Programming with Solaris Threads 9

This chapter compares the APIs for Solaris threads and POSIX threads, and explains the Solaris features that are not found in POSIX threads.

Comparing APIs for Solaris Threads and POSIX Threads

The Solaris threads API and the pthreads API are two solutions to the same problem: building parallelism into application software. Although each API is complete in itself, you can safely mix Solaris threads functions and pthread functions in the same program.

The two APIs do not match exactly, however. Solaris threads supports functions that are not found in pthreads, and pthreads includes functions that are not supported in the Solaris interface. For those functions that *do* match, the associated arguments might not, although the information content is effectively the same.

By combining the two APIs, you can use features not found in one to enhance the other. Similarly, you can run applications using Solaris threads, exclusively, with applications using pthreads, exclusively, on the same system.

Major API Differences

Solaris threads and pthreads are very similar in both API action and syntax. The major differences are listed in Table 9-1.

Table 9-1 Unique Solaris Threads and pthreads Features

Solaris Threads, Only	POSIX Threads, Only
`thr_` prefix for threads function names; `sema_` prefix for semaphore function names	`pthread_` prefix for pthreads function names; `sem_` prefix for semaphore function names
Readers/Writer locks	Attribute objects (these replace many Solaris arguments or flags with pointers to pthreads attribute objects)
Ability to create "daemon" threads	Cancellation semantics
Suspending and continuing a thread	Scheduling policies
Setting concurrency (requesting a new LWP): determining concurrency level	

Function Comparison Table

The following table compares Solaris threads functions with pthreads functions. Note that even when Solaris threads and pthreads functions appear to be essentially the same, the arguments to the functions can differ.

When a comparable interface is not available either in pthreads or Solaris threads, the character '-' appears in the column. Entries in the pthreads column that are followed by "POSIX 1003.4" or "POSIX.4" are part of the POSIX Realtime standard specification and are not part of pthreads.

Table 9-2 Solaris Threads and POSIX pthreads Comparison

Solaris Threads	pthreads
thr_create()	pthread_create()
thr_exit()	pthread_exit()
thr_join()	pthread_join()
thr_yield()	sched_yield() POSIX.4
thr_self()	pthread_self()
thr_kill()	pthread_kill()
thr_sigsetmask()	pthread_sigmask()
thr_setprio()	pthread_setschedparam()
thr_getprio()	pthread_getschedparam()
thr_setconcurrency()	-
thr_getconcurrency()	-
thr_suspend()	-
thr_continue()	-
thr_keycreate()	pthread_key_create()
-	pthread_key_delete()
thr_setspecific()	pthread_setspecific()
thr_getspecific()	pthread_getspecific()
-	pthread_once()
-	pthread_equal()
-	pthread_cancel()
-	pthread_testcancel()
-	pthread_cleanup_push()
-	pthread_cleanup_pop()
-	pthread_setcanceltype()
-	pthread_setcancelstate()
mutex_lock()	pthread_mutex_lock()
mutex_unlock()	pthread_mutex_unlock()
mutex_trylock()	pthread_mutex_trylock()
mutex_init()	pthread_mutex_init()
mutex_destroy()	pthread_mutex_destroy()
cond_wait()	pthread_cond_wait()
cond_timedwait()	pthread_cond_timedwait()
cond_signal()	pthread_cond_signal()
cond_broadcast()	pthread_cond_broadcast()

Table 9-2 Solaris Threads and POSIX pthreads Comparison

Solaris Threads	pthreads
cond_init()	pthread_cond_init()
cond_destroy()	pthread_cond_destroy()
rwlock_init()	–
rwlock_destroy()	–
rw_rdlock()	–
rw_wrlock()	–
rw_unlock()	–
rw_tryrdlock()	–
rw_trywrlock()	–
sema_init()	sem_init() POSIX 1003.4
sema_destroy()	sem_destroy() POSIX 1003.4
sema_wait()	sem_wait() POSIX 1003.4
sema_post()	sem_post() POSIX 1003.4
sema_trywait()	sem_trywait() POSIX 1003.4
fork1()	fork()
–	pthread_atfork()
fork() (multiple thread copy)	–
–	pthread_mutexattr_init()
–	pthread_mutexattr_destroy()
type argument in cond_init()	pthread_mutexattr_setpshared()
–	pthread_mutxattr_getpshared()
–	pthread_condattr_init()
–	pthread_condattr_destroy()
type argument in cond_init()	pthread_condattr_setpshared()
–	pthread_condattr_getpshared()
–	pthread_attr_init()
–	pthread_attr_destroy()
THR_BOUND flag in thr_create()	pthread_attr_setscope()
–	pthread_attr_getscope()
stack_size argument in thr_create()	pthread_attr_setstacksize()
–	pthread_attr_getstacksize()
stack_addr argument in thr_create()	pthread_attr_setstackaddr()
–	pthread_attr_getstackaddr()
THR_DETACH flag in thr_create()	pthread_attr_setdetachstate()

Table 9-2 Solaris Threads and POSIX pthreads Comparison

Solaris Threads	pthreads
–	`pthread_attr_getdetachstate()`
–	`pthread_attr_setschedparam()`
–	`pthread_attr_getschedparam()`
–	`pthread_attr_setinheritsched()`
–	`pthread_attr_getinheritsched()`
–	`pthread_attr_setsschedpolicy()`
–	`pthread_attr_getschedpolicy()`

To use the Solaris threads functions described in this chapter, you must link with the Solaris threads library (`-lthread`).

Where functionality is virtually the same for both Solaris threads and for pthreads, (even though the function names or arguments might differ), only a brief example consisting of the correct include file and the function prototype is presented. Where return values are not given for the Solaris threads functions, see the appropriate pages in *man Pages(3): Library Routines* for the function return values.

For more information on Solaris related functions, see the related pthreads documentation for the similarly named function.

Where Solaris threads functions offer capabilities that are not available in pthreads, a full description of the functions is provided.

9

Unique Solaris Threads Functions

Suspend Thread Execution

thr_suspend(3T)

`thr_suspend()` immediately suspends the execution of the thread specified by *target_thread*. On successful return from `thr_suspend()`, the suspended thread is no longer executing.

Once a thread is suspended, subsequent calls to `thr_suspend()` have no effect. Signals can not awaken the suspended thread; they remain pending until the thread resumes execution.

```
#include <thread.h>

int thr_suspend(thread_t tid);
```

In the following synopsis, `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() */

/* pthreads equivalent of Solaris tid from thread created */
/* with pthread_create() */
pthread_t ptid;

int ret;

ret = thr_suspend(tid);

/* using pthreads ID variable with a cast */
ret = thr_suspend((thread_t) ptid);
```

Return Values

Returns zero after completing successfully. Any other returned value indicates that an error occurred. When the following condition occurs, `thr_suspend()` fails and returns the corresponding value.

`ESRCH` - *tid* cannot be found in the current process.

Continue a Suspended Thread

thr_continue(3T)

`thr_continue()` resumes the execution of a suspended thread. Once a suspended thread is continued, subsequent calls to `thr_continue()` have no effect.

```
#include <thread.h>

int thr_continue(thread_t tid);
```

A suspended thread will not be awakened by a signal. The signal stays pending until the execution of the thread is resumed by `thr_continue()`.

`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()*/

/* pthreads equivalent of Solaris tid from thread created */
/* with pthread_create()*/
pthread_t ptid;

int ret;

ret = thr_continue(tid);

/* using pthreads ID variable with a cast */
ret = thr_continue((thread_t) ptid)
```

Return Values

`thr_continue()` returns zero after completing successfully. Any other returned value indicates that an error occurred. When the following condition occurs, `thr_continue()` fails and returns the corresponding value.

`ESRCH` - *tid* cannot be found in the current process.

Set Thread Concurrency Level

By default, Solaris threads attempts to adjust the system execution resources (LWPs) used to run unbound threads to match the real number of active threads. While the Solaris threads package cannot make perfect decisions, it at least ensures that the process continues to make progress.

When you have some idea of the number of unbound threads that should be simultaneously active (executing code or system calls), tell the library through `thr_setconcurrency()`. To get the number of threads being used, use `thr_getconcurrency()`.

thr_setconcurrency(3T)

`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.

```
#include <thread.h>

int new_level;
int ret;

ret = thr_setconcurrency(new_level);
```

Unbound threads in a process might or might not be required to be simultaneously active. To conserve system resources, the threads system ensures by default that enough threads are active for the process to make progress, and that the process will not deadlock through a lack of concurrency.

Because this might not produce the most effective level of concurrency, `thr_setconcurrency()` permits the application to give the threads system a hint, specified by *new_level*, for the desired level of concurrency.

The actual number of simultaneously active threads can be larger or smaller than *new_level*.

Note that an application with multiple compute-bound threads can fail to schedule all the runnable threads if `thr_setconcurrency()` has not been called to adjust the level of execution resources.

You can also affect the value for the desired concurrency level by setting the `THR_NEW_LWP` flag in `thr_create()`. This effectively increments the current level by one.

Return Values

Returns a zero when it completes successfully. Any other returned value indicates that an error occurred. When any of the following conditions are detected, `thr_setconcurrency()` fails and returns the corresponding value.

`EAGAIN` - The specified concurrency level would cause a system resource to be exceeded.

`EINVAL` - The value for *new_level* is negative.

Get Thread Concurrency

thr_getconcurrency(3T)

Use `thr_getconcurrency()` to get the current value of the concurrency level previously set by `thr_setconcurrency()`. Note that the actual number of simultaneously active threads can be larger or smaller than this number.

```
#include <thread.h>

int thr_getconcurrency(void)
```

Return Value

`thr_getconcurrency()` always returns the current value for the desired concurrency level.

9

Unique Solaris Synchronization Functions–Readers/Writer Locks

Readers/Writer locks allow simultaneous read access by many threads while restricting write access to only one thread at a time.

When any thread holds the lock for reading, other threads can also acquire the lock for reading but must wait to acquire the lock for writing. If one thread holds the lock for writing, or is waiting to acquire the lock for writing, other threads must wait to acquire the lock for either reading or writing.

Readers/Writer locks are slower than mutexes, but can improve performance when they protect data that are not frequently written but that are read by many concurrent threads.

Use readers/writer locks to synchronize threads in this process and other processes by allocating them in memory that is writable and shared among the cooperating processes (see `mmap`(2)) and by initializing them for this behavior.

By default, the acquisition order is not defined when multiple threads are waiting for a readers/writer lock. However, to avoid writer starvation, the Solaris threads package tends to favor writers over readers.

Readers/Writer locks must be initialized before use.

Initialize a Readers/Writer Lock

rwlock_init(3T)

```
#include <synch.h>  (or #include <thread.h>)

int rwlock_init(rwlock_t *rwlp, int type, void * arg);
```

Use `rwlock_init()` to initialize the readers/writer lock pointed to by *rwlp* and to set the lock state to unlocked. *type* can be one of the following (note that *arg* is currently ignored).

- `USYNC_PROCESS` The readers/writer lock can be used to synchronize threads in this process and other processes. *arg* is ignored.
- `USYNC_THREAD` The readers/writer lock can be used to synchronize threads in this process, only. *arg* is ignored.

Multiple threads must not initialize the same readers/writer lock simultaneously. Readers/Writer locks can also be initialized by allocation in zeroed memory, in which case a type of `USYNC_THREAD` is assumed. A readers/writer lock must not be reinitialized while other threads might be using it.

Initializing Readers/Writer Locks With Intraprocess Scope

```
#include <thread.h>

rwlock_t rwlp;
int ret;

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

Initializing Readers/Writer Locks With Interprocess Scope

```
#include <thread.h>

rwlock_t rwlp;
int ret;

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

Return Values

`rwlock_init()` returns zero after completing successfully. Any other returned value indicates that an error occurred. When any of the following conditions occur, the function fails and returns the corresponding value.

`EINVAL` - Invalid argument.

`EFAULT` - *rwlp* or *arg* points to an illegal address.

Acquire a Read Lock

rw_rdlock(3T)

```
#include <synch.h> (or #include <thread.h>)

int rw_rdlock(rwlock_t *rwlp);
```

Use rw_rdlock() to acquire a read lock on the readers/writer lock pointed to by *rwlp*. When the readers/writer lock is already locked for writing, the calling thread blocks until the write lock is released. Otherwise, the read lock is acquired.

Return Values

rw_rdlock() returns zero after completing successfully. Any other returned value indicates that an error occurred. When any of the following conditions occur, the function fails and returns the corresponding value.

EINVAL - Invalid argument.

EFAULT - *rwlp* points to an illegal address.

Try to Acquire a Read Lock

rw_tryrdlock(3T)

```
#include <synch.h>  (or #include <thread.h>)

int rw_tryrdlock(rwlock_t *rwlp);
```

Use rw_tryrdlock() to attempt to acquire a read lock on the readers/writer lock pointed to by *rwlp*. When the readers/writer lock is already locked for writing, it returns an error. Otherwise, the read lock is acquired.

Return Values

`rw_tryrdlock()` returns zero after completing successfully. Any other returned value indicates that an error occurred. When any of the following conditions occur, the function fails and returns the corresponding value.

`EINVAL` - Invalid argument.

`EFAULT` - *rwlp* points to an illegal address.

`EBUSY` - The readers/writer lock pointed to by *rwlp* was already locked.

Acquire a Write Lock

rw_wrlock(3T)

```
#include <synch.h>  (or #include <thread.h>)

int rw_wrlock(rwlock_t *rwlp);
```

Use `rw_wrlock()` to acquire a write lock on the readers/writer lock pointed to by *rwlp*. When the readers/writer lock is already locked for reading or writing, the calling thread blocks until all the read locks and write locks are released. Only one thread at a time can hold a write lock on a readers/writer lock.

Return Values

`rw_wrlock()` returns zero after completing successfully. Any other returned value indicates that an error occurred. When any of the following conditions occur, the function fails and returns the corresponding value.

`EINVAL` - Invalid argument.

`EFAULT` - *rwlp* points to an illegal address.

Try to Acquire a Write Lock

rw_trywrlock(3T)

```
#include <synch.h>  (or #include <thread.h>)

int rw_trywrlock(rwlock_t *rwlp);
```

Use `rw_trywrlock()` to attempt to acquire a write lock on the readers/writer lock pointed to by *rwlp*. When the readers/writer lock is already locked for reading or writing, it returns an error.

Return Values

`rw_trywrlock()` returns zero after completing successfully. Any other returned value indicates that an error occurred. When any of the following conditions occur, the function fails and returns the corresponding value.

`EINVAL` - Invalid argument.

`EFAULT` - *rwlp* points to an illegal address.

`EBUSY` - The readers/writer lock pointed to by *rwlp* was already locked.

Unlock a Readers/Writer Lock

rw_unlock(3T)

```
#include <synch.h>  (or #include <thread.h>)

int rw_unlock(rwlock_t *rwlp);
```

Use `rw_unlock()` to unlock a readers/writer lock pointed to by *rwlp*. The readers/writer lock must be locked and the calling thread must hold the lock either for reading or writing. When any other threads are waiting for the readers/writer lock to become available, one of them is unblocked.

Return Values

rw_unlock() returns zero after completing successfully. Any other returned value indicates that an error occurred. When any of the following conditions occur, the function fails and returns the corresponding value.

EINVAL - Invalid argument.

EFAULT - *rwlp* points to an illegal address.

Destroy Readers/Writer Lock State

rwlock_destroy(3T)

```
#include <synch.h>  (or #include <thread.h>)

int rwlock_destroy(rwlock_t *rwlp);
```

Use rwlock_destroy() to destroy any state associated with the readers/writer lock pointed to by *rlwp*. The space for storing the readers/writer lock is not freed.

Return Values

rwlock_destroy() returns zero after completing successfully. Any other returned value indicates that an error occurred. When any of the following conditions occur, the function fails and returns the corresponding value.

EINVAL - Invalid argument.

EFAULT - *rwlp* points to an illegal address.

Readers/Writer Lock Example

Code Example 9-1 uses a bank account to demonstrate readers/writer locks. While the program could allow multiple threads to have concurrent read-only access to the account balance, only a single writer is allowed. Note that the get_balance() function needs the lock to ensure that the addition of the checking and saving balances occurs atomically.

Code Example 9-1 Read/Write Bank Account

```
rwlock_t account_lock;
float checking_balance = 100.0;
float saving_balance = 100.0;
...
rwlock_init(&account_lock, 0, NULL);
...
float
get_balance() {
    float bal;

    rw_rdlock(&account_lock);
    bal = checking_balance + saving_balance;
    rw_unlock(&account_lock);
    return(bal);
}

void
transfer_checking_to_savings(float amount) {
    rw_wrlock(&account_lock);
    checking_balance = checking_balance - amount;
    saving_balance = saving_balance + amount;
    rw_unlock(&account_lock);
}
```

9

Similar Solaris Threads Functions

Create a Thread

The `thr_create`(3T) routine is one of the most elaborate of all the Solaris threads library routines.

thr_create(3T)

Use `thr_create`() to add a new thread of control to the current process.

Note that the new thread does not inherit pending signals, but it does inherit priority and signal masks.

```
#include <thread.h>

int thr_create(void *stack_base, size_t stack_size,
    void *(*start_routine) (void *), void *arg, long flags,
    thread_t *new_thread);

size_t thr_min_stack(void);
```

stack_base—Contains the address for the stack that the new thread uses. If *stack_base* is `NULL` then `thr_create()` allocates a stack for the new thread with at least *stack_size* bytes.

stack_size—Contains the size, in number of bytes, for the stack that the new thread uses. If *stack_size* is zero, a default size is used. In most cases, a zero value works best. If *stack_size* is not zero, it must be greater than the value returned by `thr_min_stack()`.

There is no general need to allocate stack space for threads. The threads library allocates one megabyte of virtual memory for each thread's stack with no swap space reserved. (The library uses the `MAP_NORESERVE` option of `mmap()` to make the allocations.)

start_routine—Contains the function with which the new thread begins execution. When *start_routine* returns, the thread exits with the exit status set to the value returned by *start_routine* (see "`thr_exit(3T)`").

arg—Can be anything that is described by `void`, which is typically any 4-byte value. Anything larger must be passed indirectly by having the argument point to it.

Note that you can supply only one argument. To get your procedure to take multiple arguments, encode them as one (such as by putting them in a structure).

flags–Specifies attributes for the created thread. In most cases a zero value works best.

The value in *flags* is constructed from the bitwise inclusive OR of the following:

- `THR_SUSPENDED`—Suspends the new thread and does not execute *start_routine* until the thread is started by `thr_continue()`. Use this to operate on the thread (such as changing its priority) before you run it. The termination of a detached thread is ignored.
- `THR_DETACHED`—Detaches the new thread so that its thread ID and other resources can be reused as soon as the thread terminates. Set this when you do not want to wait for the thread to terminate.

Note – When there is no explicit synchronization to prevent it, an unsuspended, detached thread can die and have its thread ID reassigned to another new thread before its creator returns from `thr_create()`.

- THR_BOUND—Permanently binds the new thread to an LWP (the new thread is a *bound thread*).
- THR_NEW_LWP—Increases the concurrency level for unbound threads by one. The effect is similar to incrementing concurrency by one with thr_setconcurrency(3T), although THR_NEW_LWP does not affect the level set through the thr_setconcurrency() function. Typically, THR_NEW_LWP adds a new LWP to the pool of LWPs running unbound threads.
- When you specify both THR_BOUND and THR_NEW_LWP, two LWPs are typically created—one for the bound thread and another for the pool of LWPs running unbound threads.
- THR_DAEMON—Marks the new thread as a daemon. The process exits when all nondaemon threads exit. Daemon threads do not affect the process exit status and are ignored when counting the number of thread exits.

  A process can exit either by calling exit() or by having every thread in the process that was not created with the THR_DAEMON flag call thr_exit(3T). An application, or a library it calls, can create one or more threads that should be ignored (not counted) in the decision of whether to exit. The THR_DAEMON flag identifies threads that are not counted in the process exit criterion.

new_thread—Points to a location (when *new_thread* is not NULL) where the ID of the new thread is stored when thr_create() is successful. The caller is responsible for supplying the storage this argument points to. The ID is valid only within the calling process.

If you are not interested in this identifier, supply a zero value to *new_thread*.

Return Values

Returns a zero and exits when it completes successfully. Any other returned value indicates that an error occurred. When any of the following conditions are detected, thr_create() fails and returns the corresponding value.

EAGAIN - A system limit is exceeded, such as when too many LWPs have been created.

ENOMEM - Not enough memory was available to create the new thread.

EINVAL - *stack_base* is not NULL and *stack_size* is less than the value returned by `thr_min_stack()`.

Stack Behavior

Stack behavior in Solaris threads is generally the same as that in pthreads. For more information about stack setup and operation, see *"About Stacks" on page 61*.

You can get the absolute minimum on stack size by calling `thr_min_stack()`, which returns the amount of stack space required for a thread that executes a null procedure. Useful threads need more than this, so be very careful when reducing the stack size.

You can specify a custom stack in two ways. The first is to supply a `NULL` for the stack location, thereby asking the runtime library to allocate the space for the stack, but to supply the desired size in the stacksize parameter to `thr_create()`.

The other approach is to take overall aspects of stack management and supply a pointer to the stack to `thr_create()`. This means that you are responsible not only for stack allocation but also for stack deallocation—when the thread terminates, you must arrange for the disposal of its stack.

When you allocate your own stack, be sure to append a red zone to its end by calling `mprotect`(2).

Get the Minimal Stack Size

thr_min_stack(3T)

Use `thr_min_stack`(3T) to get the minimum stack size for a thread.

```
#include <thread.h>

size_t thr_min_stack(void);
```

`thr_min_stack()` returns the amount of space needed to execute a null thread (a null thread is a thread that is created to execute a null procedure).

A thread that does more than execute a null procedure should allocate a stack size greater than the size of thr_min_stack().

When a thread is created with a user-supplied stack, the user must reserve enough space to run the thread. In a dynamically linked execution environment, it is difficult to know what the thread minimal stack requirements are.

Most users should not create threads with user-supplied stacks. User-supplied stacks exist only to support applications that require complete control over their execution environments.

Instead, users should let the threads library manage stack allocation. The threads library provides default stacks that should meet the requirements of any created thread.

Get the Thread Identifier

thr_self(3T)

Use thr_self(3T) to get the ID of the calling thread.

```
#include <thread.h>

thread_t thr_self(void);
```

Yield Thread Execution

thr_yield(3T)

thr_yield() causes the current thread to yield its execution in favor of another thread with the same or greater priority; otherwise it has no effect. There is no guarantee that a thread calling thr_yield() will do so.

```
#include <thread.h>

void thr_yield(void);
```

Send a Signal to a Thread

thr_kill(3T)

thr_kill() sends a signal to a thread.

```
#include <thread.h>
#include <signal.h>

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

Access the Signal Mask of the Calling Thread

thr_sigsetmask(3T)

Use thr_sigsetmask() to change or examine the signal mask of the calling thread.

```
#include <thread.h>
#include <signal.h>

int thr_sigsetmask(int how, const sigset_t *set, sigset_t *oset);
```

Terminate a Thread

thr_exit(3T)

Use thr_exit() to terminate a thread.

```
#include <thread.h>

void thr_exit(void *status);
```

Wait for Thread Termination

thr_join(3T)

Use the `thr_join()` function to wait for a thread to terminate.

```
#include <thread.h>

int thr_join(thread_t tid, thread_t *departedid, void **status);
```

Join specific

```
#include <thread.h>

thread_t tid;
thread_t departedid;
int ret;
int status;

/* waiting to join thread "tid" with status */
ret = thr_join(tid, &departedid, (void**)&status);

/* waiting to join thread "tid" without status */
ret = thr_join(tid, &departedid, NULL);

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

When the *tid* is `(thread_t)0`, then `thread_join()` waits for any undetached thread in the process to terminate. In other words, when no thread identifier is specified, any undetached thread that exits causes `thread_join()` to return.

Join any

```
#include <thread.h>

thread_t tid;
thread_t departedid;
int ret;
int status;

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

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.

Create a Thread-Specific Data Key

Except for the function names and arguments, thread specific data is the same for Solaris as it is for POSIX. The synopses for the Solaris functions are given in this section. The functions are explained in “Create a Thread-Specific Data Key” on page 207.

thr_keycreate(3T)

`thr_keycreate()` allocates a key that is used to identify thread-specific data in a process.

```
#include <thread.h>

int thr_keycreate(thread_key_t *keyp,
    void (*destructor) (void *value));
```

Set the Thread-Specific Data Key

thr_setspecific(3T)

`thr_setspecific()` binds *value* to the thread-specific data key, *key*, for the calling thread.

```
#include <thread.h>

int thr_setspecific(thread_key_t key, void *value);
```

Get the Thread-Specific Data Key

thr_getspecific(3T)

`thr_getspecific()` stores the current value bound to *key* for the calling thread into the location pointed to by *valuep*.

```
#include <thread.h>

int thr_getspecific(thread_key_t key, void **valuep);
```

Set the Thread Priority

In Solaris threads, if a thread is to be created with a priority other than that of its parent's, it is created in SUSPEND mode. While suspended, the threads priority is modified using the `thr_setprio(3T)` function call; then it is continued.

An unbound thread is usually scheduled only with respect to other threads in the process using simple priority levels with no adjustments and no kernel involvement. Its system priority is usually uniform and is inherited from the creating process.

thr_setprio(3T)

The function `thr_setprio()` changes the priority of the thread, specified by *tid*, within the current process to the priority specified by *newprio*.

```
#include <thread.h>

int thr_setprio(thread_t tid, int newprio)
```

By default, threads are scheduled based on fixed priorities that range from zero, the least significant, to the largest integer. The *tid* will preempt lower priority threads, and will yield to higher priority threads.

```
thread_t tid;
int ret;
int newprio = 20;

/* suspended thread creation */
ret = thr_create(NULL, NULL, func, arg, THR_SUSPEND, &tid);

/* set the new priority of suspended child thread */
ret = thr_setprio(tid, newprio);

/* suspended child thread starts executing with new priority */
ret = thr_continue(tid);
```

Get the Thread Priority

thr_getprio(3T)

Use `thr_getprio()` to get the current priority for the thread. Each thread inherits a priority from its creator. `thr_getprio()` stores the current priority, *tid*, in the location pointed to by *newprio*.

```
#include <thread.h>

int thr_getprio(thread_t tid, int *newprio)
```

9

Similar Synchronization Functions–Mutual Exclusion Locks

Initialize a Mutex

mutex_init(3T)

```
#include <synch.h>  (or #include <thread.h>)

int mutex_init(mutex_t *mp, int type, void *arg));
```

Use `mutex_init()` to initialize the mutex pointed to by *mp*. The *type* can be one of the following (note that *arg* is currently ignored).

- `USYNC_PROCESS` The mutex can be used to synchronize threads in this and other processes.
- `USYNC_THREAD` The mutex can be used to synchronize threads in this process, only.

Mutexes can also be initialized by allocation in zeroed memory, in which case a *type* of `USYNC_THREAD` is assumed.

Multiple threads must not initialize the same mutex simultaneously. A mutex lock must not be reinitialized while other threads might be using it.

Mutexes With Intraprocess Scope

```
#include <thread.h>

mutex_t mp;
int ret;

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

Mutexes With Interprocess Scope

```
#include <thread.h>

mutex_t mp;
int ret;

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

Destroy a Mutex

mutex_destroy(3T)

```
#include <thread.h>

int mutex_destroy (mutex_t *mp);
```

Use `mutex_destroy()` to destroy any state associated with the mutex pointed to by *mp*. Note that the space for storing the mutex is not freed.

Acquire a Mutex

mutex_lock(3T)

```
#include <thread.h>

int mutex_lock(mutex_t *mp);
```

Use mutex_lock() to lock the mutex pointed to by *mp*. When the mutex is already locked, the calling thread blocks until the mutex becomes available (blocked threads wait on a prioritized queue).

Release a Mutex

mutex_unlock(3T)

```
#include <thread.h>

int mutex_unlock(mutex_t *mp);
```

Use mutex_unlock() to unlock the mutex pointed to by *mp*. The mutex must be locked and the calling thread must be the one that last locked the mutex (the owner).

Try to Acquire a Mutex

mutex_trylock(3T)

```
#include <thread.h>

int mutex_trylock(mutex_t *mp);
```

Use mutex_trylock() to attempt to lock the mutex pointed to by *mp*. This function is a nonblocking version of mutex_lock().

Similar Synchronization Functions–Condition Variables

Initialize a Condition Variable

cond_init(3T)

```
#include <thread.h>

int cond_init(cond_t *cv, int type, int arg);
```

Use `cond_init()` to initialize the condition variable pointed to by *cv*. The *type* can be one of the following (note that *arg* is currently ignored).

- `USYNC_PROCESS` The condition variable can be used to synchronize threads in this and other processes. *arg* is ignored.
- `USYNC_THREAD` The condition variable can be used to synchronize threads in this process, only. *arg* is ignored.

Condition variables can also be initialized by allocation in zeroed memory, in which case a type of `USYNC_THREAD` is assumed.

Multiple threads must not initialize the same condition variable simultaneously. A condition variable must not be reinitialized while other threads might be using it.

Condition Variables With Intraprocess Scope

```
#include <thread.h>

cond_t cv;
int ret;

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

Condition Variables With Interprocess Scope

```
#include <thread.h>

cond_t cv;
int ret;

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

Destroy a Condition Variable

cond_destroy(3T)

```
#include <thread.h>

int cond_destroy(cond_t *cv);
```

Use `cond_destroy()` to destroy state associated with the condition variable pointed to by *cv*. The space for storing the condition variable is not freed.

Wait for a Condition

cond_wait(3T)

```
#include <thread.h>

int cond_wait(cond_t *cv, mutex_t *mp);
```

Use `cond_wait()` to atomically release the mutex pointed to by *mp* and to cause the calling thread to block on the condition variable pointed to by *cv*. The blocked thread can be awakened by `cond_signal()`, `cond_broadcast()`, or when interrupted by delivery of a signal or a `fork()`.

Wait For an Absolute Time

cond_timedwait(3T)

```
#include <thread.h>

int cond_timedwait(cond_t *cv, mutex_t *mp, timestruct_t abstime)
```

Use `cond_timedwait()` as you would use `cond_wait()`, except that `cond_timedwait()` does not block past the time of day specified by *abstime*.

`cond_timedwait()` always returns with the mutex locked and owned by the calling thread even when returning an error.

The `cond_timedwait()` function blocks until the condition is signaled or until the time of day specified by the last argument has passed. The time-out is specified as a time of day so the condition can be retested efficiently without recomputing the time-out value.

Signal One Condition Variable

cond_signal(3T)

```
#include <thread.h>

int cond_signal(cond_t *cv);
```

Use `cond_signal()` to unblock one thread that is blocked on the condition variable pointed to by *cv*. Call this function under protection of the same mutex used with the condition variable being signaled. Otherwise, the condition could be signaled between its test and `cond_wait()`, causing an infinite wait.

Signal All Condition Variables

cond_broadcast(3T)

```
#include <thread.h>

int cond_broadcast(cond_t *cv);
```

Use `cond_broadcast()` to unblock all threads that are blocked on the condition variable pointed to by *cv*. When no threads are blocked on the condition variable then `cond_broadcast()` has no effect.

Similar Synchronization Functions–Semaphores

Semaphore operations are the same in both Solaris and POSIX. The function name changed from `sema_` in Solaris to `sem_` in pthreads.

Initialize a Semaphore

sema_init(3T)

```
#include <thread.h>

int sema_init(sema_t *sp, unsigned int count, int type,
    void *arg);
```

Use `sema_init()` to initialize the semaphore variable pointed to by *sp* by *count* amount. *type* can be one of the following (note that *arg* is currently ignored).

`USYNC_PROCESS` The semaphore can be used to synchronize threads in this process and other processes. Only one process should initialize the semaphore. *arg* is ignored.

`USYNC_THREAD` The semaphore can be used to synchronize threads in this process, only. *arg* is ignored.

Multiple threads must not initialize the same semaphore simultaneously. A semaphore must not be reinitialized while other threads may be using it.

Semaphores With Intraprocess Scope

```
#include <thread.h>

sema_t sp;
int ret;
int count;
count = 4;

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

Semaphores With Interprocess Scope

```
#include <thread.h>

sema_t sp;
int ret;
int count;
count = 4;

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

Increment a Semaphore

sema_post(3T)

```
#include <thread.h>

int sema_post(sema_t *sp);
```

Use sema_post() to atomically increment the semaphore pointed to by *sp*. When any threads are blocked on the semaphore, one is unblocked.

Block on a Semaphore Count

sema_wait(3T)

```
#include <thread.h>

int sema_wait(sema_t *sp);
```

Use sema_wait() to block the calling thread until the count in the semaphore pointed to by *sp* becomes greater than zero, then atomically decrement it.

Decrement a Semaphore Count

sema_trywait(3T)

```
#include <thread.h>

int sema_trywait(sema_t *sp);
```

Use `sema_trywait()` to atomically decrement the count in the semaphore pointed to by *sp* when the count is greater than zero. This function is a nonblocking version of `sema_wait()`.

Destroy the Semaphore State

sema_destroy(3T)

```
#include <thread.h>

int sema_destroy(sema_t *sp);
```

Use `sema_destroy()` to destroy any state associated with the semaphore pointed to by *sp*. The space for storing the semaphore is not freed.

9

Synchronization Across Process Boundaries

Each of the synchronization primitives can be set up to be used across process boundaries. This is done quite simply by ensuring that the synchronization variable is located in a shared memory segment and by calling the appropriate `init` routine with *type* set to `USYNC_PROCESS`.

If this has been done, then the operations on the synchronization variables work just as they do when *type* is `USYNC_THREAD`.

```
mutex_init(&m, USYNC_PROCESS, 0);

rwlock_init(&rw, USYNC_PROCESS, 0);

cond_init(&cv, USYNC_PROCESS, 0);

sema_init(&s, count, USYNC_PROCESS, 0);
```

Using LWPs Between Processes

Using locks and condition variables between processes does not require using the threads library. The recommended approach is to use the threads library interfaces, but when this is not desirable, then the `_lwp_mutex_*` and `_lwp_cond_*` interfaces can be used as follows:

1. Allocate the locks and condition variables as usual in shared memory (either with `shmop(2)` or `mmap(2)`).

2. Then initialize the newly allocated objects appropriately with the `USYNC_PROCESS` type. Because no interface is available to perform the initialization (`_lwp_mutex_init(2)` and `_lwp_cond_init(2)` do not exist), the objects can be initialized using statically allocated and initialized dummy objects.

For example, to initialize lockp:

```
lwp_mutex_t *lwp_lockp;
lwp_mutex_t dummy_shared_mutex = SHAREDMUTEX;
    /* SHAREDMUTEX is defined in /usr/include/synch.h */
...
...
lwp_lockp = alloc_shared_lock();
*lwp_lockp = dummy_shared_mutex;
```

Similarly, for condition variables:

```
lwp_cond_t *lwp_condp;
lwp_cond_t dummy_shared_cv = SHAREDCV;
    /* SHAREDCV is defined in /usr/include/synch.h */
...
...
lwp_condp = alloc_shared_cv();
*lwp_condp = dummy_shared_cv;
```

Producer/Consumer Problem Example

Code Example 9-2 shows the producer/consumer problem with the producer and consumer in separate processes. The main routine maps zero-filled memory (that it shares with its child process) into its address space. Note that `mutex_init()` and `cond_init()` must be called because the type of the synchronization variables is `USYNC_PROCESS`.

A child process is created that runs the consumer. The parent runs the producer.

This example also shows the drivers for the producer and consumer. The `producer_driver()` simply reads characters from `stdin` and calls `producer()`. The `consumer_driver()` gets characters by calling `consumer()` and writes them to `stdout`.

The data structure for Code Example 9-2 is the same as that used for the solution with condition variables (see page 84).

Code Example 9-2 The Producer/Consumer Problem, Using `USYNC_PROCESS`

```
main() {
    int zfd;
    buffer_t *buffer;

    zfd = open("/dev/zero", O_RDWR);
    buffer = (buffer_t *)mmap(NULL, sizeof(buffer_t),
        PROT_READ|PROT_WRITE, MAP_SHARED, zfd, 0);
    buffer->occupied = buffer->nextin = buffer->nextout = 0;

    mutex_init(&buffer->lock, USYNC_PROCESS, 0);
    cond_init(&buffer->less, USYNC_PROCESS, 0);
    cond_init(&buffer->more, USYNC_PROCESS, 0);
    if (fork() == 0)
        consumer_driver(buffer);
    else
        producer_driver(buffer);
}

void producer_driver(buffer_t *b) {
    int item;

    while (1) {
        item = getchar();
        if (item == EOF) {
            producer(b, '\0');
            break;
        } else
            producer(b, (char)item);
    }
}

void consumer_driver(buffer_t *b) {
    char item;

    while (1) {
        if ((item = consumer(b)) == '\0')
            break;
        putchar(item);
    }
}
```

A child process is created to run the consumer; the parent runs the producer.

Special Issues for `fork()` and Solaris Threads

Solaris threads and POSIX threads define the behavior of `fork()` differently. See "Process Creation–exec(2)and exit(2) Issues" on page 124 for a thorough discussion of `fork()` issues.

Solaris `libthread` supports both `fork`() and `fork1`(). The `fork`() call has "fork-all" semantics—it duplicates everything in the process, including threads and LWPs, creating a true clone of the parent. The `fork1`() call creates a clone that has only one thread; the process state and address space are duplicated, but only the calling thread is cloned.

POSIX `libpthread` supports only `fork`(), which has the same semantics as `fork1`() in Solaris threads.

Whether `fork`() has "fork-all" semantics or "fork-one" semantics is dependent upon which library is used. Linking with `-lthread` assigns "fork-all" semantics to `fork`(), while linking with `-lpthread` assigns "fork-one" semantics to `fork`().

See "Linking With libthread or libpthread" on page 157 for more details.