POSIX/Solaris Comparison

C

Introduction

pthreads is a POSIX API (draft) standard that allows the creation of programs with multiple threads of control in a process. The SunSoft Early Access implementation of pthreads is based on Draft 8 of the POSIX 1003.4a pthreads standard. SunSoft currently supports a threads API known as Solaris threads. The intent of this appendix is two fold:

- *It is an aid to programmers already familiar with Solaris threads who would like to begin using the pthreads API or who would like to convert from Solaris threads to pthreads.*
- *It highlights the differences between pthreads and Solaris threads so that users can choose one API, the other, or both, based on the functionality offered.*

It is assumed that readers are proficient in threads programming so the question of method is not addressed. It is meant to be used in conjunction with other supporting documentation such as the pthreads UNIX man pages and The Guide to Multi-Thread Programming (the main body of this manual).

pthreads and Solaris threads share a high level of correspondence in both API action and syntax. However, there are several points of divergence. First, function names are entirely different, though easily correlated. The pthreads convention is to attach the prefix `pthread_` to each descriptive function root name.

There is not an exact match between the two APIs. pthreads includes functions not supported in the Solaris interface; Solaris threads supports functions not found in pthreads. For those functions that do match, the associated arguments may not - though the information content is effectively the same. In all cases function argument types will be different. The pthreads features not found in Solaris threads; attribute objects, cancellation options, and scheduling policies, are explained in this appendix. Feature differences are summarized here:

- Features in Solaris threads API but not in pthreads API
  - Readers/writer locks
  - Ability to create "daemon" threads
  - Suspending and continuing a thread
  - Setting concurrency (requesting a new LWP): determining concurrency level
- Features in pthreads API but not in Solaris threads API
  - Attribute objects
  - Cancellation options
  - Scheduling policies

The pthreads API and the Solaris threads API are two different solutions to the same problem, namely building parallelism into application software. This does not imply that they are mutually exclusive. There are no restrictions on intermixing pthreads functions with Solaris functions (although style restrictions should be considered). The strength of this approach is that functionality not found in one can be used to enhance the other. Similarly, there are no restrictions in running applications using pthreads exclusively with applications using Solaris threads exclusively on the same system.

New Features in pthreads

From a programming perspective, the pthreads API is effectively identical with the Solaris threads API - with the notable exception of scheduling, cancellation and attribute use. Moving between environments is largely a matter of simple substitution. For developers familiar with multithreaded programming techniques as used in Solaris threads, pthreads will offer no surprises.

Feature	Overview	Detail
Attribute Objects Overview	*page C-167*	*page C-175*
Cancellation Options Overview	*page C-168*	*page C-209*
Scheduling Policies Overview	*page C-170*	*page C-186*

Attribute Objects Overview

Threads entities, i.e. threads themselves and synchronization variables, can exist in a number of different states. For instance a thread may be "detached" or "non-detached". Or the scope of a mutex may be inter-process or intraprocess. The convention in Solaris threads is to use flag arguments to specify the state in which an entity is to be created. The pthreads approach is to request the initialization of an attribute object, an opaque data type allocated and returned (in a default state) by a function call. An attribute type is defined for each relevant threads entity. Any number of attribute objects of a given type can be initialized.

When a function is called to create a threads entity (e.g. thread, mutex, etc.) requiring state initialization, an argument points to an attribute object. When the entity is created it is set to the state indicated by the attribute object. If some state other than the default is needed, there are pthreads functions to set the appropriate state in the attribute object.

There are two primary advantages to using attribute objects. First it adds to code portability. There may be cases where supported attributes vary between implementations. Even so, there will be no need to modify function calls that create thread entities since the attribute object is hidden from the interface. If the target port supports attributes not found in the current port, provision must be made to manage the new attributes. This is an easy porting task though, because attribute objects need only be initialized once in a well defined location.

The second advantage is that state specification in an application is simplified. As an example, consider that several sets of threads may exist within a process, each providing a separate service, each with its own state requirements. At some point in the early stages of the application, a thread attribute object can be initialized for each set. All future thread creations will then refer to the attribute object initialized for that type of thread. The initialization phase is simple and localized. Any future modifications can be made quickly and reliably.

Attribute objects require attention at process exit time. When the object is initialized, memory is allocated for it. This memory must be returned to the system. Destructor function calls are provided to do this.

Cancellation Options Overview

pthreads introduces the notion of cancellability to threads programming. Cancellation is an option when all further operations of a related set of threads are undesirable or unnecessary. The best recourse is simply to cancel all thread action, restore things to a consistent state, and return to the point of origin. One example might be an asynchronously generated cancel condition such as a user requesting the cancellation of some running application. Another example might be the completion of a task undertaken by a number of threads. One of the threads may ultimately complete the task while the others continue to operate. Since they are serving no purpose at that point, the others should be canceled.

There are dangers in performing cancellations. Mostly, they deal with properly restoring invariants and freeing shared resources. If some thread is canceled without due care, it may leave a mutex in a locked state, leading to deadlock conditions. Or it may leave a region of memory allocated with no means of identification and therefore no way to free it.

pthreads specifies a cancellation interface that permits or forbids cancellation programmatically. It also allows the scope of cancellation handlers, which provide clean up services, to be defined so that they are sure to operate when and where intended.

Cancellations may occur under three different circumstances. They may occur 1) asynchronously, 2) at various points in the execution sequence as defined by the standard, or 3) at discrete points specified by the application. In all cases care must be taken so that resources and state are restored to a condition consistent with the point of origin.

A cancellation is effected by calling the function `pthread_cancel()` with a thread ID as an argument. How the cancellation request is treated depends on the state of the target thread. Two functions, `pthread_setcancelstate()` and `pthread_setcanceltype()`, determine that state. For more information see "Cancellation" on page C-209.

A cancellation point can be established in a thread by inserting the function call `pthread_testcancel()`. When this function is executed, and if a cancellation is pending, then `pthread_testcancel()` will not return. Otherwise, it has no effect.

In addition to the programmatically determined `pthread_testcancel()` call, the pthreads standard specifies a number of cancellation points. These include threads waiting in `pthread_cond_wait()` and `pthread_cond_timedwait()`, threads waiting for the termination of another thread in `pthread_join()`, and threads blocked on `sigwait()`. There are also a number of standard library calls that act as cancellation points. In general these are functions in which threads may block.

In order to restore conditions to a state consistent with that at the point of origin, e.g. clean up allocated resources and restore invariants, pthreads specifies the use of cleanup handlers. Two functions, `pthread_cleanup_push()` and `pthread_cleanup_pop()`, are used to manage the handlers. `pthread_cleanup_push()` pushes a cleanup handler onto a cleanup stack (FIFO). `pthread_cleanup_pop()` pulls it off the stack.

If `pthread_cleanup_pop()` is called with a nonzero argument, then the handler popped off the stack is executed; otherwise it is removed and discarded. `pthread_cleanup_pop()` is effectively called with a nonzero argument if a thread either explicitly or implicitly calls `pthread_exit()` or if the thread accepts a cancel request.

Placement of cancellation points and the effects of cancellation handlers must be based on an understanding of the application under consideration. A mutex is explicitly not a cancellation point and should be held only the minimum essential time.

Regions of asynchronous cancellation should be limited to sequences having no external dependencies which could result in dangling resources or unresolved state conditions. Care should be taken to restore cancellation state when returning from some alternate, nested cancellation state. The interface provides features to facilitate this restoration. `pthread_setcancelstate()` preserves the current cancel state in a referenced variable; `pthread_setcanceltype()` preserves the current cancel type in the same way.

Scheduling Policies Overview

pthreads defines a policy and provides a mechanism for controlling the scheduling of threads onto processors. There are two aspects to the scheduling mechanism. One deals with setting the scope in which a scheduling policy applies. The other specifies the policy within a given policy domain.

The scheduling scope of a thread can be either PTHREAD_SCOPE_SYSTEM or PTHREAD_SCOPE_PROCESS. Threads with different scope state may coexist on the same system and even in the same process. If PTHREAD_SCOPE_SYSTEM is specified, then the set of threads for which this is true compete for processor resources according to a single policy. If PTHREAD_SCOPE_PROCESS is true, then all threads within a process which share this state compete for processor resources according to a single policy. Threads in other processes whose state is also PTHREAD_SCOPE_PROCESS are independent of this policy and conform to their own. It should be noted that creating a pthreads thread in the PTHREAD_SCOPE_SYSTEM is equivalent to creating a Solaris thread in the THR_BOUND state.

There are three possible scheduling policies.They are SCHED_FIFO, SCHED_RR, and SCHED_OTHER. SCHED_FIFO refers to a simple queue scheduling sequencer. Each thread is assigned a priority level; a queue is associated with each priority level. As threads at a given priority level become runable, they are tagged on the tail of the queue. They move in turn to the head of the queue where they are then scheduled onto the next available processing element (assuming no higher priority threads).

SCHED_RR refers to a round-robin algorithm.The round-robin algorithm is the same as the FIFO algorithm except that a time-quota is associated with each thread. A running thread whose time-quota has expired moves to the tail of its run queue.

Note – SCHED_OTHER is currently the only scheduling policy supported in the SunSoft pthreads (POSIX) implementation. Solaris threads implementation does not have multiple scheduling policies; it has only one default policy. SCHED_OTHER policy in pthreads (POSIX) is same as the default policy now supported in Solaris threads. Refer to the Solaris threads documentation for details on this policy.

Implicit in both the FIFO and round-robin policies is the ability to deal with priority inversion. Priority inversion occurs when some higher priority thread blocks waiting on a resource held by a lower priority thread. This problem can manifest when threads of different priority levels are competing for ownership of a mutex. To mitigate priority inversion, pthreads defines three scheduling classes for mutexes: PTHREAD_PRIO_NONE, PTHREAD_PRIO_INHERIT, and PTHREAD_PRIO_PROTECT.

A mutex created with the PTHREAD_PRIO_NONE class attribute ignores thread priority. A mutex created with PTHREAD_PRIO_INHERIT upgrades the priority of the thread holding it if some other thread with a higher priority is blocked on the mutex. The temporarily upgraded thread will run at the priority level of the thread with the highest level that is currently blocked on the mutex. This effect cascades as well, so that all threads of a lower priority that are holding mutexes and thus blocking progress of a higher priority thread will inherit the priority of the blocked thread.

A mutex created with the PTHREAD_PRIO_PROTECT class is assigned a priority level. Any thread that holds that mutex will inherit that priority as long as its own priority is lower than the mutex priority. If a thread holds more than one mutex initialized in the PTHREAD_PRIO_PROTECT state, it will inherit the highest priority level indicated by any of the mutexes.

This priority inheritance policy applies in the context of FIFO or RR scheduling.

Threads acquire scheduling states by first initializing a thread attribute object and then creating a thread with a reference to that attribute object. Traditionally, the convention when creating a thread is to have it inherit the state of its parent. pthreads allows the priority assignment mode to be configured by setting an attribute value. If the attribute object is set to PTHREAD_INHERIT_SCHED, then the priority of the new thread will be identical to the parent. If it is set to PTHREAD_EXPLICIT_SCHED, then state is set according to that specified in the thread attribute object.

C

API Comparison Summary

The following table compares the pthreads API and the Solaris threads API. If the comparable interface is not available either in pthreads or Solaris threads, it is indicated with the character '-'. The parameter and the arguments are different between pthreads and Solaris functions. Details of the pthreads functions can be found in the man (3T) sections.

Note that the entries in the pthread column of the table followed by "POSIX 1003.4" or POSIX.4 are part of the POSIX real time standard specification and not part of pthreads.

Table C-1 POSIX pthread and Solaris Thread comparison

pthread	Solaris Thread
pthread_create()	thr_create()
pthread_exit()	thr_exit()
pthread_join()	thr_join()
sched_yield()POSIX.4	thr_yield()
pthread_self()	thr_self()
pthread_kill()	thr_kill()
pthread_sigmask()	thr_sigsetmask()
pthread_setschedparam()	thr_setprio()
pthread_getschedparam()	thr_getprio()
-	thr_setconcurrency()
-	thr_getconcurrency()
-	thr_suspend()
-	thr_continue()
pthread_key_create()	thr_keycreate()
pthread_key_delete()	-
pthread_setspecific()	thr_setspecific()
pthread_getspecific()	thr_getspecific()
pthread_once()	-
pthread_equal()	-
pthread_cancel()	-
pthread_testcancel()	-

Table C-1 POSIX pthread and Solaris Thread comparison

pthread	Solaris Thread
`pthread_cleanup_push()`	–
`pthread_cleanup_pop()`	–
`pthread_setcanceltype()`	–
`pthread_setcancelstate()`	–
`pthread_mutex_lock()`	`mutex_lock()`
`pthread_mutex_unlock()`	`mutex_unlock()`
`pthread_mutex_trylock()`	`mutex_trylock()`
`pthread_mutex_init()`	`mutex_init()`
`pthread_mutex_destroy()`	`mutex_destroy()`
`pthread_cond_wait()`	`cond_wait()`
`pthread_cond_timedwait()`	`cond_timedwait()`
`pthread_cond_signal()`	`cond_signal()`
`pthread_cond_broadcast()`	`cond_broadcast()`
`pthread_cond_init()`	`cond_init()`
`pthread_cond_destroy()`	`cond_destroy()`
–	`rwlock_init()`
–	`rwlock_destroy()`
–	`rw_rdlock()`
–	`rw_wrlock()`
–	`rw_unlock()`
–	`rw_tryrdlock()`
–	`rw_trywrlock()`
`sem_init()` POSIX 1003.4	`sema_init()`
`sem_destroy()` POSIX 1003.4	`sema_destroy()`
`sem_wait()` POSIX 1003.4	`sema_wait()`
`sem_post()` POSIX 1003.4	`sema_post()`
`sem_trywait()` POSIX 1003.4	`sema_trywait()`
`pthread_mutex_setprioceiling()`	–
`pthread_mutex_getprioceiling()`	–
`pthread_mutexattr_init()`	–
`pthread_mutexattr_destroy()`	–

Table C-1 POSIX pthread and Solaris Thread comparison

pthread	Solaris Thread
pthread_mutexattr_setpshared()	*type* argument in cond_init()
pthread_mutexattr_getpshared()	–
pthread_mutexattr_setprioceiling()	–
pthread_mutexattr_getprioceiling()	–
pthread_mutexattr_setprotocol()	–
pthread_mutexattr_getprotocol()	–
pthread_condattr_init()	–
pthread_condattr_destroy()	–
pthread_condattr_getshared()	–
pthread_condattr_setshared()	*type* argument in cond_init()
pthread_attr_init()	–
pthread_attr_destroy()	–
pthread_attr_getscope()	–
pthread_attr_setscope()	THR_BOUND flag in thr_create()
pthread_attr_getstacksize()	–
pthread_attr_setstacksize()	*stack_size* argument in thr_create()
pthread_attr_getstackaddr()	–
pthread_attr_setstackaddr()	*stack_addr* argument in thr_create()
pthread_attr_getdetachstate()	–
pthread_attr_setdetachstate()	THR_DETACH flag in thr_create()
pthread_attr_getschedparam()	–
pthread_attr_setschedparam()	–
pthread_attr_getinheritsched()	–
pthread_attr_setinherisched()	–
pthread_attr_getschedpolicy()	–
pthread_attr_setsschedpolicy()	–

C

API Comparison Details

This section presents a more detailed comparison of Solaris threads and pthreads. Each subsection includes a functional description and also details on converting an existing Solaris call to the corresponding pthreads call.

pthreads Attributes Objects

pthreads entities, i.e. threads and synchronization objects, can each exhibit more than one kind of behavior, determined by the state in which they are created. The creation state is determined by referring to an attribute object, an opaque data type that is initialized and configured under programmatic control.

Attribute object types exist for threads, mutexes, and condition variables. The object itself is supplied by the system; it cannot be directly modified by assignments. The attributes supported by each type are defined by pthreads and are invariant for an implementation. A set of functions is provided to initialize, configure, and destroy each object type.

Once an attribute is initialized and configured, it has process wide scope. The suggested method for using attributes is to configure all required state specifications one time in the early stages of program execution. The appropriate attribute object can then be referred to as needed by the pthreads entity creation function.

Subject	Overview	Details
Thread Attributes	*page C-167*	page C-175
Mutex Attributes	*page C-180*	*page C-196*
Condition Variable Attributes	*page C-181*	*page C-200*

Thread Attributes

Threads Attribute API

```
#include <pthread.h>
#include <sched.h>
int pthread_attr_init (pthread_attr_t *attr)
int pthread_attr_destroy (pthread_attr_t *attr)
int pthread_attr_setstacksize (pthread_attr_t *attr, size_t stacksize)
int pthread_attr_getstacksize (pthread_attr_t *attr, size_t *stacksize)
int pthread_attr_setstackaddr (pthread_attr_t *attr, size_t stackaddr)
int pthread_attr_getstackaddr (pthread_attr_t *attr, size_t *stackaddr)
int pthread_attr_setdetachstate (pthread_attr_t *attr, int detachstate)
int pthread_attr_getdetachstate (pthread_attr_t *attr, int *detachstate)
int pthread_attr_setscope (pthread_attr_t *attr, int scope)
int pthread_attr_getscope (pthread_attr_t *attr, int *scope)
int pthread_attr_setschedparam (pthread_attr_t *attr, const struct sched_param *param)
int pthread_attr_getschedparam (pthread_attr_t *attr, struct sched_param *param)
```

Attribute function	See section
Initialization	*Init/Destroy on page C-176*
Configure stack size	*Set/Get Stack Size on page C-177*
Configure stack address	*Set/Get Stack Address on page C-178*
Configure detach state	*Set/Get Detach State on page C-178*
Configure scope	*Set/Get Scope on page C-179*
Configure schedule policy	*Set/Get Schedule Policy on page C-179*

Init/Destroy: threads attribute initialization

pthread_attr_init()

initializes the attributes associated with this object to their default values. The storage is allocated by the thread system during execution.

pthread_attr_destroy()

removes that storage and the attribute object becomes invalid.

```
pthread_attr_t attr;
ret = pthread_attr_init (&attr);
     /* initialize an attribute to default value */
ret = pthread_attr_destroy (&attr);
     /* destroy an attribute */
```

The default values of attributes are:

scope — PTHREAD_SCOPE_PROCESS - new thread will be an unbound thread.

detachstate — PTHREAD_CREATE_JOINABLE - new thread will be non-detached.

stackaddr — NULL - new thread will have system allocated stack.

stacksize — NULL - new thread will have system defined size of the stack.

priority— NULL - new thread will inherit the parent thread's priority.

Set/Get Stack Size: Configure threads stack size attribute

The stacksize attribute defines the size of the stack in bytes the system will allocate. The size should not be less than PTHREAD_STACK_MIN.

```
pthread_attr_t attr;
int size;
size = (PTHREAD_STACK_MIN + 0x1000);
ret = pthread_attr_setstacksize (&attr, size);
     /* setting a new size */
ret = pthread_attr_getstacksize (&attr, &size);
     /* getting the stack size */
```

Set/Get Stack Address: Configure threads stack starting address attribute

The stackaddr attribute defines the base of thread's stack. If set to non-null (NULL is default) value system will initialize stack at that address.

```
pthread_attr_t attr;
void *base;
base = (void *) malloc (PTHREAD_STACK_MIN + 0x1000);
ret = pthread_attr_setstackaddr (&attr, base);
     /* setting a new address */
ret = pthread_attr_getstackaddr (&attr, base);
     /* getting the stack address */
```

Set/Get Detach State: Configure threads detach state

A detachstate of PTHREAD_CREATE_DETACHED signifies the same behavior as a THR_DETACH flag would have in thr_create () call, that is a detached thread. A detachstate of PTHREAD_CREATE_JOINABLE is equivalent to non-detached thread or a thread which can be joined upon its termination.

```
pthread_attr_t attr;
int detachstate;
ret = pthread_attr_getdetachstate (&attr, &detachstate);
     /* get detachstate of thread */
ret = pthread_attr_setdetachstate
       (&attr, PTHREAD_CREATE_DETACHED);
ret = pthread_attr_setdetachstate
       (&attr, PTHREAD_CREATE_JOINABLE);
```

Set/Get Scope: Configure threads scope attribute

A pthreads scope attribute set to PTHREAD_SCOPE_SYSTEM signifies the same behavior the THR_BOUND flag would have in a `thr_create` () call, that is a bound thread. If set to PTHREAD_SCOPE_PROCESS, an unbound thread will be created.

```
pthread_attr_t attr;
int scope;
ret = pthread_attr_getscope (&attr, &scope);
     /* get scope of thread */
ret = pthread_attr_setscope (&attr, PTHREAD_SCOPE_SYSTEM);
     /*bound thread */
ret = pthread_attr_setscope (&attr, PTHREAD_SCOPE_PROCESS);
     /*unbound thread */
```

Set/Get Schedule Policy: Configure threads scheduling attribute

Scheduling parameters are defined in the param structure. Currently, only priority is supported. The newly created thread will run with the configured priority.

```
pthread_attr_t attr;
int newprio;
sched_param param;
newprio = 30;
param.sched_priority = newprio;
     /* set the priority, others are unchanged */
ret = pthread_attr_setschedparam (&tattr, &param);
     /* setting the new scheduling param */
ret = pthread_attr_getschedparam (&tattr, &param);
     /* get existing scheduling param */
```

Mutex Attributes

```
#include <pthread.h>
int pthread_mutexattr_init (pthread_mutexattr_t *attr)
int pthread_mutexattr_destroy (pthread_mutexattr_t *attr)
int pthread_mutexattr_setpshared
     (pthread_mutexattr_t *attr, int pshared)
int pthread_mutexattr_getpshared
     (pthread_mutexattr_t *attr, int *pshared)
```

Attribute function	See Section:
Initialization	*Init/Destroy on page C-180*
Configure scope	*Set/Get Scope on page C-181*

Init/Destroy: Initialize a mutex attribute object

`pthread_mutexattr_init()` initializes the attributes associated with this object to their default values. The storage is allocated by the thread system during execution. `pthread_mutexattr_destroy()` removes that storage and the attribute object becomes invalid.

```
pthread_mutexattr_t mattr;
ret = pthread_mutexattr_init (&mattr);
     /* initialize an attribute to default value */
ret = pthread_mutexattr_destroy (&mattr);
     /* destroy an attribute */
```

The default values of attributes are:

pshared — PTHREAD_SHARE_PRIVATE - initialized mutex can be used within a process.

Set/Get Scope: Configure the scope of a mutex

The scope of a mutex variable can be either process private or system wide. If the mutex is created in the PTHREAD_SHARE_PROCESS state and it exists in shared memory, it can be shared among threads from more than one process. This is equivalent to the USYNC_PROCESS flag in `mutex_init` ().

```
pthread_mutexattr_t mattr;
int pshared;
ret = pthread_mutexattr_getpshared (&mattr, &pshared);
    /* get pshared of mutex */
ret = pthread_mutexattr_setpshared
     (&mattr, PTHREAD_SHARE_PROCESS);
ret = pthread_mutexattr_setpshared
     (&mattr, PTHREAD_SHARE_PRIVATE);
```

Condition Variable Attributes

Condition Variable Attribute API

```
#include <pthread.h>
int pthread_condattr_init (pthread_condattr_t *attr)
int pthread_condattr_destroy (pthread_condattr_t *attr)
int pthread_condattr_setpshared
     (pthread_condattr_t *attr, int pshared)
int pthread_condattr_getpshared
     (pthread_condattr_t *attr, int *pshared)
```

Attribute function	See section
Initialization	*Init/Destroy on page C-182*
Configure scope	*Set/Get Scope on page C-182*

Init/Destroy: Initialize a condition variable attribute object

`pthread_condattr_init()` initializes the attributes associated with this object to their default values. The storage is allocated by the thread system during execution. `pthread_condattr_destroy()` removes that storage and the attribute object becomes invalid.

The default values of attributes are:

pshared — PTHREAD_SHARE_PRIVATE - initialized condition variable can be used within a process.

```
pthread_condattr_t cattr;
ret = pthread_condattr_init (&cattr);
     /* initialize an attribute to default value */
ret = pthread_condattr_destroy (&cattr);
     /* destroy an attribute */
```

Set/Get Scope: Configure the scope of a condition variable

The attribute pshared defines the synchronization scope of the condition variable initialized with this attribute object. PTHREAD_SHARE_PRIVATE results in the same behavior that the USYNC_THREAD flag would have in a `cond_init` () call, that is a local condition variable. PTHREAD_SHARE_PROCESS is equivalent to a global condition variable.

```
pthread_condattr_t cattr;
int pshared;
ret = pthread_condattr_getpshared (&cattr, &pshared);
     /* get pshared of cond */
ret = pthread_condattr_setpshared (
     &cattr, PTHREAD_SHARE_PROCESS); /* all processes */
ret = pthread_condattr_setpshared (
     &cattr, PTHREAD_SHARE_PRIVATE); /* within a process */
```

Thread Functions

Thread functions are defined in the POSIX.4a (1003.4a) specifications. These functions are standard and similar to those provided in the Solaris threads implementation. Changing from Solaris threads to pthreads is as trivial as changing *thr_* prefix to *pthread_* in most of the cases.

Thread Creation

Solaris API

```
#include <thread.h>
int thr_create (
    void * stkaddr, size_t stksize, void *(*func) (void *),
   (void *arg), long flags, thread_t *tid)
```

pthreads API

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

The use of attribute objects to specify the state in which a new thread is created is the primary difference between the Solaris and the pthreads thread creation interface. In general an attribute is initialized and then set to reflect the desired thread state. When the thread is created, a reference is made to the attribute object with the desired state specification.

In the examples below, a new attribute object is initialized and set for each case. In practice, a group of attribute objects - one for each needed state behavior - should be set up one time and then referenced as needed.

In Table C-2, find the thr_create argument you need. See the section listed in the last column for the attribute object to use.

Table C-2 Thread Creation Attributes

stkaddr	stksize	func	arg	flag	tid	See section
NULL	NULL			NULL		***default*** *on page C-184*
				THR_BOUND		***bound*** *on page C-184*
				THR_DETACHED		***detached*** *on page C-184*
NULL	size					***stksize*** *on page C-185*
stackbase	NULL					***stkbase*** *on page C-185*
stackbase	size					***stksize&base*** *on page C-186*

default: Creating a thread with all default values

A default thread is created unbound, non-detached, with default stack and stack size, and it inherits the parent's priority. Creating a thread using a NULL attribute argument has the same effect as using a default attribute. Both will create a default thread. When 'tattr' is initialized, it acquires the default behavior.

```
pthread_attr_t tattr;
pthread_t tid;
int ret;
ret = pthread_create (&tid, NULL, func, arg);
      /* default behavior*/
OR
ret = pthread_attr_init (&tattr);
      /* initialized with default attributes */
ret = pthread_create (&tid, &tattr, func, arg);
      /* default behavior*
```

bound: Creating a bound thread

```
pthread_attr_t tattr;
pthread_t tid;
int ret;
ret = pthread_attr_init (&tattr);
      /* initialized with default attributes */
ret = pthread_attr_setscope (&tattr, PTHREAD_SCOPE_SYSTEM);
      /* BOUND behavior */
ret = pthread_create (&tid, &tattr, func, arg);
```

detached: Creating a detached thread

```
pthread_attr_t tattr;
pthread_t tid;
int ret;
ret = pthread_attr_init (&tattr); /* initialized with default
attributes */
ret = pthread_attr_setscope (&tattr, PTHREAD_SCOPE_SYSTEM); /*
BOUND behavior */
ret = pthread_create (&tid, &tattr, func, arg);
```

stksize: Creating a thread with a custom stack size

```
pthread_attr_t tattr;
pthread_t tid;
int ret;
void *stackbase;
int size = PTHREAD_MIN_STACK + 0x1000;
ret = pthread_attr_init (&tattr);
     /* initialized with default attributes */
ret = pthread_attr_setstacksize (&tattr, size);
     /* setting the size of the stack also */
ret = pthread_create (&tid, &tattr, func, arg);
     /* only size specified */
```

stkbase: Creating a thread with a custom stack starting address

```
pthread_attr_t tattr;
pthread_t tid;
int ret;
void *stackbase;
stackbase = (void *) malloc (size);
ret = pthread_attr_init (&tattr);
     /* initialized with default attributes */
ret = pthread_attr_setstackaddr (&tattr, stackbase);
     /* setting the base address in the attribute */
ret = pthread_create (&tid, &tattr, func, arg);
     /* only address specified */
```

stksize&base: Creating a thread with a custom stack address & stack size

```
pthread_attr_t tattr;
pthread_t tid;
int ret;
void *stackbase;
int size = PTHREAD_MIN_STACK + 0x1000;
stackbase = (void *) malloc (size);
ret = pthread_attr_init (&tattr);
     /* initialized with default attributes */
ret = pthread_attr_setstacksize (&tattr, size);
     /* setting the size of the stack also */
ret = pthread_attr_setstackaddr (&tattr, stackbase);
     /* setting the base address in the attribute */
ret = pthread_create (&tid, &tattr, func, arg);
     /*address and size specified */
```

Managing Threads Priorities

Solaris API

```
#include <thread.h>
int thr_getprio (thread_t tid, int *prio)
int thr_setprio (thread_t tid, int prio)
```

pthreads API

```
#include <pthread.h>
#include <sched.h>
int pthread_getschedparam (
     thread_t tid, int *policy, sched_param *param)
int pthread_setschedparam (
     thread_t tid, int policy, sched_param *param)
```

Creating a thread with a specified priority

In Solaris

In Solaris threads, if a thread is to be created with a priority other than its parents, it is created in SUSPEND mode. While suspended, the threads priority is modified using the `thr_setprio()` function call; then it is continued.

```
thread_t tid;
int ret;
int newprio = 20;
ret = thr_create (NULL, NULL, func, arg, THR_SUSPEND, &tid);
    /* suspended thread creation */
ret = thr_setprio (tid, newprio);
    /* set the new priority of suspended child thread */
ret = thr_continue (tid);
    /* suspended child thread starts executing with new priority
*/
```

In pthreads

In pthreads, the user can set the priority attribute before creating the thread. The child thread is created with the new priority specified in the sched_param structure. The sched_param structure contains other scheduling information. It is always a good idea to get the existing parameters, change the priority and set it again.

```
#include <sched.h>
pthread_attr_t tattr;
pthread_t tid;
int ret;
int newprio = 20;
sched_param param;
ret = pthread_attr_init (&tattr);
     /* initialized with default attributes */
ret = pthread_attr_getschedparam (&tattr, &param);
     /* safe to get existing scheduling param */
param.sched_priority = newprio;
     /* set the priority, others are unchanged */
ret = pthread_attr_setschedparam (&tattr, &param);
     /* setting the new scheduling param */
ret = pthread_create (&tid, &tattr, func, arg);
     /* with new priority specified */
```

Another way of creating a thread with a priority different from its parent is to change the parent thread's priority before thread create (so that it inherits the new priority) and then restore the parents's priority afterwards. Getting and setting the priority of an existing thread is described below.

Modifying the priority of an existing thread

Two pthreads functions are provided to deal with pthreads priority. `pthread_getschedparam()` gets the policy of the target thread and its associated scheduling parameters. `pthread_setschedparam()` sets the policy of the target thread and its associated scheduling parameters. In the current implementation only the SCHED_OTHER scheduling policy is supported; the only scheduling parameter is priority.

Getting priority

In Solaris

```
thread_t tid;
int ret;
int priority;
ret = thr_getprio (tid, &priority);
     /* returns priority of target thread tid */
```

In pthreads

```
pthread_t tid;
int ret;
sched_param param;
int priority;
int policy;
ret = pthread_getschedparam (tid, &policy, &param);
     /* scheduling parameters of target thread */
priority = schedparam.sched_priority;
     /* sched_priority contains the priority of the thread */
```

Setting priority

In Solaris

```
thread_t tid;
int ret;
int priority;
ret = thr_setprio (tid, priority);
     /* returns priority of target thread tid */
```

In pthreads

```
pthread_t tid;
int ret;
sched_param param;
int priority;
schedparam.sched_priority = priority;
     /* sched_priority will be the priority of the thread */
policy = SCHED_OTHER;
     /* only supported policy, others will result in ENOTSUP */
ret = pthread_setschedparam (tid, policy, param);
     /* scheduling parameters of target thread */
```

Thread Join

Solaris API

```
#include <thread.h>
int thr_join (thread_t tid, thread_t *departedid, int *status)
```

pthreads API

```
#include <pthread.h>
int pthread_join (thread_t tid, int *status)
```

Join specific

In Solaris

```
thread_t tid;
thread_t departedid;
int ret;
int status;
ret = thr_join (tid, &departedid, &status);
    /* waiting to join thread "tid" with status */
ret = thr_join (tid, &departedid, NULL);
    /* waiting to join thread "tid" without status */
ret = thr_join (tid, NULL, NULL);
   /* waiting to join thread "tid" without return id and status */
```

In pthreads

In pthreads, there is no concept of returning the thread id of the terminated thread.

```
pthread_t tid;
int ret;
int status;
ret = pthread_join (tid, &status);
     /* waiting to join thread "tid" with status */
ret = pthread_join (tid, NULL);
     /* waiting to join thread "tid" without status */
```

Join any

In Solaris

```
thread_t tid;
thread_t departedid;
int ret;
int status;
ret = thr_join (NULL, &departedid, &status);
     /* waiting to join thread "tid" with status */
```

In pthreads

By indicating NULL as thread id in the Solaris `thr_join()`, a join will take place when any non detached thread in the process exits. The *departedid* will indicate the thread id of exiting thread.

In pthreads, there is no provision for this functionality. Using NULL as a thread id in `pthread_join()` will result in invalid argument (EINVAL) error. There is no *departedid* argument which can be used to return an exited thread id.

Thread Specific Data

Solaris API

```
#include <thread.h>
int thr_keycreate (
     thread_key_t *keyp, void (*destructor) (void *))
int thr_setspecific (thread_key_t key, void *value)
int thr_getspecific (thread_key_t key, void **value)
```

pthreads API

```
#include <pthread.h>
int pthread_key_create (
     pthread_key_t *keyp, void (*destructor) (void *))
int pthread_setspecific (pthread_key_t key, void *value)
void *pthread_getspecific (pthread_key_t key)
int pthread_key_delete (pthread_key_t key)
```

Key create

In Solaris

```
thread_key_t mykey;
int ret;
ret = thr_keycreate (&mykey, NULL);
     /* key create without destructor */
ret = thr_keycreate (&mykey, dest_func);
     /* key create with destructor */
```

In pthreads

```
pthread_key_t mykey;
int ret;
ret = pthread_key_create (&mykey, NULL);
     /* key create without destructor */
ret = pthread_key_create (&mykey, dest_func);
     /* key create with destructor */
```

TSD get

In Solaris

```
thread_key_t mykey;
void *val;
int ret;
ret = thr_getspecific (mykey, &val);
     /* mykey is previously created key */
```

In pthreads

```
pthread_key_t mykey;
void *val;
val = pthread_getspecific (mykey);
     /* mykey is previously created key */
```

TSD set

In Solaris

```
thread_key_t mykey;
void *val;
int ret;
ret = thr_setspecific (mykey, val);
     /* mykey is previously created key */
```

In pthreads

```
pthread_key_t mykey;
void *val;
int ret;
ret = pthread_setspecific (mykey, val);
     /* mykey is previously created key */
```

Key delete

pthreads provides a means to destroy an existing thread specific data key. This can be used to cause an error return when trying to access some thread specific data set which is no longer valid. There is no comparable function in Solaris.

Once a key has been deleted any reference to it via `pthread_setspecific()` or `pthread_getspecific()` calls result in EINVAL error. It is the responsibility of the programmer to free any thread specific resources prior to calling delete function. This function does not invoke any of the destructors.

```
pthread_key_t mykey;
int ret;
ret = pthread_key_delete (mykey);
/* mykey is previously created key */
```

Other thread functions

Solaris API

```
#include <thread.h>
#include <signal.h>
void thr_exit (int *status)
thread_t thr_self ()
int thr_sigsetmask (int how, const sigset_t *new, sigset_t *old)
int thr_yield ()
int thr_kill (thread_t, int sig)
```

pthreads API

```
#include <pthread.h>
#include <signal.h>
void pthread_exit (int *status)
thread_t thread_self ()
int pthread_sigmask (int how, const sigset_t *newp, sigset_t
*oldp)
int sched_yield ()   /* POSIX.4 */
int pthread_kill (pthread_t, int sig)
```

Thread exit

In Solaris

```
int status;
thr_exit (&status); /* exit with status */
```

In pthreads

```
int status;
pthread_exit (&status); /* exit with status */
```

Thread identification

In Solaris

```
thread_t tid;
tid = thr_self();
```

In pthreads

```
pthread_t tid;
tid = pthread_self();
```

Setting a thread signal mask

In Solaris

```
int ret;
sigset_t old, new;
ret = thr_sigsetmask (SIG_SETMASK, &new, &old);
    /* setting a new mask */
ret = thr_sigsetmask (SIG_BLOCK, &new, &old); /* blocking mask */
re = thr_sigsetmask (SIG_UNBLCOK, &new, &old);
     /* unblocking mask */
```

In pthreads

```
int ret;
sigset_t old, new;
ret = pthread_sigmask (SIG_SETMASK, &new, &old);
     /* setting a new mask */
ret = pthread_sigmask (SIG_BLOCK, &new, &old);
     /* blocking mask */
ret = pthread_sigmask (SIG_UNBLCOK, &new, &old);
     /* unblocking mask */
```

Yielding a thread run status

In Solaris

```
thr_yield();
```

In pthreads

```
sched_yield(); /* POSIX.4 */
```

Killing a thread

In Solaris

```
int sig;
thread_t tid;
thr_kill (tid, sig); /* kill target thread with sig signal */
```

In pthreads

```
int sig;
pthread_t tid;
pthread_kill (tid, sig); /* kill target thread with sig signal */
```

Mutex Functions

pthreads defines the use of mutex synchronization variables. Changing from Solaris to pthreads is as trivial as adding the prefix *pthread_*. All the mutex functions have similar arguments except for mutex initialization.

Solaris API

```
#include <synch.h>
int mutex_init (mutex_t *mp, int type, void *arg)
int mutex_destroy (mutex_t *mp)
int mutex_lock (mutex_t *mp)
int mutex_unlock (mutex_t *mp)
int mutex_trylock (mutex_t *mp)
```

pthreads API

```
#include <pthread.h>
int pthread_mutex_init (
     pthread_mutex_t *mp, pthread_mutexattr_t *attr)
int pthread_mutex_destroy (pthread_mutex_t *mp)
int pthread_mutex_lock (pthread_mutex_t *mp)
int pthread_mutex_unlock (pthread_mutex_t *mp)
int pthread_mutex_trylock (pthread_mutex_t *mp)
```

Initialization of a mutex with *intraprocess* scope

In Solaris

```
mutex_t mp;
int ret;
ret = mutex_init (&mp, USYNC_THREAD, 0);
     /* to be used within this process only */
```

In pthreads

If a mutex variable is to be shared within a process, it can be initialized in two different ways. One way is to call the init function with attr as NULL. The second is to use an attribute object *mattr* which has been initialized with the default value of the pshared attribute (PTHREAD_PROCESS_PRIVATE).

It is also possible to explicitly set the pshared attribute to PTHREAD_PROCESS_PRIVATE in a previously initialized mattr.

These states are equivalent to the USYNC_THREAD state in Solaris threads.

```
pthread_mutex_t mp;
int ret;
pthread_mutexattr_t mattr;
ret = pthread_ mutex_init (&mp, NULL);
   /* Using a NULL argument to specify the default creation state */
OR
ret = pthread_mutexattr_init (&mattr);
ret = pthread_ mutex_init (&mp, &mattr);
   /* Using an attribute object initialized and left in the default state */
OR
ret = pthread_mutexattr_setpshared (&mattr, PTHREAD_PROCESS_PRIVATE);
ret = pthread_mutex_init (&mp, &mattr);
   /* Explicitly setting the attribute object to indicate PTHREAD_PROCESS_PRIVATE */
```

Initialization of a Mutex with *inter*-process scope

In Solaris

```
mutex_t mp;
int ret;
ret = mutex_init (&mp, USYNC_PROCESS, 0);
     /* to be used among all processes */
```

In pthreads

If a mutex variable is to be shared among processes it is initialized using the attribute object *mattr* which has been set with the value PTHREAD_PROCESS_PROCESS. This is equivalent to the USYNC_PROCESS state in Solaris threads.

```
mutex_t mp;
int ret;
pthread_mutexattr_t mattr;
ret = pthread_mutexattr_init (&mattr);
ret = pthread_mutexattr_setpshared (&mattr,
PTHREAD_PROCESS_SHARED);
ret = pthread_mutex_init (&mp, &mattr);
     /* to be used among all processes */
```

Destroying a Mutex

In Solaris

```
mutex_t mp;
int ret;
ret = mutex_destroy(&mp); /* Mutex is destroyed */
```

In pthreads

```
mutex_t mp;
int ret;
ret = pthread_ mutex_destroy(&mp); /* Mutex is destroyed */
```

Mutex lock

In Solaris

```
mutex_t mp;
int ret;
ret = mutex_lock (&mp); /* acquire the Mutex */
```

In pthreads

```
pthread_mutex_t mp;
int ret;
ret = pthread_ mutex_lock (&mp); /* acquire the Mutex */
```

Mutex unlock

In Solaris

```
mutex_t mp;
int ret;
ret = mutex_unlock (&mp); /* release the Mutex */
```

In pthre1ads

```
pthread_mutex_t mp;
int ret;
ret = pthread_ mutex_unlock (&mp); /* release the Mutex */
```

Mutex trylock

In Solaris

```
mutex_t mp;
int ret;
ret = mutex_trylock (&mp); /* try to grab Mutex */
```

In pthreads

```
pthread_mutex_t mp;
int ret;
ret = pthread_ mutex_trylock (&mp); /* try to grab Mutex */
```

Condition Variable Operations

pthreads condition variables are standard and similar to those provided in the Solaris thread implementation. Changing from Solaris to pthreads is as trivial as adding the prefix *pthread_*. All the mutex functions have similar arguments except for mutex initialization.

Solaris API

```
#include <synch.h>
int cond_init (cond_t *cv, int type, int arg)
int cond_destroy (cond_t *cv)
int cond_wait (cond_t *cv, mutex_t *mp)
int cond_timedwait (cond_t *cv, mutex_t *mp, timestruct_t
abstime)
int cond_signal (cond_t *cv)
int cond_broadcast (cond_t *cv)
```

pthreads API

```
#include <pthread.h>
#include <time.h>
int pthread_cond_init (
    pthread_cond_t *cv, pthread_condattr_t *attr)
int pthread_cond_destroy (pthread_cond_t *cv)
int pthread_cond_wait (pthread_cond_t *cv, pthread_mutex_t *mp)
int pthread_cond_timedwait (
    pthread_cond_t *cv, pthread_mutex_t *mp, timestruct_t
abstime)
int pthread_cond_signal (pthread_cond_t *cv)
int pthread_cond_broadcast (pthread_cond_t *cv)
```

Initialization of a condition variable with *intraprocess* scope

In Solaris

```
cond_t cv;
int ret;
ret = cond_init (cv, USYNC_THREAD, 0);
     /* to be used within this process only */
```

In pthreads

If a condition variable is to be shared within a process, it can be initialized in two different ways. One way is to call the init function with attr as NULL. The second is to use an attribute object *cattr* which has been initialized with the default value of the pshared attribute (PTHREAD_PROCESS_PRIVATE).

It is also possible to explicitly set the pshared attribute to PTHREAD_PROCESS_PRIVATE in a previously initialized cattr.

```
pthread_cond_t cv;
int ret;
pthread_condattr_t cattr;
ret = pthread_ cond_init (&cv, NULL);
 /* Using a NULL argument to specify the default creation state */
OR
ret = pthread_condattr_t (&cattr);
ret = pthread_ cond_init (&cv, &cattr);
  /* Using an attribute object initialized and left in the default
       state */
OR
ret = pthread_condattr_setpshared (&cattr,
PTHREAD_PROCESS_PRIVATE);
ret = pthread_cond_init (&cv, &cattr);
   /* Explicitly setting the attribute object to indicate
      PTHREAD_PROCESS_PRIVATE */
```

Initialization of a condition variable with *inter*-process scope
In Solaris

```
cond_t cv;
int ret;
ret = cond_init (&cv, USYNC_PROCESS, 0);
     /* to be used among all processes */
```

In pthread

If a condition variable is to be shared among processes it is initialized using the attribute object *cattr* which has been set with the value PTHREAD_PROCESS_PROCESS. This is equivalent to the USYNC_PROCESS state in Solaris threads.

```
pthread_cond_t cv;
int ret;
pthread_condattr_t cattr;
ret = pthread_condattr_init (&cattr);
ret = pthread_condattr_setpshared (&cattr,
PTHREAD_PROCESS_SHARED);
ret = pthread_cond_init (&cv, &cattr);
     /* to be used among all processes */
```

Destroying a condition variable

In Solaris

```
cond_t cv;
int ret;
ret = cond_destroy (&cv); /* Condition variable is destroyed */
```

In pthreads

```
pthread_cond_t cv;
int ret;
ret =pthread_ cond_destroy (&cv);
     /* Condition variable is destroyed */
```

Condition variable wait

In Solaris

```
cond_t cv;
mutex_t mp;
int ret;
ret = cond_wait (&cv, &mp); /* wait on condition variable */
```

In pthreads

```
pthread_cond_t cv;
pthread_mutex_t mp;
int ret;
ret = pthread_cond_wait (&cv, &mp);
     /* wait on condition variable */
```

Condition variable timedwait

In Solaris

```
cond_t cv;
mutex_t mp;
timestruct_t abstime;
int ret;
ret = cond_timedwait (&cv, &mp, &abstime);
     /* wait on condition variable */
```

In pthreads

```
pthread_cond_t cv;
pthread_mutex_t mp;
timestruct_t abstime;
int ret;
ret = pthread_cond_timedwait (&cv, &mp, &abstime);
     /* wait on condition variable */
```

Condition variable signal

In Solaris

```
cond_t cv;
int ret;
ret = cond_signal (&cv); /* one condition variable is signaled */
```

In pthreads

```
pthread_cond_t cv;
int ret;
ret = pthread_cond_signal (&cv);
     /* one condition variable is signaled */
```

Condition variable broadcast

In Solaris

```
cond_t cv;
int ret;
ret = cond_broadcast(&cv);
     /* all condition variables are signaled */
```

In pthreads

```
pthread_cond_t cv;
int ret;
ret = pthread_cond_broadcast(&cv);
     /* all condition variables are signaled */
```

Semaphore Operations

Semaphores are not defined in the POSIX.4a (pthreads) specifications, but they are included in the POSIX.4 (real-time extensions) specifications. These functions are standard and similar to those provided in Solaris thread implementation. Changing from Solaris to the POSIX.4 standard is as trivial as converting *sema* prefix to *sem*.

Solaris API

```
#include <synch.h>
int sema_init (
    sema_t *sp, unsigned int count, int type, void *arg)
int sema_destroy (sema_t *sp)
int sema_wait (sema_t *sp)
int sema_trywait (sema_t *sp)
int sema_post (sema_t *sp)
```

POSIX.4 API

```
#include <semaphore.h>
int sem_init (sem_t *sp, int pshared, int value)
int sem_destroy (sem_t *sp)
int sem_wait (sem_t *sp)
int sem_trywait (sem_t *sp)
int sem_post (sem_t *sp)
```

Initialization of a semaphore within *intraprocess* scope

In Solaris

```
sema_t sp;
int ret;
int count;
count = 4;
ret = sema_init (&sp, count, USYNC_THREAD, 0);
     /* to be used within this process only */
```

In POSIX.4

In pthreads, if pshared is 0 then semaphore can be used by all the threads in a process.

```
sem_t sp;
int ret;
int count = 4;
ret = sem_init (&sp, 0, count);
     /* to be used within this process only */
```

Initialization of a semaphore within *inter*-process scope

In Solaris -

```
sema_t sp;
int ret;
int count;
count = 4;
ret = sema_init (&sp, count, USYNC_PROCESS, 0);
     /* to be used among all the processes */
```

In POSIX.4 -

```
sem_t sp;
int ret;
int count = 4;
ret = sem_init (&sp, 1, count);
     /* to be used among all the processes */
```

In pthreads, if pshared is nonzero then a semaphore can be used by all the processes.

Destroying a semaphore

In Solaris -

```
sema_t sp;
int ret;
ret = sema_destroy (&sp); /*semaphore is destroyed */
```

In POSIX.4 -

```
sem_t sp;
int ret;
ret = sem_destroy (&sp); /* semaphore is destroyed */
```

Semaphore wait

In Solaris -

```
sema_t sp;
int ret;
ret = sema_destroy (&sp); /*wait for semaphore */
```

In POSIX.4 -

```
sem_t sp;
int ret;
ret = sem_wait (&sp); /* wait for semaphore */
```

Semaphore trywait

In Solaris -

```
sema_t sp;
int ret;
ret = sema_trywait (&sp); /*try to wait for semaphore */
```

In POSIX.4 -

```
sem_t sp;
int ret;
ret = sem_trywait (&sp); /* try to wait for semaphore*/
```

Semaphore post

In Solaris -

```
sema_t sp;
int ret;
ret = sema_post (&sp); /*semaphore is posted */
```

In POSIX.4 -

```
sem_t sp;
int ret;
ret = sem_post (&sp); /* semaphore is posted */
```

New pthreads Functions

Cancellation

Solaris API

This functionality is not supported in the Solaris threads.
In pthreads

A cancellation is effected by calling the function `pthread_cancel()` with a thread ID as an argument. How the cancellation request is treated depends on the state of the target thread. Two functions, `pthread_setcancelstate()` and `pthread_setcanceltype()`, determine that state as follows:

- `pthread_setcancelstate()` sets the calling thread state to PTHREAD_CANCEL_DISABLE or PTHREAD_CANCEL_ENABLE.
  - If the cancellation state of the target is PTHREAD_CANCEL_DISABLE, then all cancellation requests to that target are held until an enable state is reached.
  - If the state is set to PTHREAD_CANCEL_ENABLE, then the cancellation behavior depends on the cancel type.
- `pthread_setcanceltype()` can set the calling thread's state to PTHREAD_CANCEL_DEFERRED or to PTHREAD_CANCEL_ASYNCHRONOUS.
  - If the cancel type is PTHREAD_CANCEL_DEFERRED, then cancellation will not occur until the thread reaches a cancellation point.
  - If the cancel type is PTHREAD_CANCEL_ASYNCHRONOUS, then receipt of a `pthread_cancel()` call will result in an immediate cancellation.

Table C-3, “Effect of Cancellation Options” shows which combination of type and state result in which option.

Table C-3 Effect of Cancellation Options

	State: Enabled	State: Disabled
Type: Deferred	Hold until cancel point is reached, or until type is changed to asynchronous. Terminate at cancel point.	all cancellation requests held pending: a) a cancel point b) a type change to asynchronous
Type: Asynchronous	Terminate immediately	wait for enable

pthreads API

```
#include <pthread.h>
int pthread_cancel (pthread_t tid)
int pthread_setcanceltype (int type, int *oldtype)
int pthread_setcancelstate (int state, int *oldstate)
void pthread_cleanup_push (void (*routine) (void *), void *arg)
void pthread_cleanup_pop (int execute)
void pthread_testcancel()
```

Cancelling a thread

```
pthread_t tid;
int ret;
ret = pthread_cancel (tid);
```

Setting cancellation type of current thread

The cancellation type can be set to either deferred or in async mode. By default when a thread is created, the cancellation type is set to deferred mode. In deferred mode, the thread can be cancelled only at cancellation points. In async mode, a thread can be cancelled any point during its execution. Use of async mode is discouraged.

```
int oldtype;
int ret;
ret = pthread_setcanceltype (PTHREAD_CANCEL_DEFERED, &oldtype);
/* deferred mode */
ret = pthread_setcanceltype (PTHREAD_CANCEL_ASYNCHRONOUS,
&oldtype); /* async mode*/
```

Setting cancellation state of current thread

Cancellability of a thread can be enabled or disabled. By default when a thread is created, cancellability is enabled.

```
int oldstate;
int ret;
ret = pthread_setcancelstate (PTHREAD_CANCEL_ENABLE, &oldstate);
     /* enabled */
ret = pthread_setcancelstate (
     PTHREAD_CANCEL_DISABLE, &oldstate);
     /* disabled */
```

Cleanup handlers

Cleanup handlers are pushed and popped in same lexical scope of a program. They should always match; otherwise compiler errors will be generated. A nonzero argument in the pop function will remove the handler from the stack and execute it. If it is zero, then the handler is popped out without executing it.

```
ret = pthread_cleanup_push (func, arg);
     /* push the handler "func" on cleanup stack */
ret = pthread_cleanup_pop (1);
    /* pop the "func" out of cleanup stack and execute "func" */
ret = pthread_cleanup_pop (0);
     /* pop the "func" and DONT execute "func" */
```

Testing for a pending cancellation request

pthread_testcancel() provides a cancellation point during execution of a thread. It should only be inserted in sequences where it is safe to cancel a thread. It is effective when cancellability is enabled and in deferred mode. Calling this function while cancellability is disabled has no effect.

```
pthread_testcancel();
```

Thread related functions

Solaris API

These functions are not supported in the Solaris threads.

pthreads API

```
#include <pthread.h>
int pthread_once (pthread_once_t *once, void (*func) (void))
int pthread_equal (pthread_t tid1, pthread_t tid2)
```

Calling a function once in multithreaded process

A function can be called through *pthread_once*() to assure that it is executed one time only in a multithreaded program. This is useful where a particular operation such as initialization of a mutex or creating a key should be done once in the application. The *once_control* structure needs to be initialized using PTHREAD_ONCE_INIT.

```
pthread_once_t once_control = PTHREAD_ONCE_INIT;
int ret;
ret = pthread_once (&once_control, func);
```

Comparing two thread ids

```
pthread_t tid1, tid2
int ret;
ret = pthread_equal (tid1, tid2);
```

Functions in Solaris threads but NOT in pthreads

Note – Users are free to interleave pthreads functions with Solaris functions. No adverse effects will result from this mixing. This compatibility can be used to supplement Solaris threads functionality with pthreads functionality, or vice versa.

Thread related functions

Solaris API

```
#include <thread.h>
int thr_suspend (thread_t tid)
int thr_continue (thread_t tid)
int thr_setconcurrency (int new_level)
int thr_getconcurrency (void)
```

Suspending a thread

thr_suspend() suspends the specified thread. The target thread blocks until a thr_continue is called. Signals can not awaken the suspended thread, they remain pending until thread resume the execution. Calling this function for a suspended thread will have no effect.

pthread_t tid as defined in pthreads is the same as *thread_t tid* in Solaris threads. *tid* values can be used interchangeably either by assignment or through the use of casts.

```
thread_t tid; /* tid from thr_create()*/
pthread_t ptid;
     /* pthreads equivalent of Solaris tid from thread created */
     /* with pthread_create()*/
int ret;
ret = thr_suspend (tid);
ret = thr_suspend ((thread_t) ptid);
     /* using pthreads ID variable with a cast */
```

Continuing a suspended thread

thr_continue() resumes the execution of the specified suspended thread. Calling this function for a non-suspended thread will have no effect.

pthread_t tid as defined in pthreads is the same as *thread_t tid* in Solaris threads. *tid* values can be used interchangeably either by assignment of through the use of casts.

```
thread_t tid; /* tid from thr_create()*/
pthread_t ptid;
     /* pthreads equivalent of Solaris tid from thread created */
     /* with pthread_create()*/
int ret;
ret = thr_continue (tid);
ret = thr_continue ((thread_t) ptid)
     /* using pthreads ID variable with a cast */
```

Setting thread concurrency

thr_setconcurrency() provides a hint to the system about the required level of concurrency in the application. The system ensures that a sufficient number of threads are active so that the process continues to make progress.

```
int level;
int ret;
level = 5;
ret = thr_setconcurrency (level);
```

Getting thread concurrency

thr_getconcurrency() returns the current level of concurrency in the process. This number can be used to determine the desired level of concurrency.

```
int level;
level = thr_getconcurrency (void);
```

Readers/Writer Locks

Readers/writer locks allow more than one thread at time to read a variable; using this type of lock, only one thread at a time can access the variable to modify it.

Solaris API

```
#include <synch.h>
int rwlock_init (rwlock_t *rwlp, int type, void *arg)
int rwlock_destroy (rwlock_t *rwlp)
int rw_rdlock (rwlock_t *rwlp)
int rw_wrlock (rwlock_t *rwlp)
int rw_unlock (rwlock_t *rwlp)
int rw_tryrdlock (rwlock_t *rwlp)
int rw_trywrlock (rwlock_t *rwlp)
```

Initialization of a readers/writer lock with *intraprocess* scope

```
rwlock_t rwlp;
int ret;
ret = rwlock_init (&rwlp, USYNC_THREAD, 0);
     /* to be used within this process only */
```

Initialization of a readers/writer lock with *inter*-process scope

```
rwlock_t rwlp;
int ret;
ret = rwlock_init (&rwlp, USYNC_PROCESS, 0);
     /* to be used among all processes */
```

Destroying a readers/writer lock

```
rwlock_t rwlp;
int ret;
ret = rwlock_destroy(&rwlp); /* rwlock is destroyed */
```

Readers/writer lock

```
rwlock_t rwlp;
int ret;
ret = rw_rdlock (&rwlp); /* acquire the rwlock for reading */
ret = rw_wrlock (&rwlp); /* acquire the rwlock for writing */
```

Readers/writer unlock

```
rwlock_t rwlp;
int ret;
ret = rw_unlock (&rwlp); /* release the rwlock */
```

Readers/writer trylock

```
rwlock_t rwlp;
int ret;
ret = rw_tryrdlock (&rwlp); /* try to grab rwlock for reading */
ret = rw_trywrlock (&rwlp); /* try to grab rwlock for writing */
```

Process creation (fork)

It is important to note that the behavior of the *fork()* function call in pthreads is the same as fork1() in Solaris threads. Both the *fork()* call in pthreads and the *fork1()* call in Solaris creates a new process, duplicating only the calling thread in the child process.

If *fork()* is called in Solaris threads, the complete process including all threads is duplicated in the child process. No such functionality exists in pthreads.

To build an application using the pthreads *fork()*, the library libpthread.so should be included during the link phase (i.e. *-lpthread*). Current Solaris threads users who want to move to the pthreads API can replace *fork1()* with *fork()*; if they are using *fork()* (in the Solaris context) they will have to revise their applications because there is no equivalent to *fork()* in pthreads.

Fork API - All threads

In Solaris -

```
pid_t pid;
pid = fork ();
```

In pthreads -

Not supported.

Fork API - Calling thread only

In Solaris -

```
pid_t pid;
pid = fork1 ();
```

In pthreads -

```
pid_t pid;
pid = fork ();
```

Building applications

Table C-4 Use this table to determine what header information and shared libraries are necessary to build an application using pthreads or Solaris entities.

Function	Build Info.	Solaris thread	POSIX pthreads
threads	Include File	thread.h	pthread.h
	Library to link	libthread.so	libpthread.so
mutexes	Include File	synch.h	pthread.h
	Library to link	libthread.so	libpthread.so
condition variables	Include File	synch.h	pthread.h
	Library to link	libthread.so	libpthread.so
semaphores	Include File	synch.h	semaphore.h
	Library to link	libthread.so	libposix4.so

Linking with only pthreads functions

To use only pthreads calls, link programs as follows:

```
cc -D_RENTRANT foo.c -lpthread
```

Building applications with pthreads functions and semaphores

To use pthreads and POSIX semaphores, link as follows:

```
cc -D_RENTRANT foo.c -lposix4 -lpthread
```

Building applications with Solaris threads

To use Solaris threads, link as follows:

```
cc -D_RENTRANT foo.c -lthread
```

Building applications with Solaris threads and pthreads

To use Solaris threads and pthreads function calls in the same application, links should be made as follows (see Note below):

```
cc -D_RENTRANT foo.c -lposix4 -lthread
    OR
cc -D_RENTRANT foo.c -lposix4 -lpthread
```

Note – Linking with libthread or libpthread provides you with all the interfaces of pthreads as well as Solaris threads. The only difference is in the definition of the fork() call. If the program is linked with libpthread, the behavior of fork() is same as described in the POSIX.4a specification document. In other words, fork() in pthreads behaves like fork1() in Solaris thread. There is no interface in pthreads which is equivalent of fork() in Solaris.