The following paper was originally published in the
Proceedings of the Fifth Annual Tcl/Tk Workshop
Boston, Massachusetts, July 1997

PtTcl: Using Tcl with Pthreads

D. Richard Hipp
Hwaci, Charlotte, NC
Mike Cruse
CTI, Ltd., Prescott, AZ

PtTcl: Using Tcl with Pthreads

D. Richard Hipp

Hwaci
6200 Maple Cove Lane
Charlotte, NC 28269

Mike Cruse

CTI, Ltd.
1040 Whipple St. #225
Prescott, AZ 86301

Abstract

Tcl is not thread-safe. If two or more threads attempt to use Tcl at the same time, internal data structures can be corrupted and the program can crash. This is true even if the threads are using separate Tcl interpreters.

PtTcl is a modification to the Tcl core that makes Tcl safe to use with POSIX threads. With PtTcl, each thread can create and use its own Tcl interpreters that will not interfere with interpreters used in other threads. A message-passing mechanism allows Tcl interpreters running in different treads to communicate. However, even with PtTcl, the same interpreter still cannot be accessed by more than one thread.

This paper describes the design, implementation and use of PtTcl.

1 Introduction

Tcl was originally designed for use in single-threaded programs only. But recently, there has been an increasing need to use the power of Tcl in applications that contain multiple threads of control. Unfortunately, the core Tcl library uses several static data structures that can become corrupted if accessed simultaneously by two or more threads. A program crash is the usual result.

There is a least one prior effort to make Tcl thread-safe. Steve Jankowski created a modified version of the Tcl sources called MTtcl [MtTcl] which allows the use of Tcl in a multi-threaded environment. But Jankowski's implementation only works with Solaris threads and on Tcl versions 7.4 and earlier.

This article describes a new implementation of multi-threaded Tcl that is based on POSIX threads [Pthreads] and works with Tcl version 7.6. (An upgrade to Tcl version 8.0 is planned.) We call the package "PtTcl". PtTcl borrows some of Jankowski's ideas but is a completely new implementation.

2 Threading Model

The usual model for a multi-threaded program is that each thread has its own stack used to store subroutine return addresses and local variables. In a compiled program, the hardware in cooperation with the operating system and thread library take care of providing and managing these separate stacks. But in an interpreted language like Tcl, the interpreter must create and manage the separate stacks itself. The standard Tcl interpreter only makes provisions for a single stack. In order to make Tcl truly multi-threaded, it is necessary to change the Tcl core to allow multiple stacks per interpreter.

Unfortunately, the concept of one stack per interpreter is a fundamental assumption in the design of Tcl, and to change this assumption would require a major rewrite of many key routines. In order to avoid excessive rework of the Tcl core, we chose to use a more restrictive thread model for PtTcl.

PtTcl allows an application to have multiple Tcl interpreters running in independent threads, and each thread in a PtTcl program can contain any number of interpreters (including zero). But PtTcl only allows an interpreter to be run from a single thread. If another thread tries to use an interpreter, an error message is returned.

In ordinary Tcl, there is a single event queue used to process all timer and file events. In PtTcl, this concept is extended to one event queue per thread. The fact that each thread has its own event queue is a necessary consequence of the restriction that Tcl interpreters must always be run in the same thread. Recall that the usual action taken when an event arrives is for a Tcl script to run in response. Suppose an interpreter in thread A registers to receive an event, but the event arrives while executing thread B. There is no way for the receiving interpreter, running in thread B, to execute the desired script because that script can only be run from thread A. Hence, if we want to be able to invoke scripts in response to events, each thread must have its own event queue.

Each thread has the concept of a *main interpreter*. The main interpreter is different from other interpreters in the same thread in only one way: you can send messages to the main interpreter.

Messages are a new kind of Tcl event, so a Tcl interpreter running in a given thread will not process any messages until it visits its event loop. A Tcl interpreter visits its event loop whenever it executes one of the standard Tcl commands `vwait` or `update` or one of two commands added by PtTcl: `thread eventloop` and `thread send`.

Messages can be either synchronous (meaning they will wait for a response) or asynchronous (fire and forget). The result returned to the sending thread from a synchronous message is the result of the Tcl script in the receiving thread or an error message if the message could not be sent for some reason. An asynchronous message has no result usually, but it still might return an error if the message could not be sent. Asynchronous messages can be broadcast to all main interpreters or to all main interpreters except the interpreter that is doing the sending.

A message can be sent from any interpreter, not just the main interpreter, or directly from C code. There does not have to be a Tcl interpreter running in a thread in order for that thread to send a message, but a main interpreter is necessary for the message to be received.

Most variables used by a Tcl interpreter are private to that interpreter. But PtTcl implements a mechanism for sharing selected variables between two or more interpreters, even interpreters running in different threads. However, it is not possible to put trace events on shared variables, which limits their usefulness.

Here is a quick summary of the execution model used by PtTcl:

- A single thread can contain any number of Tcl interpreters.
- A particular Tcl interpreter may only be used from within a single thread.
- Each thread has its own event queue.
- A message (in the form of a Tcl script) can be sent to the main Tcl interpreter in any thread.
- Tcl variables may be shared between two or more Tcl interpreters, even interpreters running in separate threads.

3 New Tcl Commands

The PtTcl package implements two new Tcl commands. The "`shared`" command is used to designate variables that are to be shared with other interpreters, and the "`thread`" command is used to create and control threads.

3.1 The "`shared`" command

The `shared` is similar to the standard `global` command. `Shared` takes one or more arguments which are names of variables that are to be shared by all Tcl interpreters, including interpreters in other threads. Note that both interpreters must execute the `shared` command independently before they will really be using the same variable.

Unfortunately, the `trace` command will not work on shared variables. This is another consequence of the fact that an interpreter can only be used in a single thread. When a trace is set on a variable, a Tcl script is run whenever that variable is read, written or deleted. But, if the trace was set by thread A and the variable is changed by thread B, there is no way for thread B to invoke the trace script in thread A.

3.2 The "`thread`" command

The `thread` command is more complex than `shared`. `Thread` contains nine separate subcommands used to create new threads, send and receive messages, query the thread database, and so forth. Each is described separately below.

`thread self`

Every thread in PtTcl that contains an interpreter is assigned a unique positive integer Id. This Id is used by other thread commands to designate a message recipient or the target of a join. The `thread self` command returns the Id of the thread that executes the command.

`thread create` [*command*] [`-detach` *boolean*]

New threads can be created using the `thread create` command. The optional argument to this command is a Tcl script that is executed by the new thread. After the specified script is completed, the new thread exits. If no script is specified, the command "`thread eventloop`" is used instead. Assuming the new thread is created successfully, the `thread create` command returns the thread Id of the new thread.

After a thread finishes executing its Tcl script, it normally waits for another thread to join with it and takes its return value. (See the `thread join` command below.) But if the `-detach` option evaluates to true, then the thread will terminate immediately upon finishing its script. A detached thread can never be joined.

Note that the joining and detaching of threads is an abstraction implemented by the PtTcl library. From the point of view of the Pthreads library, all threads created by the `thread create` command run detached.

`thread send` *whom message* [`-async` *boolean*]

Use the `thread send` command to send a message from one thread to another. The arguments to this command specify the target thread and the message to be sent. The message is simply a Tcl script that is executed on the remote thread. The `thread send` command normally waits for the message to complete on the remote thread, then returns the result of the script. But, if the `-async` option is true, the `thread send` will return immediately, not waiting on a reply.

`thread broadcast` *message* [`-sendtoself` *boolean*]

The `thread broadcast` works like `thread send` except that it sends the message to all threads and it always operates asynchronously. Normally, it will not send the message to itself, unless you also specify the `-sendtoself` flag.

`thread update`

This command causes the current thread to process all pending messages, that is, messages that other threads have sent and are waiting for this thread to process. Only thread messages are processed by this command – other kinds of pending events are ignored. If you want to process all pending events including thread messages, use the `update` command from regular Tcl.

`thread eventloop`

This command causes the current thread to go into an infinite loop processing events including incoming messages. This command will not return until the interpreter is destroyed by either an `exit` command or a `interp destroy {}` command.

`thread join` [`-id` *Id*] [`-timeout` *milliseconds*]

The `thread join` command causes the current thread to join with another thread that has completed processing. The return value of this command is the result of the last command executed by the thread that was joined. By default, the first available thread is joined. But you can wait on a particular thread by using the `-id` option.

The calling thread will wait indefinitely for another thread to join unless you specify a timeout value. When a timeout is specified, the `thread join` will return after that timeout regardless of whether or not it has found another thread to join. A timeout of zero (0) can be used if you want to quickly see if any threads are waiting to be joined.

`thread list`

This command returns a list of Tcl thread Id numbers for each existing thread.

```
thread yield
```

Finally, the `thread yield` command causes the current thread to yield its timeslice to some other thread that is ready to run, if any.

4 New C Functions

In addition to the new Tcl commands, PtTcl also provides several new C functions that can be used by C or C++ programs to create and control Tcl interpreters in a multi-threaded environment.

```
int Tcl_ThreadCreate(
   char *cmdText,
   void (*initProc)(Tcl_Interp*,void*),
   void *argPtr
);
```

The `Tcl_ThreadCreate()` function creates a new thread and starts a Tcl interpreter running in that thread. The first argument is the text of a Tcl script that the Tcl interpreter running in the new thread will execute. You can specify `NULL` for this first argument and the Tcl interpreter will execute the command `thread eventloop`. The second argument to `Tcl_ThreadCreate()` is a pointer to a function that can be used to initialize the new Tcl interpreter before it tries to execute its script. The third argument is the second parameter to this initialization function. Either or both of these arguments can be `NULL`. All threads created by `Tcl_ThreadCreate()` are detached.

The `Tcl_ThreadCreate()` returns an integer which is the Tcl thread Id of the new thread it creates. This is exactly the same integer that would have been returned if the thread had been created using the `thread create` Tcl command.

The `Tcl_ThreadCreate()` may be called from a thread that does not itself have a Tcl interpreter. This function allows threads that do not use Tcl to create subthreads that do.

Note that the `(*initProc)()` function might not have executed in the new thread by the time `Tcl_ThreadCreate()` returns, so the calling function should not delete the `argPtr` right away. It is safer to let the `(*initProc)()` take responsibility for cleaning up `argPtr`.

```
int Tcl_ThreadSend(
   int toWhom,
   char **replyPtr,
   char *format,
   ...
);
```

The `Tcl_ThreadSend()` function allows C or C++ code to send a message to the main Tcl interpreter in another thread. The first argument is the Tcl thread Id number (not the pthread_t identifier) of the destination thread. You can specify a destination of zero (0) in order to broadcast a message.

The second parameter is used for the reply. The message response is written into memory obtained from `ckalloc()` and `**replyPtr` is made to point to this memory. If the value of the second parameter is `NULL`, then the message is sent asynchronously. If the first parameter is 0, then the second parameter must be `NULL` or else an error will be returned and no messages will be sent.

The third parameter is a format string in the style of `printf()` that specifies the message that is to be sent. Subsequent arguments are added as needed, exactly as with `printf()`.

The return value from `Tcl_ThreadSend()` is the return value of the call to `Tcl_Eval()` in the destination thread, if this is a synchronous message. For an asynchronous message, the return value is `TCL_OK` unless an error prevents the message from being sent.

```
Tcl_Interp *Tcl_GetThreadInterp(
   Tcl_Interp *interp
);
```

The `Tcl_GetThreadInterp` routine will return a pointer to the main interpreter for the calling thread. If the calling thread does not have a main interpreter, then the interpreter specified as its argument is made the main interpreter. If the argument is `NULL`, then `Tcl_CreateInterp()` is called to create a new Tcl interpreter which becomes the main interpreter. At the conclusion of this function, the calling thread is guaranteed to have a main interpreter and a pointer to that interpreter will be returned.

5 Changes Made To The Tcl Core

The biggest change in PtTcl is the implementation of separate event queues for each thread. In the standard Tcl distribution, the event queue is constructed as a linked list of structures with a static pointer to the head of the list. In PtTcl, we simply converted this static pointer into thread-specific data. Actually, once you get into the details, you find that more than this one static pointer can cause problems. Dozens of static variables in Tcl had to be converted into thread-specific data variables. And, of course, mutexes had to be added to the few static variables that were not converted to thread-specific data.

PtTcl changes the semantics of the `exit` command slightly. In ordinary Tcl, `exit` terminates the whole application, and so it does not worry too much about releasing file descriptors or freeing memory obtained from `ckalloc()`. In PtTcl, `exit` will only terminate the current thread. This necessitated some additional clean-up actions in the Tcl core in order to avoid file-descriptor and memory leaks. Corresponding changes were made to the code that implements the `interp` command in order to get the command

```
interp delete {}
```

to do the right thing.

The `lsort` command was rewritten to use a merge-sort algorithm [Knuth] instead of the `qsort()` function. This change had the side-effect of making the `lsort` command both recursive and stable. It turns out that it is also a little faster. This change has been folded into the Tcl core as of version 8.0.

In standard Tcl, the `env` array variable contains the values of all environment variables. Changes made to `env` are applied to all interpreters. This behavior is not implemented in PtTcl, however. Each interpreter in PtTcl still has the `env` array containing the environment, but changes to this array are not copied into other interpreters.

During early testing, we discovered that the `printf()` function supplied with MIT Pthreads was not thread-safe. Rather than fix MIT Pthreads, we found it easier to supply our own thread-safe version of the `printf()` function, which we placed in the source file "`generic/tclPrintf.c`". We later used some enhanced features of this alternative `printf()` in the implementation of `Tcl_ThreadSend()`, so even though MIT Pthreads has now been fixed the new `printf()` implementation must remain.

New code was added to implement the `shared` and `thread` commands. The code for `shared` was added to "`generic/tclVar.c`" since it needed access to information local to that file. The `thread` command and the new `Tcl_ThreadCreate()` and `Tcl_ThreadSend()` functions are all found in a new source file named "`generic/tclThread.c`".

And, of course, an occasional mutex had to be added here and there, and some functions of the standard C library were changed to their thread-safe equivalents. (Example: `gmtime()` was changed to `gmtime_r()`.) Overall, the implementation of PtTcl was reasonably simple thanks to the extraordinarily clean implementation of the original Tcl core.

6 Obtaining And Building PtTcl

The latest sources to PtTcl can be found at

http://users.vnet.net/drh/pttcl.tar.gz

To build PtTcl, first obtain and unpack the source tree, then `cd` into the directory `pttcl7.6a1/unix` and enter one of the commands

```
./configure --enable-pthreads
```

or

```
./configure --enable-mit-pthreads
```

Use the first form at installations where POSIX threads programs can be built by linking in the special `-lpthreads` library. The second form is for installations that use MIT Pthreads [Provenzano] and require the special `pgcc` C compiler.

After configuring the distribution, type

```
make
```

to build a `tclsh` executable. Note that if you omit the `--enable-pthreads` or `--enable-mit-pthreads` option from the `./configure` command, then the `tclsh` you build will not contain support for Pthreads.

7 Status Of PtTcl Development

We developed and tested PtTcl under the Linux version 2.0. For the Pthreads library, we have used both Chris Provenzano's user-level implementation [Provenzano] (also known as MIT Pthreads) and a kernel-level Pthreads implementation by Xavier Leroy [Xavier] built on the `clone()` system call of Linux. Neither of these Pthreads implementations is without flaw. Under some versions of MIT Pthreads, the `exec` Tcl command did not work reliably. (Later versions of MIT Pthreads work better.) The `exec` command works fine using Linux kernel Pthreads, but under heavy load, the kernel's process table has been known to become corrupt, resulting in a system crash. We suspect that both of these problems are bugs in the underlying Pthreads implementation (or the Linux kernel), not in PtTcl.

PtTcl was written for and has been heavily used in a multi-processor industrial controller that implements its control algorithms using a data-flow model. Each node of the data-flow graph is a PtTcl script running in its own thread. PtTcl has survived extensive abuse testing of the controller software with no errors or memory leaks. But these tests have only exercised those parts of PtTcl which are actually used in the controller application. The `shared` or `exec` commands have not be heavily tested nor have there been many attempts to run more than one interpreter in a thread at a time. We suspect that bugs remain in these areas.

While PtTcl has so far only been tested under Unix, there is nothing in the implementation of PtTcl that would preclude its use under Windows or MacIntosh. All that is needed is a library for the target platform that implements basic Pthreads functionality. We are not aware of any such library but suspect that they do exist. It should not be much trouble to implement Pthreads as a wrapper around the native Windows or MacIntosh thread capability. PtTcl only uses a few of the more basic Pthreads routines, so most of the Pthreads library could remain unimplemented.

8 Acknowledgements

PtTcl was developed for and released by Conservation Through Innovation, Ltd., a manufacturer of environmental and industrial control systems based in Prescott, Arizona.

9 Availability

An online manual and complete source code for PtTcl is available from

http://users.vnet.net/drh/pttcl.html

References

[MtTcl] "MT-Tcl" To appear in *Tcl/Tk Tools* by Mark Harrison. O'Reilly & Associates, Sebastopol, CA. Estimated publication date: June 1997.

[Pthreads] *Portable Operating System Interface (POSIX) – ANSI/IEEE Std 1003.1.* Institute of Electrical and Electronics Engineers, New York, NY. 1996.

[Knuth] Algorithm L on page 165 of *The Art Of Computer Programming* Vol. 3. By Donald E. Knuth. Addison Wesley Publishing Company, Reading, MA. 1973.

[Provenzano] *Pthreads: General Information.* A web page by Christopher Angelo Provenzano. `http://www.mit.edu:8001/people/proven/p-threads.html`

[Xavier] *The Linux Threads Library.* A web page by Xavier Leroy. `http://pauillac.inria.fr/ xleroy/linux-threads/`