Programming With Synchronization Objects 4

This chapter describes the synchronization types available with threads and discusses synchronization concerns.

Synchronization objects are variables in memory that you access just like data. Threads in different processes can communicate with each other through synchronization objects placed in threads-controlled shared memory, even though the threads in different processes are generally invisible to each other.

Synchronization objects can also be placed in files and can have lifetimes beyond that of the creating process.

The available types of synchronization objects are:

- Mutex Locks
- Condition Variables
- Semaphores

Here are situations that can profitably use synchronization:

- When synchronization is the only way to ensure consistency of shared data.
- When threads in two or more processes can use a single synchronization object jointly. Note that the synchronization object should be initialized by only one of the cooperating processes, because reinitializing a synchronization object sets it to the *unlocked* state.
- When synchronization can ensure the safety of mutable data.
- When a process can map a file and have a thread in this process get a record's lock. Once the lock is acquired, any other thread in any process mapping the file that tries to acquire the lock is blocked until the lock is released.
- Even when accessing a single primitive variable, such as an integer. On machines where the integer is not aligned to the bus data width or is larger than the data width, a single memory load can use more than one memory cycle. While this cannot happen on the SPARC® architecture, portable programs cannot rely on this.

Note – On 32-bit architectures a `long long` is not atomic[1] and is read and written as two 32-bit quantities. The types `int`, `char`, `float`, and pointers are atomic on SPARC and x86 machines.

Mutual Exclusion Lock Attributes

Use mutual exclusion locks (mutexes) to serialize thread execution. Mutual exclusion locks synchronize threads, usually by ensuring that only one thread at a time executes a critical section of code. Mutex locks can also preserve single-threaded code.

1. An *atomic* operation cannot be divided into smaller operations.

To change the default mutex attributes, you can declare and initialize an attribute object. Often, the mutex attributes are set in one place at the beginning of the application so they can be located quickly and modified easily. The following table lists the functions discussed in this section that manipulate mutex attributes.

Table 4-1 Mutex Attributes Routines

Initialize a Mutex Attribute Object	*pthread_mutexattr_init(3T)*	*page 72*
Destroy a Mutex Attribute Object	*pthread_mutexattr_destroy(3T)*	*page 73*
Set the Scope of a Mutex	*pthread_mutexattr_setpshared(3T)*	*page 74*
Get the Scope of a Mutex	*pthread_mutexattr_getpshared(3T)*	*page 75*

The differences in defining the scope of a mutex from the original Solaris threads are shown in Table 4-2.

Table 4-2 Mutex Scope Comparison

Solaris	POSIX	Definition
USYNC_PROCESS	PTHREAD_PROCESS_SHARED	Use to synchronize threads in this and other processes
USYNC_THREAD	PTHREAD_PROCESS_PRIVATE	Use to synchronize threads in this process only

Initialize a Mutex Attribute Object

pthread_mutexattr_init(3T)

Use `pthread_mutexattr_init()` to initialize attributes associated with this object to their default values. Storage for each attribute object is allocated by the threads system during execution.

The default value of the *pshared* attribute when this function is called is `PTHREAD_PROCESS_PRIVATE`, which means that the initialized mutex can be used within a process.

```
Prototype:
int pthread_mutexattr_init(pthread_mutexattr_t *mattr);
```

```
#include <pthread.h>

pthread_mutexattr_t mattr;
int ret;

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

mattr is an opaque type that contains a system-allocated attribute object. The possible values of *mattr*'s scope are `PTHREAD_PROCESS_PRIVATE` (the default) and `PTHREAD_PROCESS_SHARED`.

Before a mutex attribute object can be reused, it must first be destroyed by `pthread_mutexattr_destroy(3T)`. The `pthread_mutexattr_init()` call returns a pointer to an opaque object. If the object is not destroyed, a memory leak will result.

Return Values

Returns zero after completing successfully. Any other returned value indicates that an error occurred. If either of the following conditions occurs, the function fails and returns the corresponding value.

`ENOMEM` – There is not enough memory to initialize the thread attributes object.

`EINVAL` – The value specified by *mattr* is invalid.

Destroy a Mutex Attribute Object

pthread_mutexattr_destroy(3T)

`pthread_mutexattr_destroy()` deallocates the storage space used to maintain the attribute object created by `pthread_mutexattr_init()`.

Prototype:
```
int pthread_mutexattr_destroy(pthread_mutexattr_t *mattr)
```

```
#include <pthread.h>

pthread_mutexattr_t mattr;
int ret;

/* destroy an attribute */
ret = pthread_mutexattr_destroy(&mattr);
```

Return Values

Returns zero after completing successfully. Any other returned value indicates that an error occurred. If the following condition occurs, the function fails and returns the corresponding value.

`EINVAL` – The value specified by *mattr* is invalid.

Set the Scope of a Mutex

pthread_mutexattr_setpshared(3T)

The scope of a mutex variable can be either process private (intraprocess) or system wide (interprocess). If the mutex is created with the *pshared* attribute set to the PTHREAD_PROCESS_SHARED 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() in the original Solaris threads.

Prototype:
```
int pthread_mutexattr_setpshared(pthread_mutexattr_t *mattr,
    int pshared);
```

```
#include <pthread.h>

pthread_mutexattr_t mattr;
int pshared;
int ret;

ret = pthread_mutexattr_init(&mattr);
/*
 * resetting to its default value
 */
ret = pthread_mutexattr_setpshared(&mattr,
    PTHREAD_PROCESS_PRIVATE);
```

If the mutex *pshared* attribute is set to PTHREAD_PROCESS_PRIVATE, only those threads created by the same process can operate on the mutex.

Return Values

Returns zero after completing successfully. Any other returned value indicates that an error occurred. If the following condition occurs, the function fails and returns the corresponding value.

EINVAL – The value specified by *mattr* is invalid.

Get the Scope of a Mutex

pthread_mutexattr_getpshared(3T)

```
Prototype:
int pthread_mutexattr_getpshared(pthread_mutexattr_t *mattr,
    int pshared);
```

```
#include <pthread.h>

pthread_mutexattr_t mattr;
int pshared, ret;

/* get pshared of mutex */
ret = pthread_mutexattr_getpshared(&mattr, &pshared);
```

Get the current value of *pshared* for the attribute object *mattr*. It is either `PTHREAD_PROCESS_SHARED` or `PTHREAD_PROCESS_PRIVATE`.

Return Values

Returns zero after completing successfully. Any other returned value indicates that an error occurred. If the following condition occurs, the function fails and returns the corresponding value.

`EINVAL` – The value specified by *mattr* is invalid.

Using Mutual Exclusion Locks

After the attributes for a mutex are configured, you initialize the mutex itself. The following functions are used to initialize or destroy, lock or unlock, or try to lock a mutex. Table 4-3 lists the functions discussed in this chapter that manipulate mutex locks.

Table 4-3 Routines for Mutual Exclusion Locks

Initialize a Mutex	*pthread_mutex_init(3T)*	*page 76*
Lock a Mutex	*pthread_mutex_lock(3T)*	*page 78*
Unlock a Mutex	*pthread_mutex_unlock(3T)*	*page 79*
Lock With a Nonblocking Mutex	*pthread_mutex_trylock(3T)*	*page 80*
Destroy a Mutex	*pthread_mutex_destroy(3T)*	*page 81*

The default scheduling policy, SCHED_OTHER, does not specify the order in which threads can acquire a lock. When multiple threads are waiting for a mutex, the order of acquisition is undefined. When there is contention, the default behavior is to unblock threads in priority order.

Initialize a Mutex

pthread_mutex_init(3T)

Use pthread_mutex_init() to initialize the mutex pointed at by *mp* to its default value (*mattr* is NULL), or to specify mutex attributes that have already been set with pthread_mutexattr_init().

```
Prototype:
int pthread_mutex_init(pthread_mutex_t *mp,
    const pthread_mutexattr_t *mattr);
```

```
#include <pthread.h>

pthread_mutex_t mp = PTHREAD_MUTEX_INITIALIZER;
pthread_mutexattr_t mattr;
int ret;

/* initialize a mutex to its default value */
ret = pthread_mutex_init(&mp, NULL);

/* initialize a mutex */
ret = pthread_mutex_init(&mp, &mattr);
```

When the mutex is initialized, it is in an unlocked state.

The effect of *mattr* being `NULL` is the same as passing the address of a default mutex attribute object, but without the memory overhead.

Statically defined mutexes can be initialized directly to have default attributes with the macro `PTHREAD_MUTEX_INITIALIZER`.

If a mutex is dynamically allocated and was initialized with an attribute object, its attribute object should be freed with `pthread_mutexattr_destroy()` before the mutex itself is freed.

A mutex lock must not be reinitialized or destroyed while other threads might be using it. Program failure will result if either action is not done correctly. If a mutex is reinitialized or destroyed, the application must be sure the mutex is not currently in use.

Return Values

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.

`EBUSY` – The mutex cannot be reinitialized or modified because it still exists.

`EINVAL` – The attribute value is invalid. The mutex has not been modified.

`EAGAIN` – There are not enough resources to initialize another mutex.

`ENOMEM` – There is not enough memory to initialize another mutex.

Lock a Mutex

pthread_mutex_lock(3T)

```
Prototype:
int pthread_mutex_lock(pthread_mutex_t *mp);
```

```
#include <pthread.h>

pthread_mutex_t mp;
int ret;

ret = pthread_ mutex_lock(&mp); /* acquire the mutex */
```

Use `pthread_mutex_lock()` to lock the mutex pointed to by *mp*. When the mutex is already locked, the calling thread blocks and the mutex waits on a prioritized queue. When `pthread_mutex_lock()` returns, the mutex is locked and the calling thread is the owner.

Return Values

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` – The value specified by *mp* does not refer to an initialized mutex object.

`EDEADLK` – The current thread already owns the mutex.

Unlock a Mutex

pthread_mutex_unlock(3T)

Use pthread_mutex_unlock() to unlock the mutex pointed to by *mp*.

```
Prototype:
int pthread_mutex_unlock(pthread_mutex_t *mp);
```

```
#include <pthread.h>

pthread_mutex_t mp;
int ret;

ret = pthread_ mutex_unlock(&mp); /* release the mutex */
```

The mutex must be locked and the calling thread must be the one that last locked the mutex (the owner). When any other threads are waiting for the mutex to become available, the thread at the head of the queue is unblocked.

Return Values

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 – The value specified by *mp* does not refer to an initialized mutex object.

EPERM – The current thread does not own the mutex.

Lock With a Nonblocking Mutex

pthread_mutex_trylock(3T)

Use `pthread_mutex_trylock()` to attempt to lock the mutex pointed to by *mp*.

```
Prototype:
int pthread_mutex_trylock(pthread_mutex_t *mp);
```

```
#include <pthread.h>

pthread_mutex_t mp;
int ret;

ret = pthread_ mutex_trylock(&mp); /* try to lock the mutex */
```

This function is a nonblocking version of `pthread_mutex_lock()`. When the mutex is already locked, this call returns with an error. Otherwise, the mutex is locked and the calling thread is the owner.

Return Values

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.

`EBUSY` – The mutex pointed to by *mp* was already locked.

`EINVAL` – The value specified by *mp* does not refer to an initialized mutex object.

Destroy a Mutex

pthread_mutex_destroy(3T)

Use `pthread_mutex_destroy()` to destroy any state associated with the mutex pointed to by *mp.*

```
Prototype:
int pthread_mutex_destroy(pthread_mutex_t *mp);
```

```
#include <pthread.h>

mutex_t mp;
int ret;

ret = pthread_mutex_destroy(&mp); /* mutex is destroyed */
```

Note that the space for storing the mutex is not freed.

Return Values

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.

`EBUSY` – The mutex you are trying to destroy is locked or in use.

`EINVAL` – The value specified by *mp* does not refer to an initialized mutex object.

Mutex Lock Code Examples

Here are some code fragments showing mutex locking.

Code Example 4-1 Mutex Lock Example

```
#include <pthread.h>

pthread_mutex_t count_mutex;
long long count;

void
increment_count()
{
    pthread_mutex_lock(&count_mutex);
    count = count + 1;
    pthread_mutex_unlock(&count_mutex);
}

long long
get_count()
{
    long long c;

    pthread_mutex_lock(&count_mutex);
    c = count;
    pthread_mutex_unlock(&count_mutex);
    return (c);
}
```

The two functions in Code Example 4-1 use the mutex lock for different purposes. The `increment_count` function uses the mutex lock simply to assure an atomic update of the shared variable. The `get_count` function uses the mutex lock to guarantee that the 64-bit quantity `count` is read atomically. On a 32-bit architecture, a `long long` is really two 32-bit quantities.

Note that if `count` were an `int`, `get_count` would not need a mutex lock to read the value of `count`, because integer operations are atomic.

Using Locking Hierarchies

You will occasionally want to access two resources at once. Perhaps you are using one of the resources, and then discover that the other resource is needed as well. As shown in Code Example 4-2, there could be a problem if two threads attempt to claim both resources but lock the associated mutexes in different orders. In this example, if the two threads lock mutexes 1 and 2 respectively, then a deadlock occurs when each attempts to lock the other mutex.

Code Example 4-2 Deadlock

Thread 1	Thread 2

`pthread_mutex_lock(&m1);`
`/* use resource 1 */`
`pthread_mutex_lock(&m2);`
`/* use resources 1 and 2 */`
`pthread_mutex_unlock(&m2);`
`pthread_mutex_unlock(&m1);`
	`pthread_mutex_lock(&m2);`
	`/* use resource 2 */`
	`pthread_mutex_lock(&m1);`
	`/* use resources 1 and 2 */`
	`pthread_mutex_unlock(&m1);`
	`pthread_mutex_unlock(&m2);`

The best way to avoid this problem is to make sure that whenever threads lock multiple mutexes, they do so in the same order. This technique is known as *lock hierarchies*: order the mutexes by logically assigning numbers to them.

Also, honor the restriction that you cannot take a mutex that is assigned *i* when you are holding any mutex assigned a number greater than *i*.

Note – The `lock_lint` tool can detect the sort of deadlock problem shown in this example. The best way to avoid such deadlock problems is to use lock hierarchies. When locks are always taken in a prescribed order, deadlock should not occur.

However, this technique cannot always be used—sometimes you must take the mutexes in an order other than prescribed. To prevent deadlock in such a situation, use `pthread_mutex_trylock()`. One thread must release its mutexes when it discovers that deadlock would otherwise be inevitable.

Code Example 4-3 shows how this is done.

Code Example 4-3 Conditional Locking

Thread 1	Thread 2

`pthread_mutex_lock(&m1);`
`  pthread_mutex_lock(&m2);`
 	
`  pthread_mutex_unlock(&m2);`
 	
`pthread_mutex_unlock(&m1);`
	`for (;;) {`
	`  pthread_mutex_lock(&m2);`
	`  if (pthread_mutex_trylock(&m1)==0)`
	`     /* got it! */`
	`     break;`
	
	`  /* didn't get it */`
	`  pthread_mutex_unlock(&m2);`
	`}`
	`pthread_mutex_unlock(&m1);`
	`pthread_mutex_unlock(&m2);`

In this example, thread 1 locks mutexes in the prescribed order, but thread 2 takes them out of order. To make certain that there is no deadlock, thread 2 has to take mutex 1 very carefully; if it were to block waiting for the mutex to be released, it is likely to have just entered into a deadlock with thread 1.

To ensure this does not happen, thread 2 calls `pthread_mutex_trylock()`, which takes the mutex if it is available. If it is not, thread 2 returns immediately, reporting failure. At this point, thread 2 must release mutex 2, so that thread 1 can lock it, and then release both mutex 1 and mutex 2.

Nested Locking With a Singly Linked List

Code Example 4-4 and Code Example 4-5 show how to take three locks at once, but prevent deadlock by taking the locks in a prescribed order.)

Code Example 4-4 Singly Linked List Structure

```
typedef struct node1 {
    int value;
    struct node1 *link;
    pthread_mutex_t lock;
} node1_t;

node1_t ListHead;
```

This example uses a singly-linked list structure with each node containing a mutex. To remove a node from the list, first search the list starting at ListHead (which itself is never removed) until the desired node is found.

To protect this search from the effects of concurrent deletions, lock each node before any of its contents are accessed. Because all searches start at ListHead, there is never a deadlock because the locks are always taken in list order.

When the desired node is found, lock both the node and its predecessor since the change involves both nodes. Because the predecessor's lock is always taken first, you are again protected from deadlock. Here is the C code to remove an item from a singly linked list.

Code Example 4-5 Singly-Linked List with Nested Locking

```
node1_t *delete(int value)
{
    node1_t *prev, *current;

    prev = &ListHead;
    pthread_mutex_lock(&prev->lock);
    while ((current = prev->link) != NULL) {
        pthread_mutex_lock(&current->lock);
        if (current->value == value) {
            prev->link = current->link;
            pthread_mutex_unlock(&current->lock);
            pthread_mutex_unlock(&prev->lock);
            current->link = NULL;
            return(current);
        }
        pthread_mutex_unlock(&prev->lock);
        prev = current;
    }
    pthread_mutex_unlock(&prev->lock);
    return(NULL);
}
```

Nested Locking With a Circular Linked List

Code Example 4-6 modifies the previous list structure by converting it into a circular list. There is no longer a distinguished head node; now a thread might be associated with a particular node and might perform operations on that node and its neighbor. Note that lock hierarchies do not work easily here because the obvious hierarchy (following the links) is circular.

Code Example 4-6 Circular Linked List Structure

```
typedef struct node2 {
    int value;
    struct node2 *link;
    pthread_mutex_t lock;
} node2_t;
```

Here is the C code that acquires the locks on two nodes and performs an operation involving both of them.

Code Example 4-7 Circular Linked List With Nested Locking

```
void Hit Neighbor(node2_t *me) {
    while (1) {
        pthread_mutex_lock(&me->lock);
        if (pthread_mutex_lock(&me->link->lock)!= 0) {
            /* failed to get lock */
            pthread_mutex_unlock(&me->lock);
            continue;
        }
        break;
    }
    me->link->value += me->value;
    me->value /=2;
    pthread_mutex_unlock(&me->link->lock);
    pthread_mutex_unlock(&me->lock);
}
```

Condition Variable Attributes

Use condition variables to atomically block threads until a particular condition is true. Always use condition variables together with a mutex lock.

With a condition variable, a thread can atomically block until a condition is satisfied. The condition is tested under the protection of a mutual exclusion lock (mutex).

When the condition is false, a thread usually blocks on a condition variable and atomically releases the mutex waiting for the condition to change. When another thread changes the condition, it can signal the associated condition variable to cause one or more waiting threads to wake up, reacquire the mutex, and reevaluate the condition.

Condition variables can be used to synchronize threads among processes when they are allocated in memory that is writable and shared by the cooperating processes.

The scheduling policy determines how blocking threads are awakened. For the default `SCHED_OTHER`, threads are awakened in priority order.

The attributes for condition variables must be set and initialized before the condition variables can be used. The functions that manipulate condition variable attributes are listed in Table 4-4.

Table 4-4 Condition Variable Attributes

Initialize a Condition Variable Attribute	*pthread_condattr_init(3T)*	*page 88*
Remove a Condition Variable Attribute	*pthread_condattr_destroy(3T)*	*page 89*
Set the Scope of a Condition Variable	*pthread_condattr_setpshared(3T)*	*page 90*
Get the Scope of a Condition Variable	*pthread_condattr_getpshared(3T)*	*page 91*

The differences from the original Solaris threads in defining the scope of a condition variable are shown in Table 4-5.

Table 4-5 Condition Variable Scope Comparison

Solaris	POSIX	Definition
USYNC_PROCESS	PTHREAD_PROCESS_SHARED	Use to synchronize threads in this and other processes
USYNC_THREAD	PTHREAD_PROCESS_PRIVATE	Use to synchronize threads in this process only

Initialize a Condition Variable Attribute

pthread_condattr_init(3T)

Use `pthread_condattr_init()` to initialize attributes associated with this object to their default values. Storage for each attribute object is allocated by the threads system during execution. The default value of the *pshared* attribute when this function is called is `PTHREAD_PROCESS_PRIVATE`, which means that the initialized condition variable can be used within a process.

```
Prototype:
int pthread_condattr_init(pthread_condattr_t *cattr);
```

```
#include <pthread.h>
#include <time.h>

pthread_condattr_t cattr;
int ret;

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

cattr is an opaque data type that contains a system-allocated attribute object. The possible values of *cattr*'s scope are `PTHREAD_PROCESS_PRIVATE` (the default) and `PTHREAD_PROCESS_SHARED`.

Before a condition variable attribute can be reused, it must first be removed by pthread_condattr_destroy(3T). The pthread_condattr_init() call returns a pointer to an opaque object. If the object is not destroyed, a memory leak will result.

Return Values

Returns zero after completing successfully. Any other returned value indicates that an error occurred. When either of the following conditions occurs, the function fails and returns the corresponding value.

ENOMEM – There is not enough memory to initialize the thread attributes object.

EINVAL – The value specified by *cattr* is invalid.

Remove a Condition Variable Attribute

pthread_condattr_destroy(3T)

Use this routine to remove storage and render the attribute object invalid.

```
Prototype:
int pthread_condattr_destroy(pthread_condattr_t *cattr);
```

```
#include <pthread.h>
#include <time.h>

pthread_condattr_t cattr;
int ret;

/* destroy an attribute */
ret = pthread_condattr_destroy(&cattr);
```

Return Values

Returns zero after completing successfully. Any other returned value indicates that an error occurred. If the following condition occurs, the function fails and returns the corresponding value.

EINVAL – The value specified by *cattr* is invalid.

Set the Scope of a Condition Variable

pthread_condattr_setpshared(3T)

The scope of a condition variable can be either process private (intraprocess) or system wide (interprocess). If the condition variable is created with the *pshared* attribute set to the PTHREAD_PROCESS_SHARED 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() in the original Solaris threads.

If the mutex *pshared* attribute is set to PTHREAD_PROCESS_PRIVATE, only those threads created by the same process can operate on the mutex. Using PTHREAD_PROCESS_PRIVATE results in the same behavior as with the USYNC_THREAD flag in the original Solaris threads cond_init() call, which is that of a local condition variable. PTHREAD_PROCESS_SHARED is equivalent to a global condition variable.

```
Prototype:
int pthread_condattr_setpshared(pthread_condattr_t *cattr,
    int pshared);
```

```
#include <pthread.h>
#include <time.h>

pthread_condattr_t cattr;
int ret;

/* all processes */
ret = pthread_condattr_setpshared(&cattr, PTHREAD_PROCESS_SHARED);

/* within a process */
ret = pthread_condattr_setpshared(&cattr, PTHREAD_PROCESS_PRIVATE);
```

Return Values

Returns zero after completing successfully. Any other returned value indicates that an error occurred. If the following condition occurs, the function fails and returns the corresponding value.

EINVAL – The value of *cattr* is invalid, or the *pshared* value is invalid.

Get the Scope of a Condition Variable

pthread_condattr_getpshared(3T)

Get the current value of *pshared* for the attribute object *cattr*. The value is either `PTHREAD_PROCESS_SHARED` or `PTHREAD_PROCESS_PRIVATE`.

```
Prototype:
int pthread_condattr_getpshared(const pthread_condattr_t *cattr,
    int *pshared);
```

```
#include <pthread.h>
#include <time.h>

pthread_condattr_t cattr;
int pshared;
int ret;

/* get pshared value of condition variable */
ret = pthread_condattr_getpshared(&cattr, &pshared);
```

Return Values

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

`EINVAL` – The value of *cattr* is invalid.

4

Using Condition Variables

This section explains using condition variables. Table 4-6 lists the functions that are available.

Table 4-6 Condition Variables Functions

Initialize a Condition Variable	*pthread_cond_init(3T)*	*page 92*
Block on a Condition Variable	*pthread_cond_wait(3T)*	*page 94*
Unblock a Specific Thread	*pthread_cond_signal(3T)*	*page 96*
Block Until a Specified Event	*pthread_cond_timedwait(3T)*	*page 98*
Unblock All Threads	*pthread_cond_broadcast(3T)*	*page 99*
Destroy Condition Variable State	*pthread_cond_destroy(3T)*	*page 101*

Initialize a Condition Variable

pthread_cond_init(3T)

Use `pthread_cond_init()` to initialize the condition variable pointed at by *cv* to its default value (*cattr* is NULL), or to specify condition variable attributes that are already set with `pthread_condattr_init()`. The effect of *cattr* being NULL is the same as passing the address of a default condition variable attribute object, but without the memory overhead.

```
Prototype:
int pthread_cond_init(pthread_cond_t *cv,
    const pthread_condattr_t *cattr);
```

```
#include <pthread.h>

pthread_cond_t cv;
pthread_condattr_t cattr;
int ret;

/* initialize a condition variable to its default value */
ret = pthread_cond_init(&cv, NULL);

/* initialize a condition variable */
ret = pthread_cond_init(&cv, &cattr);
```

Statically-defined condition variables can be initialized directly to have default attributes with the macro PTHREAD_COND_INITIALIZER. This has the same effect as dynamically allocating pthread_cond_init() with null attributes. No error checking is done.

Multiple threads must not simultaneously initialize or reinitialize the same condition variable. If a condition variable is reinitialized or destroyed, the application must be sure the condition variable is not currently in use.

If a condition variable is dynamically allocated and was initialized with an attribute object, before the condition variable itself is freed, its attribute object should first be freed with pthread_condattr_destroy() .

Return Values

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 – The value specified by *cattr* is invalid.

EBUSY – The condition variable is being used.

EAGAIN – The necessary resources are not available.

ENOMEM – There is not enough memory to initialize the condition variable.

Block on a Condition Variable

pthread_cond_wait(3T)

Use this routine 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*.

Prototype:

```
int pthread_cond_wait(pthread_cond_t *cv,pthread_mutex_t *mutex);
```

```
#include <pthread.h>

pthread_cond_t cv;
pthread_mutex_t mp;
int ret;

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

The blocked thread can be awakened by a `pthread_cond_signal()`, a `pthread_cond_broadcast()`, or when interrupted by delivery of a signal.

Any change in the value of a condition associated with the condition variable cannot be inferred by the return of `pthread_cond_wait()`, and any such condition must be reevaluated.

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

This function blocks until the condition is signaled. It atomically releases the associated mutex lock before blocking, and atomically reacquires it before returning.

In typical use, a condition expression is evaluated under the protection of a mutex lock. When the condition expression is false, the thread blocks on the condition variable. The condition variable is then signaled by another thread when it changes the condition value. This causes one or all of the threads waiting on the condition to unblock and to try to reacquire the mutex lock.

Because the condition can change before an awakened thread returns from `pthread_cond_wait()`, the condition that caused the wait must be retested before the mutex lock is acquired. The recommended test method is to write the condition check as a while loop that calls `pthread_cond_wait()`.

```
pthread_mutex_lock();
    while(condition_is_false)
        pthread_cond_wait();
pthread_mutex_unlock();
```

No specific order of acquisition is guaranteed when more than one thread blocks on the condition variable.

Note – `pthread_cond_wait()` is a cancellation point. If a cancel is pending and the calling thread has cancellation enabled, the thread will be terminated and will begin executing its cleanup handlers.

Return Values

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

`EINVAL` – The value specified by *cv* or *mp* is invalid.

Unblock a Specific Thread

pthread_cond_signal(3T)

Use `pthread_cond_signal()` to unblock one thread that is blocked on the condition variable pointed to by *cv*.

```
Prototype:
int pthread_cond_signal(pthread_cond_t *cv);
```

```
#include <pthread.h>

pthread_cond_t cv;
int ret;

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

Call `pthread_cond_signal()` under the protection of the same mutex used with the condition variable being signaled. Otherwise, the condition variable could be signaled between the test of the associated condition and blocking in `pthread_cond_wait()`, which can cause an infinite wait.

The scheduling policy determines the order in which blocked threads are awakened. For `SCHED_OTHER`, threads are awakened in priority order.

When no threads are blocked on the condition variable, then calling `pthread_cond_signal()` has no effect.

Return Values

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

`EINVAL` – *cv* points to an illegal address.

Code Example 4-8 Using `pthread_cond_wait()` and `pthread_cond_signal()`

```
pthread_mutex_t count_lock;
pthread_cond_t count_nonzero;
unsigned int count;

decrement_count()
{
    pthread_mutex_lock(&count_lock);
    while (count == 0)
        pthread_cond_wait(&count_nonzero, &count_lock);
    count = count - 1;
    pthread_mutex_unlock(&count_lock);
}
increment_count()
{
    pthread_mutex_lock(&count_lock);
    if (count == 0)
        pthread_cond_signal(&count_nonzero);
    count = count + 1;
    pthread_mutex_unlock(&count_lock);
}
```

Block Until a Specified Event

pthread_cond_timedwait(3T)

```
Prototype:
int pthread_cond_timedwait(pthread_cond_t *cv,
    pthread_mutex_t *mp, const struct timespec *abstime);
```

```
#include <pthread.h>

pthread_cond_t cv;
pthread_mutex_t mp;
timestruct_t abstime;
int ret;

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

Use `pthread_cond_timedwait()` as you would use `pthread_cond_wait()`, except that `pthread_cond_timedwait()` does not block past the time of day specified by *abstime*. `pthread_cond_timedwait()` always returns with the mutex locked and owned by the calling thread even when it is returning an error.

The `pthread_cond_timedwait()` function blocks until the condition is signaled or until the time of day specified by the last argument has passed.

Note – `pthread_cond_timedwait()` is also a cancellation point.

Return Values

Returns zero after completing successfully. Any other returned value indicates that an error occurred. When either of the following conditions occurs, the function fails and returns the corresponding value.

`EINVAL` – *cv* or *abstime* points to an illegal address.

`ETIMEDOUT` – The time specified by *abstime* has passed.

The time-out is specified as a time of day so that the condition can be retested efficiently without recomputing the value, as shown in Code Example 4-9.

Code Example 4-9 Timed Condition Wait

```
pthread_timestruc_t to;
pthread_mutex_t m;
pthread_cond_t c;
...
pthread_mutex_lock(&m);
to.tv_sec = time(NULL) + TIMEOUT;
to.tv_nsec = 0;
while (cond == FALSE) {
    err = pthread_cond_timedwait(&c, &m, &to);
    if (err == ETIME) {
        /* timeout, do something */
        break;
    }
}
pthread_mutex_unlock(&m);
```

Unblock All Threads

pthread_cond_broadcast(3T)

```
Prototype:
int pthread_cond_broadcast(pthread_cond_t *cv);
```

```
#include <pthread.h>

pthread_cond_t cv;
int ret;

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

Use `pthread_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, `pthread_cond_broadcast()` has no effect.

This function wakes all the threads blocked in `pthread_cond_wait()`.

Return Values

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

`EINVAL` – *cv* points to an illegal address.

Condition Variable Broadcast Example

Since `pthread_cond_broadcast()` causes all threads blocked on the condition to contend again for the mutex lock, use it with care. For example, use `pthread_cond_broadcast()` to allow threads to contend for varying resource amounts when resources are freed, as shown in Code Example 4-10.

Code Example 4-10 Condition Variable Broadcast

```
pthread_mutex_t rsrc_lock;
pthread_cond_t rsrc_add;
unsigned int resources;

get_resources(int amount)
{
    pthread_mutex_lock(&rsrc_lock);
    while (resources < amount) {
        pthread_cond_wait(&rsrc_add, &rsrc_lock);
    }
    resources -= amount;
    pthread_mutex_unlock(&rsrc_lock);
}

add_resources(int amount)
{
    pthread_mutex_lock(&rsrc_lock);
    resources += amount;
    pthread_cond_broadcast(&rsrc_add);
    pthread_mutex_unlock(&rsrc_lock);
}
```

Note that in `add_resources()` it does not matter whether *resources* is updated first or `pthread_cond_broadcast()` is called first inside the mutex lock.

Call pthread_cond_broadcast() under the protection of the same mutex that is used with the condition variable being signaled. Otherwise, the condition variable could be signaled between the test of the associated condition and blocking in pthread_cond_wait(), which can cause an infinite wait.

Destroy Condition Variable State

pthread_cond_destroy(3T)

Use pthread_cond_destroy() to destroy any state associated with the condition variable pointed to by *cv*.

```
Prototype:
int pthread_cond_destroy(pthread_cond_t *cv);
```

```
#include <pthread.h>

pthread_cond_t cv;
int ret;

/* Condition variable is destroyed */
ret = pthread_cond_destroy(&cv);
```

Note that the space for storing the condition variable is not freed.

Return Values

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.

EBUSY – The object has been initialized before, and is not destroyed.

EINVAL – The value specified by *cv* is invalid.

The Lost Wake-Up Problem

Calling `pthread_cond_signal()` or `pthread_cond_broadcast()` when the thread does not hold the mutex lock associated with the condition can lead to *lost wake-up* bugs.

A lost wake-up occurs when

- A thread calls `pthread_cond_signal()` or `pthread_cond_broadcast()`
- *And* another thread is between the test of the condition and the call to `pthread_cond_wait()`
- *And* no threads are waiting.

The signal has no effect, and therefore is lost.

The Producer/Consumer Problem

This problem is one of the small collection of standard, well-known problems in concurrent programming: a finite-size buffer and two classes of threads, *producers* and *consumers*, put items into the buffer (producers) and take items out of the buffer (consumers).

A producer must wait until the buffer has space before it can put something in, and a consumer must wait until something is in the buffer before it can take something out.

A condition variable represents a queue of threads waiting for some condition to be signaled.

Code Example 4-11 has two such queues, one (`less`) for producers waiting for a slot in the buffer, and the other (`more`) for consumers waiting for a buffer slot containing information. The example also has a mutex, as the data structure describing the buffer must be accessed by only one thread at a time.

This is the code for the buffer data structure.

Code Example 4-11 The Producer/Consumer Problem and Condition Variables

```
typedef struct {
    char buf[BSIZE];
    int occupied;
    int nextin;
    int nextout;
    mutex_t mutex;
    cond_t more;
    cond_t less;
} buffer_t;

buffer_t buffer;
```

As Code Example 4-12 on page 104 shows, the producer thread takes the mutex protecting the `buffer` data structure and then makes certain that space is available for the item being produced. If not, it calls `pthread_cond_wait()`, which causes it to join the queue of threads waiting for the condition `less`, representing *there is room in the buffer*, to be signaled.

At the same time, as part of the call to `pthread_cond_wait()`, the thread releases its lock on the mutex. The waiting producer threads depend on consumer threads to signal when the condition is true (as shown in Code Example 4-12). When the condition is signaled, the first thread waiting on `less` is awakened. However, before the thread can return from `pthread_cond_wait()`, it must reacquire the lock on the mutex.

This ensures that it again has mutually exclusive access to the buffer data structure. The thread then must check that there really is room available in the buffer; if so, it puts its item into the next available slot.

At the same time, consumer threads might be waiting for items to appear in the buffer. These threads are waiting on the condition variable `more`. A producer thread, having just deposited something in the buffer, calls `pthread_cond_signal()` to wake up the next waiting consumer. (If there are no waiting consumers, this call has no effect.)

Finally, the producer thread unlocks the mutex, allowing other threads to operate on the buffer data structure.

Code Example 4-12 The Producer/Consumer Problem – the Producer

```
void producer(buffer_t *b, char item)
{
    pthread_mutex_lock(&b->mutex);

    while (b->occupied >= BSIZE)
        pthread_cond_wait(&b->less, &b->mutex);

    assert(b->occupied < BSIZE);

    b->buf[b->nextin++] = item;

    b->nextin %= BSIZE;
    b->occupied++;

    /* now: either b->occupied < BSIZE and b->nextin is the index
       of the next empty slot in the buffer, or
       b->occupied == BSIZE and b->nextin is the index of the
       next (occupied) slot that will be emptied by a consumer
       (such as b->nextin == b->nextout) */

    pthread_cond_signal(&b->more);

    pthread_mutex_unlock(&b->mutex);
}
```

Note the use of the `assert()` statement; unless the code is compiled with `NDEBUG` defined, `assert()` does nothing when its argument evaluates to true (that is, nonzero), but causes the program to abort if the argument evaluates to false (zero). Such assertions are especially useful in multithreaded programs—they immediately point out runtime problems if they fail, and they have the additional effect of being useful comments.

The comment a few lines later could better be expressed as an assertion, but it is too complicated as a Boolean-valued expression and so is given in English.

Both the assertion and the comments are examples of *invariants*. These are logical statements that should not be falsified by the execution of the program, except during brief moments when a thread is modifying some of the program variables mentioned in the invariant. (An assertion, of course, should be true whenever any thread executes it.)

Using invariants is an extremely useful technique. Even if they are not stated in the program text, think in terms of invariants when you analyze a program.

The invariant in the producer code that is expressed as a comment is always true whenever a thread is in the part of the code where the comment appears. If you move this comment to just after the `mutex_unlock()`, this does not necessarily remain true. If you move this comment to just after the `assert`, this is still true.

The point is that this invariant expresses a property that is true at all times, except when either a producer or a consumer is changing the state of the buffer. While a thread is operating on the buffer (under the protection of a mutex), it might temporarily falsify the invariant. However, once the thread is finished, the invariant should be true again.

Code Example 4-13 shows the code for the consumer. Its flow is symmetric with that of the producer.

Code Example 4-13 The Producer/Consumer Problem – the Consumer

```
char consumer(buffer_t *b)
{
    char item;
    pthread_mutex_lock(&b->mutex);
    while(b->occupied <= 0)
        pthread_cond_wait(&b->more, &b->mutex);

    assert(b->occupied > 0);

    item = b->buf[b->nextout++];
    b->nextout %= BSIZE;
    b->occupied--;

    /* now: either b->occupied > 0 and b->nextout is the index
       of the next occupied slot in the buffer, or
       b->occupied == 0 and b->nextout is the index of the next
       (empty) slot that will be filled by a producer (such as
       b->nextout == b->nextin) */

    pthread_cond_signal(&b->less);
    pthread_mutex_unlock(&b->mutex);

    return(item);
}
```

4

Semaphores

Semaphores are a programming construct designed by E. W. Dijkstra in the late 1960s. Dijkstra's model was the operation of railroads: consider a stretch of railroad in which there is a single track over which only one train at a time is allowed.

Guarding this track is a semaphore. A train must wait before entering the single track until the semaphore is in a state that permits travel. When the train enters the track, the semaphore changes state to prevent other trains from entering the track. A train that is leaving this section of track must again change the state of the semaphore to allow another train to enter.

In the computer version, a semaphore appears to be a simple integer. A thread waits for permission to proceed and then signals that it has proceeded by performing a P operation on the semaphore.

The semantics of the operation are such that the thread must wait until the semaphore's value is positive, then change the semaphore's value by subtracting one from it. When it is finished, the thread performs a V operation, which changes the semaphore's value by adding one to it. It is crucial that these operations take place *atomically*—they cannot be subdivided into pieces between which other actions on the semaphore can take place. In the P operation, the semaphore's value must be positive just before it is decremented (resulting in a value that is guaranteed to be nonnegative and one less than what it was before it was decremented).

In both P and V operations, the arithmetic must take place without interference. If two V operations are performed simultaneously on the same semaphore, the net effect should be that the semaphore's new value is two greater than it was.

The mnemonic significance of P and V is lost on most of the world, as Dijkstra is Dutch. However, in the interest of true scholarship: P stands for *prolagen*, a made-up word derived from *proberen te verlagen*, which means *try to decrease*. V stands for *verhogen*, which means *increase*. This is discussed in one of Dijkstra's technical notes, *EWD 74*.

`sem_wait(3T)` and `sem_post(3T)` correspond to Dijkstra's P and V operations. `sem_trywait(3T)` is a conditional form of the P operation: if the calling thread cannot decrement the value of the semaphore without waiting, the call returns immediately with a nonzero value.

There are two basic sorts of semaphores: *binary* semaphores, which never take on values other than zero or one, and *counting* semaphores, which can take on arbitrary nonnegative values. A binary semaphore is logically just like a mutex.

However, although it is not enforced, mutexes should be unlocked only by the thread holding the lock. There is no notion of “the thread holding the semaphore,” so any thread can perform a V (or `sem_post(3T)`) operation.

Counting semaphores are about as powerful as conditional variables (used in conjunction with mutexes). In many cases, the code might be simpler when it is implemented with counting semaphores rather than with condition variables (as shown in the next few examples).

However, when a mutex is used with condition variables, there is an implied bracketing—it is clear which part of the program is being protected. This is not necessarily the case for a semaphore, which might be called the *go to* of concurrent programming—it is powerful but too easy to use in an unstructured, unfathomable way.

Counting Semaphores

Conceptually, a semaphore is a nonnegative integer count. Semaphores are typically used to coordinate access to resources, with the semaphore count initialized to the number of free resources. Threads then atomically increment the count when resources are added and atomically decrement the count when resources are removed.

When the semaphore count becomes zero, indicating that no more resources are present, threads trying to decrement the semaphore block wait until the count becomes greater than zero.

Table 4-7 Routines for Semaphores

Initialize a Semaphore	*sem_init(3R)*	*page 108*
Increment a Semaphore	*sem_post(3R)*	*page 110*
Block on a Semaphore Count	*sem_wait(3R)*	*page 111*
Decrement a Semaphore Count	*sem_trywait(3R)*	*page 112*
Destroy the Semaphore State	*sem_destroy(3R)*	*page 113*

Because semaphores need not be acquired and released by the same thread, they can be used for asynchronous event notification (such as in signal handlers). And, because semaphores contain state, they can be used asynchronously without acquiring a mutex lock as is required by condition variables. However, semaphores are not as efficient as mutex locks.

By default, there is no defined order of unblocking if multiple threads are waiting for a semaphore.

Semaphores must be initialized before use, but they do not have attributes.

Initialize a Semaphore

sem_init(3R)

```
Prototype:
int     sem_init(sem_t *sem, int pshared, unsigned int value);
```

```
#include <semaphore.h>

sem_t sem;
int pshared;
int ret;
int value;

/* initialize the semaphore */
ret = sem_init(&sem, pshared, value);
```

Use `sem_init()` to initialize the semaphore variable pointed to by *sem* by *value* amount. If the value of *pshared* is zero, then the semaphore cannot be shared between processes. If the value of *pshared* is nonzero, then the semaphore can be shared between processes.

Multiple threads must not initialize the same semaphore simultaneously.

A semaphore must not be reinitialized while other threads might be using it.

Return Values

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 – The value argument exceeds SEM_VALUE_MAX.

ENOSPC – A resource required to initialize the semaphore has been exhausted. The limit on semaphores SEM_NSEMS_MAX has been reached.

EPERM – The process lacks the appropriate privileges to initialize the semaphore.

Initializing Semaphores With Intraprocess Scope

When *pshared* is 0, the semaphore can be used by all the threads in this process, only.

```
#include <semaphore.h>

sem_t sem;
int ret;
int count = 4;

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

Initializing Semaphores With Interprocess Scope

When *pshared* is nonzero, the semaphore can be shared by other processes.

```
#include <semaphore.h>

sem_t sem;
int ret;
int count = 4;

/* to be used within this process only */
ret = sem_init(&sem, 1, count);
```

Named Semaphores

The functions `sem_open(3R)`, `sem_getvalue(3R)`, `sem_close(3R)`, and `sem_unlink(3R)` are available to open, retrieve, close, and remove named semaphores. Using `sem_open()`, you can create a semaphore that has a name defined in the filesystem name space.

Named semaphores are like process shared semaphores, except that they are referenced with a pathname rather than a *pshared* value.

For more information about named semaphores, see `sem_open(3R)`, `sem_getvalue(3R)`, `sem_close(3R)`, and `sem_unlink(3R)`.

Increment a Semaphore

sem_post(3R)

```
Prototype:
int     sem_post(sem_t *sem);
```

```
#include <semaphore.h>

sem_t sem;
int ret;

ret = sem_post(&sem); /* semaphore is posted */
```

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

Return Values

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

`EINVAL` – *sem* points to an illegal address.

Block on a Semaphore Count

sem_wait(3R)

```
Prototype:
int     sem_wait(sem_t *sem);
```

```
#include <semaphore.h>

sem_t sem;
int ret;

ret = sem_wait(&sem); /* wait for semaphore */
```

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

Return Values

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` – *sem* points to an illegal address.

`EINTR` – A signal interrupted this function.

`EDEADLK` – A deadlock condition was detected.

Decrement a Semaphore Count

sem_trywait(3R)

```
Prototype:
int     sem_trywait(sem_t *sem);
```

```
#include <semaphore.h>

sem_t sem;
int ret;

ret = sem_trywait(&sem); /* try to wait for semaphore*/
```

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

Return Values

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` – *sem* points to an illegal address.

`EINTR` – A signal interrupted this function.

`EDEADLK` – A deadlock condition was detected.

`EAGAIN` – The semaphore was already locked, so it cannot be immediately locked by the `sem_trywait()` operation.

Destroy the Semaphore State

sem_destroy(3R)

```
Prototype:
int     sem_destroy(sem_t *sem);
```

```
#include <semaphore.h>

sem_t sem;
int ret;

ret = sem_destroy(&sem); /* the semaphore is destroyed */
```

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

Return Values

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

`EINVAL` – *sem* points to an illegal address.

The Producer/Consumer Problem, Using Semaphores

The data structure in Code Example 4-14 is similar to that used for the solution with condition variables (see page 84). Two semaphores represent the number of full and empty buffers and ensure that producers wait until there are empty buffers and that consumers wait until there are full buffers.

Code Example 4-14 The Producer/Consumer Problem With Semaphores

```
typedef struct {
    char buf[BSIZE];
    sem_t occupied;
    sem_t empty;
    int nextin;
    int nextout;
    sem_t pmut;
    sem_t cmut;
} buffer_t;

buffer_t buffer;

sem_init(&buffer.occupied, 0, 0);
sem_init(&buffer.empty,0, BSIZE);
sem_init(&buffer.pmut, 0, 1);
sem_init(&buffer.cmut, 0, 1);
buffer.nextin = buffer.nextout = 0;
```

Another pair of (binary) semaphores plays the same role as mutexes, controlling access to the buffer when there are multiple producers and multiple empty buffer slots, and when there are multiple consumers and multiple full buffer slots. Mutexes would work better here, but would not provide as good an example of semaphore use.

Code Example 4-15 The Producer/Consumer Problem – the Producer

```
void producer(buffer_t *b, char item) {
    sem_wait(&b->empty);

    sem_wait(&b->pmut);

    b->buf[b->nextin] = item;
    b->nextin++;
    b->nextin %= BSIZE;

    sem_post(&b->pmut);

    sem_post(&b->occupied);
}
```

Code Example 4-16 The Producer/Consumer Problem – the Consumer

```
char consumer(buffer_t *b) {
    char item;

    sem_wait(&b->occupied);

    sem_wait(&b->cmut);

    item = b->buf[b->nextout];
    b->nextout++;
    b->nextout %= BSIZE;

    sem_post(&b->cmut);

    sem_post(&b->empty);

    return(item);
}
```

4

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, after the primitive has been initialized with its shared attribute set as interprocess.

Producer/Consumer Problem Example

Code Example 4-17 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.

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 4-17 is the same as that used for the solution with condition variables (see page 84). Two semaphores represent the number of full and empty buffers and ensure that producers wait until there are empty buffers and that consumers wait until there are full buffers.

Code Example 4-17 Synchronization Across Process Boundaries

```
main() {
    int zfd;
    buffer_t *buffer;
    pthread_mutexattr_t mattr;
    pthread_condattr_t cvattr_less, cvattr_more;

    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_attr_init(&mattr);
    pthread_mutexattr_setpshared(&mattr,
PTHREAD_PROCESS_SHARED);
    mutex_init(&buffer->lock, &mattr);
```

Code Example 4-17 Synchronization Across Process Boundaries

```
    pthread_condattr_init(cvattr_less);
    pthread_condattr_setpshared(&cvattr_less,
        PTHREAD_PROCESS_SHARED);
    pthread_cond_init(&buffer->less, &cvattr_less);

    pthread_condattr_init(cvattr_more);
    pthread_condattr_setpshared(&cvattr_more,
        PTHREAD_PROCESS_SHARED);
    pthread_cond_init(&buffer->more, &cvattr_more);

    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);
    }
}
```

```
void consumer_driver(buffer_t *b) {
    char item;

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

Interprocess Locking Without the Threads Library

Although not generally recommended, it is possible in Solaris threads to do interprocess locking without using the threads library. If this is something you want to do, see the instructions in "Using LWPs Between Processes" on page 220.

Comparing Primitives

The most basic synchronization primitive in threads is the mutual exclusion lock. So, it is the most efficient mechanism in both memory use and execution time. The basic use of a mutual exclusion lock is to serialize access to a resource.

The next most efficient primitive in threads is the condition variable. The basic use of a condition variable is to block on a change of state. Remember that a mutex lock must be acquired before blocking on a condition variable and must be unlocked after returning from `pthread_cond_wait(3T)`. The mutex lock must also be held across the change of state that occurs before the corresponding call to `pthread_cond_signal(3T)`.

The semaphore uses more memory than the condition variable. It is easier to use in some circumstances because a semaphore variable functions on state rather than on control. Unlike a lock, a semaphore does not have an owner. Any thread can increment a semaphore that has blocked.