and `thread::rwmutex wlock` are required corresponding to reader and writer locks.

```
thread::rwmutex rlock $rw_mutex  → (empty) ❶
thread::rwmutex unlock $rw_mutex → (empty)
thread::rwmutex wlock $rw_mutex  → (empty) ❷
thread::rwmutex unlock $rw_mutex → (empty)
```

❶ Obtain a reader lock
❷ Obtain a writer lock

All mutexes should be destroyed when no longer required with `thread::mutex destroy` or `thread::rwmutex destroy` depending on the type of mutex.

```
thread::mutex destroy $exclusive_mutex → (empty)
thread::mutex destroy $recursive_mutex → (empty)
thread::rwmutex destroy $rw_mutex      → (empty)
```

You must ensure there are no outstanding locks on a mutex before destroying it. Otherwise, undefined behaviour including process crashes may result depending on the platform.

Here is a trivial example of the use of mutexes. As we stated earlier, standard channels are available in all threads. If we want to ensure output from multiple threads is not intermixed, we can use a mutex for the purpose.

```
% set io_mutex [thread::mutex create]
→ mid3
```

Then threads have to **by convention** lock the mutex before doing I/O.

```
% thread::mutex lock $io_mutex
% puts "Line one"
→ Line one
% puts "Line two"
→ Line two
% thread::mutex unlock $io_mutex
```

There is one issue with the above pattern when more complex code is involved. If an exception is raised in the script, the mutex will stay locked. Safer practice is to enclose the block within a `try` or `catch` command which will release the lock even in error cases. This situation is common enough that the `Thread` package provides the `thread::eval` command as a convenience.

```
thread::eval ?-lock MUTEX? ARG ?ARG …?
```

The command locks the specified mutex, or an internal global mutex if the `-lock` option is not provided. The command then executes the script formed from concatenating the *ARG* arguments and returns its result taking care to unlock the mutex even under exceptional returns.

Our example above could then be written with better error handling as

```
thread::eval -lock $io_mutex {
    puts "Line one"
    puts "Line two"
}
```

For an equally superficial illustration of reader-writer mutexes, assume we are keeping bank account information in shared variables `current` and `savings` whose elements are keyed by an account number. To ensure consistent data, we have to ensure we do not read in the middle of a transaction that modifies the data. However, there is no reason to disallow multiple readers so we can use a reader-writer lock. Our desire for a reader-writer lock precludes us from using `thread::eval`.

```
proc bank::init {} {
    variable acct_mutex [thread::rwmutex create]
    ...create and initialize account data...
}

proc transfer {from_type from_acct to_type to_acct amount} {
    variable acct_mutex
    thread::rwmutex wlock $acct_mutex
    try {
        tsv::incr $from_type $from_acct -$amount
        tsv::incr $to_type $to_acct $amount
    } finally {
        thread::rwmutex unlock $acct_mutex
```

```
    }
}

proc balance {acct_type acct} {
    variable acct_mutex
    thread::rwmutex rlock $acct_mutex
    try {
        set balance [tsv::get $acct_type $acct]
    } finally {
        thread::rwmutex unlock $acct_mutex
    }
    return $balance
}
```

22.11.2. Condition variables: thread::cond

Condition variables are used in conjunction with exclusive mutexes as a race-free mechanism for a thread to block until some predicate is true. The condition variable is used for notification while the mutex is used to ensure that the data underlying the predicate is not modified by another thread while being checked.

The general usage pattern is as follows. The initialization code

1. Creates an **exclusive** mutex
2. Creates a condition variable

Each waiting thread (there may be multiple of these) runs the following sequence of steps:

1. Lock the mutex. Since the mutex is held, the predicate can be safely checked in the next step without fear of interference from other threads.
2. Check the predicate. If the predicate evaluates to true, go to step 5. Otherwise go to the next step.
3. Block on the condition variable. Blocking on the condition variable also releases the held mutex lock allowing other threads to modify the protected predicate data.
4. When the blocking call returns, go back to Step 2. See the explanation below as to why. Note that the mutex is held when the blocking call returns.
5. Do the expected work. Note the mutex is held at this point.
6. Unlock the mutex.

Steps 3 and 4 warrant some additional explanation. The blocking call on the condition variable also internally releases the mutex. This allows another thread to modify the protected data possibly changing the predicate to evaluate to true. If so the thread will signal the condition variable (see below) thereby waking up the thread waiting on the condition variable. Before returning in Step 3, the woken thread also relocks the mutex thereby ensuring the predicate is again protected. **However between the time the thread was signalled and the time the mutex was relocked, the predicate might have changed again.** This can happen for any number of reasons. For example, in a worker thread model, multiple threads may wait on a condition variable in which all of them are woken up on the signal. The first one to succesfully grab the mutex may modify the predicate data, for example by removing a work item from a queue. When other worker threads lock the mutex, the predicate is then no longer true. For this reason, after a thread wakes up it needs to recheck the predicate as indicated by Step 4.

The sequence for the signalling threads is simpler.

1. Lock the mutex.
2. Modify the protected data.
3. If the predicate evaluates to true, signal the condition variable to wake up the waiting thread(s).
4. Unlock the mutex.

We will illustrate the use of condition variables by extending our previous examples to implement a worker thread model for fetching URLs. Our main thread will queue URL requests to a shared queue. A set of worker threads will pick up entries from the queue, fetch the URL and write the content to a file.

First we initialize the data — the URL queue, the condition variable and mutex — that will be shared between the threads. We have already described mutex creation but make a note that this must be an exclusive mutex. The condition variables is created with the `thread::cond create` command. The returned handles are stored in shared variables since they will need to be available to the worker threads as well.

```
tsv::set shared_data url {}       → (empty)
set mutex [thread::mutex create]  → mid4
tsv::set shared_data mutex $mutex → mid4
set cond [thread::cond create]    → cid5
tsv::set shared_data cond $cond   → cid5
```

Next we define the script that each worker thread will run.

```
set worker_script {
    package require http
    package require fileutil
    proc url_to_file {url} {
        return [file join \
                    [fileutil::tempdir] \
                    [file rootname [file tail $url]]_content.html]
    }
    set mutex [tsv::get shared_data mutex]
    set cond  [tsv::get shared_data cond]
    while {1} {
        thread::mutex lock $mutex
        while {[tsv::llength shared_data urls] == 0} {
            thread::cond wait $cond $mutex
        }
        set url [tsv::lpop shared_data urls]
        thread::mutex unlock $mutex
        set tok [http::geturl $url]
        fileutil::writeFile [url_to_file $url] [http::data $tok]
        http::cleanup $tok
    }
}
```

The script initializes the thread by loading the omnipresent `http` package and retrieving the mutex and condition variable handles from shared memory. It then sits in an infinite loop waiting for work to be queued. Within each iteration of the loop, it follows the general pattern we described earlier. The predicate in this case is for the URL queue to have at least one entry in it. If the queue is empty, it waits on the condition variable with the `thread::cond wait` command.

```
thread::cond wait COND MUTEX ?TIMEOUT?
```

This command will unlock the specified mutex and then suspend execution until the condition variable *`COND`* is signalled or the optional timeout occurs. The timeout value *`TIMEOUT`* is specified in milliseconds. If unspecified or 0, the command will only return when the condition variable is signalled.

The timeout value has some caveats associated with it. It controls how long the command will internally wait for the condition to be signalled. However, even after being awoken, the `wait` command semantics call for the mutex to be re-acquired. There is no control over how long this might take if there are many threads competing for the mutex.

Before returning to the caller, the `wait` command re-locks the mutex. Our worker thread is then free to check the status of the URL queue without suffering race conditions from other threads. If the queue is empty (some other worker beat us to it), the code loops back into the conditional wait. If the queue is not empty, the thread removes one entry from it. It then unlocks the mutex to let other threads process additional entries if any, and proceeds to download the dequeued URL. The whole sequence is then repeated.

The work dispatcher is even simpler following our aforementioned pattern. We first start up a suitable number of worker threads, say 4, to run the worker thread script we showed earlier.

```
for {set i 0} {$i < 4} {incr i} {
    lappend workers [thread::create $worker_script]
}
```

We can then queue up URL's to be fetched in the background at any time with a sequence of calls similar to the following.

```
thread::mutex lock $mutex
tsv::lpush shared_data urls http://www.example.com/page.html end
thread::cond notify $cond
thread::mutex unlock $mutex
```

The only new command in this sequence is `thread::cond notify`. This command signals the specified condition variable causing all threads waiting on it to resume execution. The first thread to run and acquire the mutex will then dequeue and download the URL.

One final point about condition variables. They should be freed when no longer required by calling `thread::cond destroy`.

```
thread::cond destroy COND
```

As with mutexes, care must be taken that the condition variable is not in use when the command is invoked.

22.12. Thread pools

In the previous section we used a minimal, incomplete and custom implementation of the commonly used worker thread model. The `Thread` package includes a generalized version of this functionality through its *thread pools*. A thread pool consists of a work queue where *jobs* can be posted and an associated set of worker threads that pull them off the queue and execute them. An application may create any number of such thread pools.

Each thread pool has a variable number of threads that can be configured to run in it with minimum and maximum limits. The pool starts out with the minimum number when it is created. If the number of active or queued jobs exceeds the current number of threads, new threads are created to handle the additional jobs until the maximum thread limit is reached. Beyond that point jobs remain queued until they are removed by some thread that has completed its previous job. If a thread finds there are no jobs remaining to be processed, it goes into an idle state. If it stays in that state for some period of time without getting a new job, it will exit provided the current number of threads is greater than the minimum limit.

All commands related to thread pools are contained in the `tpool` namespace.

22.12.1. Creating a thread pool: `tpool::create`

A thread pool is created with `tpool::create`.

```
tpool::create ?OPTIONS?
```

The command returns the id of the thread pool which can be used to post jobs to the work queue for the pool. The options shown in Table 22.2 may be specified to control various associated configuration parameters.

Table 22.2. Thread pool options

Option	Description
`-exitcmd` *SCRIPT*	Specifies a script to be run just before the thread exits. A thread will exit if it has been idle for some period of time.
`-idletime` *SECONDS*	Specifies the number of seconds that a thread has to remain idle before it exits as described above.
`-initcmd` *SCRIPT*	Specifies a script to be run in the worker thread when it first starts up. This can be used to load packages, initialize data and so on. If the script raises an error in a worker thread, the exception is reflected back into `tpool::create` or `tpool::post` depending on which call initiated the creation of the thread.
`-maxworkers` *COUNT*	The maximum number of worker threads that should be maintained in the thread pool. The default is 4,
`-minworkers` *COUNT*	The minimum number of worker threads that should be maintained in the thread pool. The default is 0 so that the pool starts with 0 threads until the first job is queued.

Any number of thread pools may be created. The command `tpool::names` returns the list of currently existing thread pools.

Let us create a thread pool version of the example in the last section with one difference. It returns the content of the URL back to the main thread instead of writing it to a file. This provides a more complete example as it shows communication in both directions and some rudimentary error handling.

First we define the initialization script each worker thread needs to run. This just loads the `http` package and defines a procedure to retrieve URL's.

```
set init_script {
    package require http
    proc fetch_url {url} {
        set tok [http::geturl $url]
        try {
            switch -exact -- [http::status $tok] {
                ok { return [http::data $tok] }
                eof { error "Server closed connection." }
                error { error [http::error $tok] }
                default { error "Unknown error retrieving $url" }
            }
        } finally {
            http::cleanup $tok
        }
    }
}
```

Creating the thread pool is straightforward. We will use the defaults for all options except `-initcmd` since we need to pass in the above script.

```
set tpool [tpool::create -initcmd $init_script] → tpool00000000041564D0
tpool::names                                    → tpool00000000041564D0 ❶
```

❶ Lists all thread pools

Just as for threads, we will mark the thread pool as "in-use". This is discussed further in Section 22.12.7.

```
tpool::preserve $tpool → 1
```

22.12.2. Posting jobs to a thread pool: tpool::post, tpool::get

Any thread can post a *job* to a thread pool with the tpool::post command. A job is nothing but a Tcl script to executed by a worker thread just as we referred to scripts sent to specific threads as messages. The difference in terminology is conceptual.

```
tpool::post ?-detached? ?-nowait? TPOOL SCRIPT
```

The result of the command is a *job identifier* unless the -detached option, described below, is specified. The command initiates evaluation of *SCRIPT* in a worker thread in the following manner:

- If an idle thread is available in the thread pool, the *SCRIPT* is passed to it for execution and the command returns.
- If the -nowait option is specified, the command places the job on the work queue and returns.
- If no threads are idle and the maximum thread count limit has not been reached for the thread pool, a new thread is created. The tpool::post command will then wait in the event loop of the current thread processing events until the new thread is initialized. It will then pass the script to the new thread and return.
- If no threads are idle and the thread limit is reached the command will wait for a thread to become idle. In this case as well, while waiting the current thread's event loop will continue processing events.

The -detached option provides a "fire and forget" capability. If specified, the command creates a detached job which cannot be canceled or waited on, and whose result cannot be obtained. In this case the command returns an empty string.

Make note of one difference between using the thread::send command to send messages to a specific thread and the tpool::post command. The script passed in the tpool::post command may be processed by **any** thread in the pool. Therefore do not rely on context to be maintained between two tpool::post calls. Thus the following sequence

```
tpool::post $tpool {set x 1}
tpool::post $tpool {puts $x}
```

is likely to fail with an undefined variable error since the second script may not be executed by the same thread as the first.

Let us fetch two URL with our previously created thread pool. The second URL is invalid just so we can demonstrate error handling in fetching results.

```
set job [tpool::post $tpool {fetch_url http://www.example.com}]    → 1
set bad_job [tpool::post $tpool {fetch_url xyz://www.example.com}] → 2
```

Now that the jobs have been posted, we need to know when they complete so we can retrieve the results.

22.12.3. Waiting for job completion: tpool::wait

The tpool::wait command is used to wait for the completion of one or more posted jobs.

```
tpool::wait TPOOL JOBLIST ?VARNAME?
```

Here *JOBLIST* is a list of job identifiers as returned by the `tpool::post` command. The command enters the current thread's event loop processing events until at least one of the jobs in *JOBLIST* has completed. It then returns the list of completed job identifiers (there may be more than one). If the optional *VARNAME* argument is provided, the command stores the identifier for jobs still pending into the variable of that name.

We can wait for our previously posted jobs to complete with the following loop.

```
set jobs [list $job $bad_job]
while {[llength $jobs]} {
    tpool::wait $tpool $jobs jobs
}
```

22.12.4. Retrieving a job result: `tpool::get`

Once a job is completed, its result can be retrieved with the `tpool::get` command.

```
tpool::get TPOOL JOBID
```

The result of the job is the result of the script evaluation in the worker thread. We can get the content of our URL in our example.

```
% tpool::get $tpool $job
→ <!doctype html>
  <html>
  <head>
      <title>Example Domain</title>

 ...Additional lines omitted...
```

If the script threw an error, the `tpool::get` command will also throw an error with the same `errorCode` and `errorInfo` values set by the original error. The command will also throw an error on an attempt to retrieve the result of a job that has not completed yet.

We provided one URL above that was invalid. Accordingly, an attempt to retrieve the result will raise an error.

```
% tpool::get $tpool $bad_job
Ø Unsupported URL type "xyz"
% set errorInfo
→ Unsupported URL type "xyz"
      while executing
  "http::geturl $url"
      (procedure "fetch_url" line 2)
      invoked from within
 ...Additional lines omitted...
```

Note how the error message and stack are those raised by the worker thread.

22.12.5. Canceling a job: `tpool::cancel`

Jobs on the work queue that are still pending **and not yet assigned to worker threads** can be cancelled with the `tpool::cancel` command.

```
tpool::cancel TPOOL JOBLIST ?VARNAME?
```

Here *JOBLIST* is the list of identifiers for the jobs to be cancelled. The command returns the list of identifiers for the cancelled jobs. If the *VARNAME* argument is provided, the list of jobs that could not be cancelled is stored in a variable of that name.

22.12.6. Suspending thread pools: tpool::suspend, tpool::resume

A thread pool can be suspended at any time by calling the tpool::suspend command.

```
tpool::suspend TPOOL
```

Suspending a thread pool will suspend execution of all threads in the pool. However, a caller may still post jobs to the pool's work queue.

The suspension may be rescinded with the tpool::resume command.

```
tpool::resume TPOOL
```

The command will then cause the worker threads to resume execution. Idle workers will pick up any additional jobs placed on the work queue while the thread pool was suspended. However, no new threads are created to handle additional pending jobs **even if the maximum thread count limit has not been reached**.

22.12.7. Thread pool lifetimes: tpool::preserve, tpool::release

Just as for threads, thread pool lifetimes need to be managed. The reasons are much the same. There may be multiple users of a thread pool and they do not want the pool to be disappearing from under them while in use. The solution provided is also much the same. A reference count is associated with each thread pool. The reference count is incremented with tpool::preserve and decremented with tpool::release. The pool is freed when the reference count drops to 0. Beyond that point, attempts to post a job to the pool will result in a failure. However, this does not necessarily mean the threads in the pool have exited. They will do so on their own schedule.

Since we done with our URL fetching thread pool, we can inform the package accordingly.

```
tpool::release $tpool → 0
```

22.13. Distributing interpreter state: the Ttrace package

A common need when working with multiple threads is to ensure that they are all running in the same runtime "environment" in terms of namespaces, procedure definitions and the like. The Ttrace package partially satisfies this need. It allows for procedure and namespace definitions to be replicated across all threads.

Because this is a separate package and not part of Thread, we have to load it separately.

```
package require Ttrace → 2.8.0
```

Although the package contains many commands, we only describe one, ttrace::eval, which encapsulates the other commands into a simple-to-use interface. The other commands allow for finer grain control but we do not describe them here.

```
ttrace::eval ARG ?ARG …?
```

The ttrace::eval command evaluates the script formed from concatenation of it arguments. Any changes in procedure or namespace definitions are then propagated to all threads.

The functionality is most easily illustrated with an example. Let us start up a thread. All threads making use of this feature must load the package as well.

```
set tid [thread::create {
    package require Ttrace
    thread::wait
}]
→ tid0000000000002730
```

Obviously this thread will not have the namespace `ns` or the `ping` procedure defined.

```
% set tid [thread::create]
→ tid0000000000002AAC
% thread::send $tid {namespace exists ns}
→ 0
% thread::send $tid ping
Ø invalid command name "ping"
```

Now we define both the namespace and the procedure within the **current** thread but do so within a `ttrace::eval`.

```
ttrace::eval {
    namespace eval ns {}
    proc ping {} {return "Ping from [thread::id]"}
}
```

We can then verify that the namespace and procedure have been replicated in the current thread **and** in the other thread.

```
namespace exists ns                     → 1
ping                                    → Ping from tid0000000000002FE8
thread::send $tid {namespace exists ns} → 1
thread::send $tid ping                  → Ping from tid0000000000002AAC
```

If a new thread is created, that will also automatically have these definitions.

```
% set tid2 [thread::create {
    package require Ttrace
    thread::wait
}]
→ tid000000000000174C
% thread::send $tid2 {namespace exists ns}
→ 1
% thread::send $tid2 ping
→ Ping from tid000000000000174C
```

You can see how the `Ttrace` package simplifies keeping threads in sync in terms of the namespace and procedure definitions. However, there are some important limitations to keep in mind. In particular, data and TclOO classes and objects are not replicated.

22.14. Using extensions in threads

Generally speaking, packages that are purely script-based do not require any special consideration for use in threads. However, use of binary extensions in a Tcl application that uses multiple threads takes some care. These fall into three categories:

- Extensions that are written to be thread-safe and can be safely loaded and used in multiple threads.
- Extensions that are **not** thread-safe but can be safely loaded and used in a **single** thread within a multithreaded applications. If other threads want to make use of the extension functionality, they need to do so by sending messages to the thread in which the extension is loaded. The Tk extension is one such example.

- Extensions that not thread-safe and should not be used in a threaded Tcl application.

Refer to the documentation for each extension to determine its thread safety characteristics.

22.15. Comparing coroutines and threads

Having worked through both coroutines and threads, you can see that at some level they serve a similar purpose in enabling computational tasks to proceed in concurrent fashion. We now summarise their differences to help you choose the one appropriate for the problem you are trying to solve.

The primary differences between coroutines and threads arise from the fact that threads are operating system level constructs whereas coroutines are implemented purely in user mode.

- Multiple threads can be assigned to multiple processors leading to increased performance. Coroutines run within a single thread and therefore are limited to the processor on which the thread is running. They derive no benefit from the presence of multiple processors. Of course, there is no reason not to run coroutines within multiple threads but that is a different kettle of fish as coroutines can only communicate within the interpreter where they are defined and would have to use the thread mechanisms for anything else.
- Blocking operations in a thread do not block other threads. In contrast, a blocking operation in a coroutine will block other coroutines as well. This is often a primary consideration in the decision to move to a thread-based architecture. Certain operations, for example accessing some database implementations, are only implemented in blocking form by the database drivers. In such cases, moving those operations to a separate thread ensures other parts of an application continue to run while a long database operation is in progress.
- Coroutines are defined within a single interpreter and share its namespaces, commands, channels etc. There is limited isolation between coroutines. Threads on the other hand enclose independent interpreters. There is no sharing of any kind except through the threading commands discussed earlier. This means errors in one thread can be isolated to that thread.
- Threads are relatively expensive to create and consume significant system resources like memory while coroutines are cheap in that respect. Inter-thread communication is also slower as it forces operating system context switches.
- Partly because they are cheap and share code and data, coroutines can be used for implementing generators and the like. Threads are not suitable for such purposes.

Having contrasted the two, keep in mind that coroutines and threads are not mutually exclusive. It is common and perfectly reasonable to use both in an application architecture.

The article Modeling a Queuing System compares and contrasts multiple implementations of a queueing system model that include threads and coroutines in addition to other mechanisms.

22.16. Chapter summary

In this chapter, we covered the use of operating system threading capabilities from within Tcl. Multithreading permits the application to take advantage of multiple processors in the system. However, its use must be carefully considered as it adds significant complexity to an application. In some languages, multithreading is required for concurrent I/O; however, this is not the case in Tcl where the asynchronous I/O model through the event loop is not only adequate but often more efficient. Similarly, in cases where the purpose of multithreading is to simplify programming of concurrent independent tasks while writing in a “sequential” style, coroutines can fulfil the need at a cheaper cost.

Nevertheless there are instances where only threads meet the desired needs and Tcl provides for that possibility. In conjunction with Tcl’s event loop and coroutines, practically any software architecture geared towards multitasking and concurrent computation can be implemented in Tcl.

22.17. References

MARK2015

Modeling a Queuing System, Arjen Markus, http://www.magicsplat.com/articles/queueing.html. Compares thread and coroutine-based implementations of a queueing system model.