USENIX Association

Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference

Boston, Massachusetts, USA
June 25–30, 2001

Building an open-source Solaris-compatible threads library

John Wood
Compaq Computer (UK) Ltd
woodjohn@compaq.com

Abstract

Applications that use the Solaris threads application programming interface (API), e.g. *thr_create(), mutex_lock(), cond_signal(),* etc. [1], are generally non-portable. Thus to port an application that uses Solaris threads to another platform will require some degree of work.

Solaris now supports the POSIX threads API as well as the Solaris threads API. Therefore to make a Solaris threaded application portable, the ideal is to re-code the threaded part of the application to use POSIX threads. However, the Solaris threads API has some unique functionality over the POSIX threads API. This can make the task of converting a Solaris threaded application to use POSIX threads be very time-consuming and hence expensive, sometimes prohibitively so.

This paper outlines an alternative approach to porting applications that use the Solaris threads API, which is to use an open-source Solaris-compatible threads library that layers upon a POSIX threads library. The objective is to allow an otherwise-portable Solaris threaded application to be ported by simply rebuilding on the target platform using the Solaris-compatible threads library and header-files. This reduces the cost of porting the application.

1 Introduction

1.1 Porting applications from Solaris

Many independent software vendors (ISVs) develop applications on Sun's Solaris platform. These applications may have used the Solaris threads[1] API: this is most likely for mature applications that pre-date the POSIX threads[2] API.

To port an application that uses the Solaris threads API to another UNIX platform typically requires re-working the application to use the POSIX threads API. Depending upon the application, the amount of re-working needed may vary from simply being a matter of a few editor substitutions, to re-engineering the application.

An alternative to requiring every Solaris threads application to be re-worked for portability is to provide a portable Solaris-compatible threads library. Once the Solaris-compatible threads library has been ported to a target platform, threaded Solaris applications may be ported by just re-compiling.[3] This paper describes a Solaris-compatible threads library, for which the acronym STL is used. An ISV porting a threaded Solaris application may choose not to use STL, but may find the details of the STL implementation useful to re-work their application for portability.

1.2 Objective

The objective of STL is to provide a high degree of Solaris threads compatibility on the target platform for the minimum amount of effort. To reduce effort and maximize portability, STL is layered upon the POSIX threads library. This limits what STL is capable of, but this was considered an acceptable compromise rather

[1] The Solaris threads API is a subset of the UNIX International (UI) threads API, which is also supported by SCO's UnixWare 2.

[2] *POSIX threads* refers to the thread-specific part of the formal standard ISO/IEC 9945-1:1996 [POS96] that is commonly known as POSIX 1003.1-1996. This standard integrates the original POSIX 1003.1-1990 [POS90] standard (base operating system API) with the amendments 1003.1b-1993 (real-time extensions) and 1003.1c-1995 (threads). See *Threaded Programming Standards* [2] and the comp.programming.threads FAQ [3] for further information.

[3] But see also the section *Non-objectives* for a description of other potential porting issues.

than trying to implement a fully compatible Solaris threads library from scratch.

1.3 Non-objectives

The scope of STL is limited to providing the Solaris threads API. STL does not attempt to solve generic porting issues such as:

- Differences in system and library calls.
- Architectural differences. E.g. 32 to 64 bits; endianism, etc.
- Compiler and other build-tool differences.

Platform-specific porting guides such as the *Sun Solaris to Compaq Tru64 UNIX Porting Guide* [4] cover these sorts of issues.

1.4 Solaris Compatibility Libraries for Tru64 UNIX

STL was originally developed as part of the *Solaris Compatibility Libraries (SCL)* [5] for Compaq's Tru64 UNIX operating system. SCL was developed to ease the porting of Solaris applications to Tru64 UNIX.

SCL v1.1 was released in April 2000, and is available for free download. SCL v1.1 provides:

- Solaris Threads: An implementation of Solaris thread functions layered upon POSIX threads.
- Remote Procedure Calls: A port of Sun's freely redistributable ONC RPC v2.3 software.
- Miscellaneous: Various library functions including asynchronous I/O, large file support, wide character and signal name functions.

SCL is supplied as-is, with no formal support. The SCL source-code is freely available, and is included in the download kit.

2 Implementation strategy

STL is written in the C programming language, and is compiled to produce a shared object library, which is supplied with two header files.

2.1 Comparing the Solaris threads API to the POSIX threads API

Most threads APIs provide similar functionality of being able to create and manage a thread, and to create and manage synchronization objects. The Solaris threads API supports mutexes, condition variables, read/write locks and semaphores as thread synchronization objects. POSIX threads supports most of these synchronization objects[4], but does not include read/write locks, which are an extension from the Single UNIX Specification Version 2 (SUSv2) [UX98] for the UNIX 98 brand[5]. Therefore many UNIX systems that support POSIX threads also support read/write locks.

In many cases the Solaris threads API and POSIX threads API are almost identical. For example, consider the *kill* function that sends a signal to another thread:

Solaris threads API:

```
int thr_kill(
       thread_t  target_thread_id,
       int       sig );
```

POSIX threads API:

```
int pthread_kill(
      pthread_t  target_thread_id,
      int        sig );
```

Both functions take the same parameters of a thread identifier and signal number, although the thread identifier type is different. Both functions return an integer value, which if 0 indicates success.

In essence, to implement *thr_kill()*, STL first defines the Solaris type *thread_t* to match the POSIX threads type *pthread_t*, and then *thr_kill()* just becomes a jacket routine to *pthread_kill()*.

Mapping the Solaris threads types directly onto their POSIX threads equivalents potentially allows an application to mix Solaris thread API calls with POSIX thread API calls. However, STL does not currently support this. To support mixing would require STL to intercept application calls to some of the POSIX threads functions for thread management, such as:

---

[4] Actually, semaphores were defined by POSIX 1003.1b-1993 (real-time extensions) rather than POSIX 1003.1c-1995 (threads).

[5] Specifically from the *X/Open CAE Specification, System Interfaces and Headers, Issue 5* (also known as XSH5) part of SUSv2.

- *pthread_create()*
- *pthread_join()*
- *pthread_detach()*

so that STL's internal data structures are properly maintained. But mixing calls from the different APIs for synchronization objects should not be an issue.

For most Solaris threads functions, the STL jacket routines have to perform extra work to ensure compatibility. Two examples are given later in this section.

There are a number of areas where Solaris threads provides functionality that is not available within POSIX threads. The main areas are:

- Daemon threads.
- Joining any thread.
- Suspending and continuing a thread.

Each of these areas is described in a separate section. There is also a section on getting and setting information of another thread, which became necessary to implement several pieces of Solaris threads functionality.

For details of STL's functionality and restrictions, see the *SCL Users Guide* [6]. Specifically, see section 3.2: *STL Functionality*; Appendix A: *Mapping of Solaris thread types to POSIX thread types by STL*, and Appendix B: *Solaris thread functions implemented by STL*.

2.2 STL design issues

The following decisions were made when designing and implementing STL:

- Always try to be compatible with Solaris threads, even if this may impact performance.
- Be compatible with both the documented behaviour and the undocumented, observed behaviour of Solaris threads.

Two examples are now described.

2.2.1 Function return values

STL implements many Solaris thread functions by calling the equivalent POSIX threads routine. But often the function return values documented for the two APIs differ. STL handles this using the following rules:

- If the STL implementation of a Solaris routine receives an error value from calling a POSIX threads function, and that value matches one of the error values documented for the Solaris routine, then it just returns that value.
- If the STL implementation of a Solaris routine receives an error value from calling a POSIX threads function that does not match one of the error values documented for the Solaris routine, then it maps the value to a valid Solaris error value for that function, logs a message[6], and returns the mapped value.

Thus a Solaris application does not need changing to handle the potentially different error value returns from POSIX threads functions, because STL handles this. But be aware that the reasons for a particular error value may be quite different when using STL.

For example, consider the Solaris threads and Tru64 UNIX POSIX threads functions to perform a timed-wait on a condition variable. The APIs are:

Solaris threads API:

```
int cond_timedwait(
      cond_t      *cvp,
      mutex_t     *mp,
      timestruc_t *abstime );
```

Function return values documented on Solaris:

> 0: successful completion.
>
> EFAULT: a parameter has an invalid address.
>
> EINVAL: invalid *abstime* parameter: if *abstime* is more than 100,000,000 seconds in the future, or the nanoseconds field of *abstime* is >= 1,000,000,000.
>
> ETIME: the time specified by *abstime* has passed.

The POSIX threads API:

```
int pthread_cond_timedwait(
      pthread_cond_t        *cond,
      pthread_mutex_t       *mutex,
      const struct timespec *abstime
      );
```

[6] See the *Error Logging* chapter of the *SCL Users Guide* [6]. Requires setting the *SCL_LOG_FILE* environment variable.

Function return values documented on Tru64 UNIX:

> 0: successful completion.
>
> EINVAL: the value specified by *cond*, *mutex* or *abstime* is invalid, or: the mutex was not owned by the calling thread at the time of the call, or: different mutexes are supplied for concurrent *pthread_cond_timedwait()* or *pthread_cond_wait()* operations on the same condition variable.
>
> ETIMEDOUT: the time specified by *abstime* has passed.
>
> ENOMEM: the POSIX threads library cannot acquire the memory needed to block using statically initialised objects.

The Solaris *cond_timedwait()* function can return a value of 0, EFAULT, EINVAL or ETIME. The *pthread_cond_timedwait()* function on Tru64 UNIX can return 0, EINVAL, ETIMEDOUT or ENOMEM.

If the STL implementation of *cond_timedwait()* receives an ENOMEM return value from calling *pthread_cond_timedwait()*, then it maps this value to EFAULT, which is one of those documented for the Solaris function. This is because Solaris applications might only check for specific error codes, rather than just testing the status for success. Additionally, STL logs a message to indicate when it is performing a mapping of error statuses (e.g. *"ENOMEM from pthread_cond_wait() mapped to EFAULT from cond_timed_wait()"*): these messages may be helpful to understanding the real reason for a particular STL function return value.

Note that a message is only logged when the POSIX threads function's return value is mapped to a different return value for the Solaris threads routine. For example, with *cond_timedwait():*

- If STL's *cond_timedwait()* is called with an uninitialized mutex, then EINVAL is returned from *pthread_cond_timedwait()* which is passed back from *cond_timedwait()* with no message logged because there is no mapping of errors (even though the reason for the EINVAL is different to what Solaris documents).

- If STL's *cond_timedwait()* receives ETIMEDOUT from *pthread_cond_timedwait()*, then it maps this to ETIME and logs a message indicating the mapping (even though the error codes have the same meaning).

2.2.2 Documented and observed behaviour of synchronization objects

Solaris documents that synchronization objects that are statically initialized to all zeros do not need to be explicitly initialized. It does not define the result when attempting to use uninitialized objects as function parameters: the observed behaviour is that Solaris implicitly initialises the object. This is consistent with the Solaris routines not having the error return EINVAL defined for uninitialized objects.

POSIX threads documents and returns EINVAL when uninitialized objects are used as parameters.

There is no portable way to validate that a POSIX threads synchronization object has been initialized, so STL does not support Solaris' observed behaviour of implicitly initializing uninitialized synchronization objects. Attempts to use an uninitialized object with STL functions results in an error value being returned, and typically an error-mapping message is logged. But there is an exception for statically-initialized-to-zero objects: STL tests for these, and will explicitly initialize them, to conform to Solaris' documented behaviour. These checks will impact performance, but compatibility is the main objective.

3 Daemon Threads

3.1 Introduction

Solaris threads provides the concept of *daemon* threads. Solaris threads defines that a process terminates when its last non-daemon thread terminates.[7]

POSIX threads does not support daemon threads. A POSIX threads process terminates when its last thread exits.

Daemon threads could be used by applications for housekeeping tasks. For example, a daemon thread may be created that periodically monitors disk space whilst the application is running.

[7] A threaded process will also terminate if any thread calls *exit()*, either explicitly, or, for the main thread only, implicitly if the main thread finishes without calling *pthread_exit()* (POSIX threads) or *thr_exit()* (Solaris threads).

3.2 Implementation

Conceptually, to implement support for daemon threads is quite simple: a count of the number of non-daemon threads must be maintained. This count may need adjusting when a thread is created, or when a thread terminates. If the non-daemon thread count becomes zero, then the process must be terminated.

The daemon-thread implementation sounds simple in outline, but now we look at the implementation in more detail. It requires:

- Keeping a global count of the number of non-daemon threads.

  Updates to the count need to be coordinated by using a mutex.

- Knowing when a thread is created, and whether it is a daemon or not.

  A daemon thread is created when the *THR_DAEMON* bit is set in the flags parameter to *thr_create().*

- Knowing when a thread terminates, and whether it is a daemon or not.

  Adjust non-daemon thread count if appropriate; if count is now zero, terminate the process.

The tricky bit here is the last bit: to know when a thread terminates, and to determine if it is a daemon-thread or not. The non-daemon thread count has to be decremented if the terminating thread is not a daemon. Thread-specific data, along with a destructor-routine, is used to implement this. This works as follows.

When STL initializes, it creates a thread-specific data key *STL.tsd_key*, and associates a destructor routine, *stl_tsd_key_destructor(),* with that key. When a thread that has data associated with *STL.tsd_key* terminates, it runs the *stl_tsd_key_destructor()* routine.[8]

When a new thread is created by calling *thr_create(),* the STL implementation of *thr_create()* dynamically allocates some memory *M* for a *stl_tsd_t* data structure, and fills in its fields. The *stl_tsd_t* structure has fields for:

- The flags parameter to *thr_create().*
- The *start-routine* parameter to *thr_create().*
- The *arg* parameter to *thr_create().*

STL's *thr_create()* then calls *pthread_create()* but specifies *stl_thread_start_rtn()* as the start-routine parameter, and *M* as the start-routine argument.

When the new thread starts, it first executes *stl_thread_start_rtn(M).* Within this routine the thread makes the memory *M* into thread-specific data for this thread by calling *pthread_setspecific(STL.tsd_key, M).* The new thread also determines from the thread's attribute flags in *M* if it is a daemon thread, and if not then the count of non-daemon threads is incremented.

The new thread then calls the user-specified start-routine with the user-specified argument, which are both extracted from *M*.

When the new thread terminates, it automatically executes the *stl_tsd_key_destructor()* routine. Ultimately, this frees up the memory *M,* but first it extracts the thread's attribute-flags from *M,* and determines from the flags whether this thread is a daemon thread or not. If the thread is not a daemon, then the count of non-daemon threads is decremented; and if this count is now zero, then process is terminated by calling *exit().*

3.3 Daemon thread considerations

On Solaris, when a Solaris-threaded process forks, the child process has a copy of all the threads, whereas with POSIX threads the child only has one thread. Hence the STL count of non-daemon threads needs resetting. This is implemented by calling *pthread_atfork()* to declare a routine which is invoked by the child process after a fork, that resets the thread count.

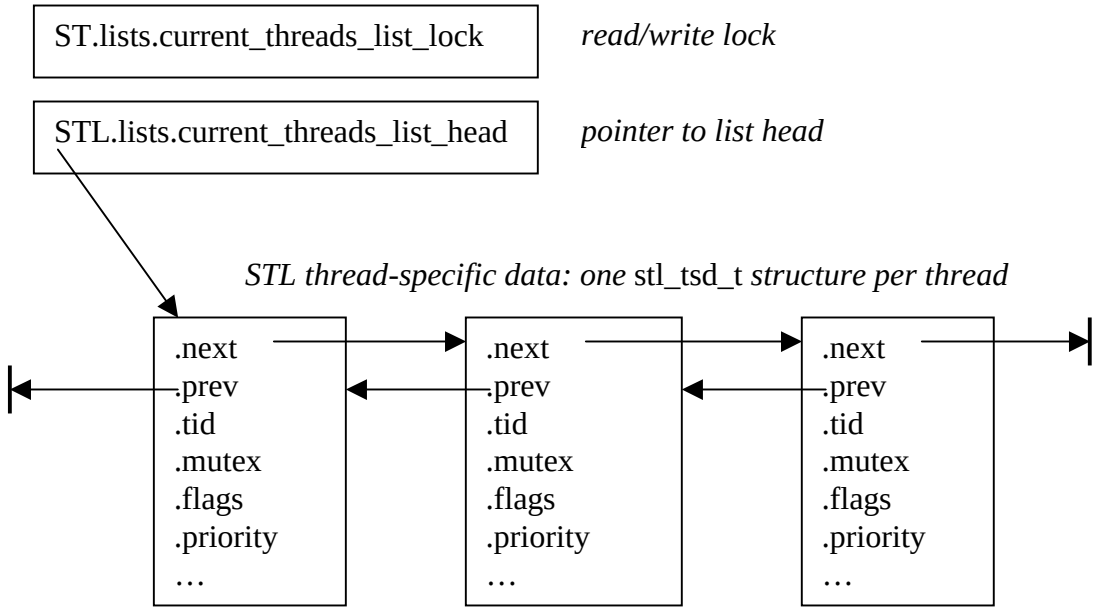
For STL to support the mixing of Solaris threads and POSIX threads calls, which Solaris allows, it would need to intercept calls to *pthread_create()* to increment the non-daemon thread count.

4 Getting and setting information of another thread

Normally a thread only needs to access its own thread-specific data, which it can readily find by calling *pthread_getspecific().* However, there are a few in-

[8] An important feature of the thread-specific data destructor routine is that this routine always gets called, regardless of how the thread terminates.

Figure 1. List of current threads.



stances when it is necessary for one thread to access another thread's STL thread-specific data. For example:

- when *thr_getprio()* or *thr_setprio()* are called, to get/set another thread's priority.[9]
- when *pthread_detach()* is called, to make the target thread detached.

So a mechanism is needed whereby one thread can find and access another thread's STL thread-specific data.

4.1 List of Current Threads

STL maintains a list of all the current threads in the process. One thread can search the list of current threads to find and subsequently access another thread's STL thread-specific data.

The list of current threads is currently implemented as an unsorted linked list. The STL thread-specific data-structure *stl_tsd_t* is extended to make it be an element of the linked-list by adding pointers to the next and previous entries in the list. These two pointers make it quicker for a thread to unlink itself from the list. The *stl_tsd_t* structure is also extended to include the thread identifier *tid* of that thread, and a mutex for that thread. If one thread wants to get or set another thread's attributes, then it must lock the target thread's mutex.

A global pointer, *STL.lists.current_threads_list_head*, points to the head of the list of current threads. New thread-list entries are added to the head of the list. The list will never be empty, except when the last thread of a process is terminating.

Access to the list is coordinated by a read/write lock, *STL.lists.current_threads_list_lock*. The read/write lock gives better concurrency than a mutex, because write-access to the list is only required when a thread is being created (added to the list) or terminating (removed from the list).

Figure 1 illustrates the current-threads list.

Only the thread itself can add itself to the list of current threads, or remove itself from the list of current threads. This has implications for the locking assumptions. A new thread adds itself to the current-threads list by executing code within the *stl_thread_start_rtn()* routine. A thread removes itself from the list when it terminates by executing code within the *stl_tsd_key_destructor()* routine.

[9] Solaris provides functions to set and get a thread's priority. Since the POSIX threads API has no direct equivalent to explicitly get or set a thread's priority, STL only stores a thread's priority so that *thr_getprio()* returns the value set by *thr_setprio()*.

The STL implementation currently uses a linear search to locate a specified thread's *stl_tsd_t* entry. This may have poor-performance implications when the number of current-threads is large, but this is considered acceptable based on the premise that it is not that often that one thread modifies another thread's *stl_tsd_t* entry. Usually a thread will modify its own *stl_tsd_t* entry, which it finds quickly by calling *pthread_getspecific()*. Thus to implement a function like *thr_setprio(tid,pri)* the sequence of events is:

- Lock the current-threads-list for reading.
- Search the current-threads-list, starting from the list-head:
    - If list-entry's thread-ID matches the one we're looking for, then:
        - Lock that thread's mutex
        - Access that thread's data
        - Unlock target thread's mutex
        - Set *return_status* to indicate success
        - Break out of search
    - Else:
        - Move on to the next thread's entry in list
        - If this is the end of the list
            - Set *return_status* to indicate thread not found (ESRCH)
- Unlock the current-threads-list
- Return *return_status*

5 Joining any thread

5.1 Introduction

When a non-detached Solaris or POSIX thread terminates, it should be joined to obtain its return-status, and to enable the threads library to release any resources that were not released when the thread terminated. Failure to join a non-detached thread results in a *zombie*[10]-like thread, which can cause memory leaks.

Compare the Solaris threads and POSIX threads APIs for joining a thread:

[10] In UNIX, a zombie process is a process that has terminated but has not been reaped by the parent process calling one of the *wait()* system calls.

Solaris threads API:

```
int thr_join(
      thread_t tid,
      thread_t *ret_tid,
      void **ret_val );
```

POSIX threads API:

```
int pthread_join(
      pthread_t tid,
      void **ret_val );
```

Both the Solaris threads and POSIX threads APIs let you join with a specific thread identified by *tid*. In addition, if you specify a thread identifier *tid* of 0 on Solaris, *thr_join()* will join with any terminated non-detached thread that has not yet been joined, and will return the identifier of the joined thread in *ret_tid*. If there are no non-detached terminated threads waiting to be joined, then *thr_join()* with a *tid* of 0 will wait for the first such thread to terminate, and will join with it. This unique Solaris functionality is called *join-any-thread*.

Note that when a joinable thread terminates, there may be zero, one or many threads waiting to join that specific thread, as well as other threads waiting to join any thread. Solaris does not define the behaviour for this situation, but STL gives preference to the thread(s) waiting to join the specific thread over the threads waiting to join an unspecified thread.

5.2 Implementation

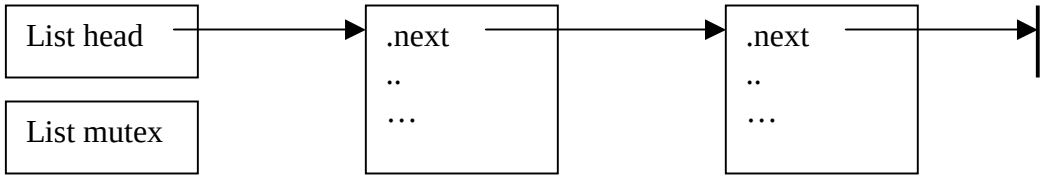
Three new lists are used by STL to implement join-any-thread:

1. List of threads waiting to join a specific thread (*join-specific list).*
2. List of threads waiting to join any thread *(join-any list).*
3. List of joinable threads that have terminated and are awaiting joining *(terminated list).*

The format of these lists is illustrated in Figure 2. Code is added to *thr_join()* and to the STL thread-specific data destructor routine to maintain these lists. The pseudo-code is shown below.

Figure 2. Lists to implement *join-any-tread*.

Three lists, all follow the same format (immediately below), but contain different data (see table below).



List entries

Contents of list entry:

Join-specific list	***Join-any list***	***Terminated list***
• ID of specific thread being waited upon to join with	• ID of thread to join with,. Initially set to 0. Used by waiter as a predicate for a *pthread_cond_wait()* loop • Mutex • Condition Variable	• ID of terminated thread

5.2.1 To join with a specific thread:

- If (specific thread exists in list of current threads) and (specific thread is joinable [not detached])
  - Add an entry for this thread to the join-specific list
- *pthread_join(specific_thread, ret_val)*

5.2.2 To join with any thread:

- If (terminated-list is not empty)
  - Remove entry from head of terminated-list; extract terminated thread ID
- Else
  - Add a new entry with *tid=0* to the join-any list
  - Block until the join-any list entry contains a suitable terminated thread ID to join with
  - Remove our entry from the join-any list
- *pthread_join(terminated thread ID, ret_val)*

5.2.3 When a thread terminates:

- If (terminating thread is detached)
  - Return */* thread is not joinable */*
- If (join-specific list has any entries that match this thread's ID)
  - Remove all matching entries from the list
- Else if (join-any list is not empty)
  - Put terminating thread's ID into the entry at head of list, and unblock the waiting thread
- Else
  - Add this thread's ID to the terminated list
- */* thread now finally terminates */*

6 Suspending and Continuing a thread

6.1 Introduction

The Solaris threads API allows one thread to stop and re-start another thread by calling the functions:

```
int thr_suspend( thread_t tid );
```

```
int thr_continue( thread_t tid );
```

Threads can also be created in a suspended state by setting the *flags* parameter to *thr_create().*

POSIX threads does not support the suspend and continue operations on a thread.

6.2 Implementation

To implement *thr_suspend()* and *thr_continue(),* two approaches were considered:

1. Tell the thread scheduler that the specified thread is to be suspended and hence not scheduled to run until further notice.
2. Asynchronously interrupt the to-be-suspended thread.

The first approach is ruled out because it would require changes to the underlying POSIX threads implementation or operating system. Thus an asynchronous mechanism to interrupt another thread is required. The only suitable mechanism available is signals.

It is worth noting that in threaded programs, a signal handler is process wide (i.e. the same for all threads), and the signal mask of blocked signals is thread-specific (i.e. can vary per thread).

In the book *Programming with POSIX Threads* [7], there is an example implementation of the thread-suspend and thread-continue routines, which uses two signals for suspend and continue respectively. This forms the basis of the STL implementation of *thr_suspend()* and *thr_continue(),* with a few modifications.

In essence, STL implements *thr_suspend()* by calling *pthread_kill()* to send a *suspend* signal to the target thread. The signal handler for the *suspend* signal then calls *sigwait()* to block until it receives a *continue* signal. *sigwait()* is one of the few functions that can be safely called from within a signal handler.[11]

A suspended thread is resumed by another thread calling *thr_continue().* This function is also implemented by calling *pthread_kill(),* but sends a *continue* signal to the target thread. The signal handler for the *continue* signal is a null routine. Upon receipt of the *continue* signal the target thread returns from both the *continue* and *suspend* signal handlers to resume whatever it was doing.

By default on Tru64 UNIX, STL uses *SIGUSR1* as the *suspend* signal, and *SIGUSR2* as the *continue* signal. However, the signal numbers used by STL can be changed by setting environment variables: see the *SCL Users Guide* [6].

Solaris documents that *thr_suspend()* does not return until the target thread is suspended. In other words, the thread executing *thr_suspend()* needs to know that the target thread has received the *suspend* signal. (A thread may have temporarily blocked a set of signals, in which case the *suspend* signal is pending until the thread unblocks that signal). Thus within the *suspend* signal handler routine, the thread being suspended needs to indicate to the caller of *thr_suspend()* that it is now suspended. A global semaphore is used for this purpose. The suspended thread calls *sem_post(),* which the thread executing *thr_suspend()* waits upon by calling *sem_wait(). sem_post()* is another function that can safely be called from within a signal-handler.

6.3 Further complications

The actual implementation has other factors to consider:

- The use of a global semaphore to acknowledge that a thread is suspended means that STL must serialize access to *thr_suspend().*

  Actually, serializing access to *thr_suspend()* has the advantage that it makes it easier to implement, albeit at the expense of concurrency.

- STL needs to handle a thread suspending or continuing itself, and handle threads that are created in a suspended state.

- STL cannot allow the last non-suspended thread to suspend itself: STL must return EDEADLK in this situation, as documented by Solaris.

- If one thread calls *thr_suspend(X),* and a fraction later another thread calls *thr_continue(X),*

[11] POSIX 1003.1-1996 defines which functions are reentrant with respect to POSIX signals.

STL needs to ensure that the result is that thread X is running, not suspended.[12]

This is achieved by serialising access to *thr_suspend()* and *thr_continue()* by using the same mutex for both functions.

- STL prevents a thread from being suspended whilst it holds one of the internal STL locks (such as a mutex for a join-any thread list) by calling *pthread_sigmask()* to temporarily blocking signals, to prevent the application from hanging within STL.

7 STL Status

As previously stated, STL has been released as part of the Solaris Compatibility Libraries for Tru64 UNIX. STL is also being ported to Linux.

7.1 Linux port

A provisional STL library has been built on Linux, and some test programs run. Problems with thread suspend and continue have been encountered, and are as yet unresolved.[13]

It is hoped that the Linux version of STL will be complete and available by the time of the USENIX Annual Technical Conference in June 2001.

7.1.1 Experiences from the Linux port

The main requirements for porting STL to another platform are that the target platform has an ANSI-C compiler ([ANSI89], ratified by [ISO90]), and a POSIX threads library with the read/write locks extensions.

The GNU C compiler is being used on Linux, with the *-ansi* and *-Wall* switches. This has flagged several warnings, which is to be expected given that STL had never been ported before. The code has been changed and is now more portable.

When considering the Linux port of STL there was concern about *LinuxThreads* [8], the POSIX threads library on Linux. LinuxThreads implements POSIX threads by creating separate processes with the *clone()* system call. But the concern seems unfounded: the only real problem encountered so far with LinuxThreads is that threads within the same (logical) process actually have different process identifiers.

Initially it was thought that LinuxThreads did not have POSIX threads' read/write locks. The reasons for thinking this were:

- Most POSIX thread functions on Linux have man-pages or info-pages, but the read/write lock functions are not documented.
- A test program built on Linux got unresolved symbols for read/write locks.

The answer, found via the threads programming newsgroup [9], was that when compiling you had to define:

_XOPEN_SOURCE=500

before including the <*pthread.h*> header-file, in addition to defining:

_POSIX_C_SOURCE=199506L

These explicit definitions are not necessary on Tru64 UNIX.

Two other problems have been encountered during the Linux port. The first problem was trying to get the shared object library to run an initialization routine. This is achieved by specifying the

__attribute__ ((__constructor__))

directive, but you must use *cc* as the link driver, rather than *ld*, for this directive to be recognized. The other problem was with message catalogs, and the *gencat* utility in particular. On Linux *gencat* is white-space sensitive, in accordance with SUSv2. Thus with non-conformant input (that happened to work on Tru64 UNIX), *gencat* on Linux produced blank messages. The solution was to edit the input to *gencat* to be conformant.

7.2 Performance

It is worth restating that STL performance, whilst desirable, has never been a goal: compatibility is the objective. Using STL will always incur some overhead compared to using native POSIX threads.

[12] When a thread processes its pending signals, there is no guarantee that they are processed in the same order that they were received.

[13] Thread suspend and continue were most troublesome in the original STL implementation on Tru64 UNIX.

Table 1 shows how STL affects the performance in a couple of simple tests, where a test was coded in both Solaris threads and POSIX threads. The tests were performed using STL v1.1 on a Tru64 UNIX v5.1 system (a Compaq Alphaserver ES40 with 4 CPUs, but with the number of CPUs active varied from 1 through to 4).

The STL performance numbers look bad in isolation, especially for thread create/join. But these should be considered worst-case figures, and need to be viewed in the context of how frequently each operation occurs in a real application.

Table 1. Relative performance of POSIX threads and STL threads for simple tests.

Test	***Ratio of POSIX:STL performance***
Loop of thread create and thread join	varies from 7:1 to 3:1 (depends on # CPUs active)
Loop of mutex-lock and mutex-unlock	1.1:1

For example, for the mutex-locking test loop, just a single thread is executing, so that there is no contention for the mutex. Consequently the fastest path is taken through the POSIX mutex-locking code. In real applications there will probably be some contention for the mutex, which may result in a locking-thread having to do extra work to block on an already-locked mutex, and the unlocking thread also having to do extra work to wake up the blocked thread. This extra work will make the overhead of STL be less apparent.

For the thread-create-and-join loop using STL, a considerable amount of CPU time was observed being spent in system-mode, compared to using native POSIX threads. This indicates a high number of system calls, the most probable cause being the calls to *pthread_sigmask()* to block and restore signals when the thread locks an STL resource.

The way an application uses threads is also a big factor on STL performance. For example, consider how a threaded program might be coded to find the first 10,000 prime numbers, using a 4-CPU system. One approach might be to create a new worker-thread to test if one specific number N is prime (for N = 1, 2, 3, 4, etc.), and to have three of these worker threads concurrently active (the main thread makes four threads; i.e. one per CPU). But a better approach would be to create three "permanent" worker-threads that loop repeatedly, testing successive numbers for prime. The latter approach requires more synchronization between the threads to determine which number to test next, (whereas in the former case the main thread can tell each new thread which number to test via the user-argument to the thread-create routine), but it avoids the overhead of repeatedly creating threads. Both programs require synchronization when a prime is found, to increment a global counter and to store the new prime number in a global array.

Table 2 shows the comparative performance of two prime-number programs coded to each approach, using both the native POSIX threads and STL. Values in the table give the number of primes found per second, using the 4-CPU ES40 again. Bear in mind that for half the numbers tested for prime, the test will complete within a very few instructions, because even numbers are not prime.

Table 2. Performance of prime-number programs. Results in primes-per-second: higher is better**.**

Program	***POSIX threads***	***STL threads***	***Ratio POSIX:STL***
New thread for every number	2.5K	0.6K	4.3:1
Three *permanent* worker threads	63K	61K	1.03:1

The results show that for the case when a new thread is created for every number being tested, the overhead of STL's thread-create and thread-join has considerable impact: the STL program is over four times slower. But by creating the worker-threads just once and using additional synchronization, the overall performance is much higher in both cases, and the overhead of STL is minimal.

The results confirm that the overhead of STL is substantial for the areas of thread creation, thread termination, and joining a thread. The results also show that the overhead of STL for synchronization object manipulation is low. It is envisaged that threaded applications will use the synchronization routines much more frequently than the thread routines, and hence the general overhead of STL should be low.

7.3 Future plans

STL is open-source. The hope is that it will be enhanced and extended by anyone who finds it useful.

Completing the Linux port of STL should broaden its potential use.

8 Summary

This paper describes STL, a Solaris-compatible threads library, which layers upon a POSIX threads library. STL enables applications that use the Solaris threads API to be recompiled on another UNIX platform that supports the POSIX threads API.

This paper describes the major functionality of STL, and how it is implemented.

STL is freely available in open-source format as part of SCL, and a binary library is available for Tru64 UNIX.

References

1. Sun Microsystems.

   Solaris Multithreaded Programming Guide.

   Prentice Hall, 1995. ISBN 0-13-160896-7.

2. Dave Butenhof.

   Threaded Programming Standards.

   http://csa.compaq.com/CTJ_Article_29.html

   February 2000.

3. Bryan O'Sullivan. 1997.

   Answers to frequently asked questions for comp.programming.threads.

   http://www.serpentine.com/~bos/threads-faq/

4. *Sun Solaris to Compaq Tru64 UNIX Porting Guide.*

   http://www.unix.digital.com/faqs/publications/pub_page/porting.html

5. *Solaris Compatibility Libraries (SCL) for Compaq Tru64 UNIX*.

   http://www.tru64unix.compaq.com/complibs/

6. *Solaris Compatibility Libraries' Users Guide for Tru64 UNIX*.

   http://www.tru64unix.compaq.com/complibs/documentation/html/scl_ug.html

7. David R. Butenhof.

   Programming with POSIX Threads.

   Addison-Wesley, 1997. ISBN 0-201-63392-2.

8. Leroy Xavier.

   The LinuxThreads Library.

   http://pauillac.inria.fr/~xleroy/linuxthreads/

9. *comp.programming.threads*

   Internet newsgroup for discussing threads programming.

Standards

ANSI89: American National Standards Institute (ANSI): *American National Standard X3.159-1989 - ANSI C*

ISO90: International Standardization Organization (ISO): ISO/IEC 9899:1990, *Programming languages - C.*

POS90: ISO/IEC standard 9945-1:1990: [IEEE Std 1003.1-1990]. *Information technology - Portable Operating System Interface (POSIX®) - Part 1: System Application Program Interface (API) [C Language]*.

POS96: ISO/IEC standard 9945-1:1996 [IEEE/ANSI Std 1003.1, 1996 Edition]. *Information Technology-Portable Operating System Interface (POSIX®)-Part 1: System Application: Program Interface (API) [C Language]*.

UX98: The Open Group. *Single UNIX Specification Version 2*. 1998.

http://www.unix-systems.org/