# *Programming With Synchronization Objects* 3

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


| | |
|---|---|
| *Mutual Exclusion Locks* | *page 40* |
| *Condition Variables* | *page 51* |
| *Multiple-Readers, Single-Writer Locks* | *page 63* |
| *Semaphores* | *page 69* |
| *Synchronization Across Process Boundaries* | *page 78* |
| *Comparing Primitives* | *page 80* |


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

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

The types of synchronization objects are:

- Mutex Locks
- Condition Variables
- Readers/Writer Locks
- Semaphores

# 3

Here are some multithreading situations in which synchronization is important.

- Threads in two or more processes can use a single synchronization variable jointly. Note that the synchronization variable should be initialized by only one of the cooperating processes, as reinitializing a synchronization variable sets it to the *unlocked* state.
- Synchronization is the only way to ensure consistency of shared data.
- A process can map a file and have a thread in this process get a record's lock. When the modification is done, the thread releases the lock and unmaps the file. 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.
- Synchronization can ensure the safety of mutable data.
- Synchronization can be important 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.

## *Mutual Exclusion Locks*

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.

*Table 3-1* Routines for Mutual Exclusion Locks

| Routine | Operation | Page |
|---|---|---|
| *mutex_init(3T)* | *Initialize a Mutual Exclusion Lock* | *page 41* |
| *mutex_lock(3T)* | *Lock a Mutex* | *page 42* |
| *mutex_trylock(3T)* | *Lock With a Nonblocking Mutex* | *page 42* |
| *mutex_unlock(3T)* | *Unlock a Mutex* | *page 43* |
| *mutex_destroy(3T)* | *Destroy Mutex State* | *page 44* |

Mutexes can be used to synchronize threads in processes when they are allocated in memory that is writable and shared among the cooperating processes (see `mmap`(2)). They must be initialized for this behavior.

Mutexes must be initialized before use.

Note that there is no defined order of acquisition when multiple threads are waiting for a mutex.

## *Initialize a Mutual Exclusion Lock*

### **`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.

**Return Values—**`mutex_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 *mp* or *arg* points to an illegal address.

## Lock a Mutex

### mutex_lock(3T)

```
#include <synch.h>  (or #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). When mutex_lock() returns, the mutex is locked and the calling thread is the owner.

**Return Values—**mutex_lock() 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 *mp* points to an illegal address.

## Lock With a Nonblocking Mutex

### mutex_trylock(3T)

```
#include <synch.h>  (or #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(). 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—**`mutex_trylock`() 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 *mp* points to an illegal address.

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

## *Unlock a Mutex*

### `mutex_unlock`*(3T)*

```
#include <synch.h>  (or #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). When any other threads are waiting for the mutex to become available, the thread at the head of the queue is unblocked.

**Return Values—**`mutex_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 *mp* points to an illegal address.

## *Destroy Mutex State*

### `mutex_destroy`*(3T)*

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

**Return Values—**`mutex_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 *mp* points to an illegal address.

## Mutex Lock Code Example

*Code Example 3-1* Mutex Lock Example

```
mutex_t count_mutex;
int count;

increment_count()
{
    mutex_lock(&count_mutex);
    count = count + 1;
    mutex_unlock(&count_mutex);
}

int
get_count()
{
    int c;

    mutex_lock(&count_mutex);
    c = count;
    mutex_unlock(&count_mutex);
    return (c);
}
```

The two functions in Code Example 3-1 use the mutex lock for different purposes. `increment_count()` uses the mutex lock simply to assure an atomic[1] update of the shared variable. `get_count()` uses the mutex lock to guarantee that memory is synchronized when it refers to `count`.

### 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 3-2, there could be a problem if two threads attempt to claim both resources but lock the associated mutexes in

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

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 3-2* Deadlock

| **Thread 1** | **Thread 2** |
|---|---|
| `mutex_lock(&m1);`<br><br>`/* use resource 1 */`<br><br>`mutex_lock(&m2);`<br><br>`/* use resources 1 and 2 */`<br><br>`mutex_unlock(&m2);`<br>`mutex_unlock(&m1);` | `mutex_lock(&m2);`<br><br>`/* use resource 2 */`<br><br>`mutex_lock(&m1);`<br><br>`/* use resources 1 and 2 */`<br><br>`mutex_unlock(&m1);`<br>`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 cannot occur.

---

However, this technique cannot always be used—sometimes you must take the mutexes in an order other than the prescribed one. To prevent deadlock in such a situation, one thread must release the mutexes it currently holds if it discovers that deadlock would otherwise be inevitable. Code Example 3-3 shows how this is done.

*Code Example 3-3* Conditional Locking

| **Thread 1** | **Thread 2** |
|---|---|
| `mutex_lock(&m1);`<br>`  mutex_lock(&m2);`<br><br>`  mutex_unlock(&m2);`<br><br>`mutex_unlock(&m1);` | `for (;;) {`<br>`  mutex_lock(&m2);`<br>`  if (mutex_trylock(&m1)`<br>`  ==0)`<br>`  /* got it! */`<br>`  break;`<br><br>`  /* didn't get it */`<br>`  mutex_unlock(&m2);`<br>`}`<br>`mutex_unlock(&m1);`<br>`mutex_unlock(&m2);` |

In this example, thread 1 is locking the mutexes in the prescribed order, but thread 2 is taking 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 make sure this does not happen, thread 2 calls `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, then release both mutex 1 and mutex 2.

## Nested Locking With a Singly Linked List

Code Example 3-4 and Code Example 3-5 show how to take three locks at once, but prevents deadlock by taking the locks in a prescribed order.

*Code Example 3-4* Singly Linked List Structure

```
typedef struct node1 {
   int value;
   struct node1 *link;
   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 can be 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 because 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 3-5* Singly Linked List with Nested Locking

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

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

## Nested Locking With a Circular Linked List

Code Example 3-6 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 3-6* Circular Linked List Structure

```
typedef struct node2 {
    int value;
    struct node2 *link;
    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 3-7* Circular Linked List With Nested Locking

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

# Condition Variables

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

*Table 3-2* Routines for Condition Variables

| Routine | Operation | Page |
|---|---|---|
| *cond_init(3T)* | *Initialize a Condition Variable* | *page 52* |
| *cond_wait(3T)* | *Block on a Condition Variable* | *page 53* |
| *cond_signal(3T)* | *Unblock a Specific Thread* | *page 54* |
| *cond_timedwait(3T)* | *Block Until a Specified Event* | *page 55* |
| *cond_broadcast(3T)* | *Unblock All Threads* | *page 57* |
| *cond_destroy(3T)* | *Destroy Condition Variable State* | *page 58* |

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.

Always initialize condition variables before using them. Also, note that there is no defined order of unblocking when multiple threads are waiting for a condition variable.

# 3

## *Initialize a Condition Variable*

### `cond_init`*(3T)*

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

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

Use `cond_init`() to initialize the condition variable pointed to by *cvp*. 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.

**Return Values —** `cond_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 *type* is not a recognized type.

EFAULT *cvp* or *arg* points to an illegal address.

## *Block on a Condition Variable*

### `cond_wait`*(3T)*

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

int cond_wait(cond_t *cvp, 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 *cvp*. The blocked thread can be awakened by `cond_signal`(), `cond_broadcast`(), or when interrupted by delivery of a signal or a `fork`().

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

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

The 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 `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 `cond_wait`().

```
mutex_lock();
    while(condition_is_false)
        cond_wait();
mutex_unlock();
```

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

**Return Values —** `cond_wait`() 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.

EFAULT *cvp* points to an illegal address.

EINTR The wait was interrupted by a signal or a `fork`().

## *Unblock a Specific Thread*

### `cond_signal`*(3T)*

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

int cond_signal(cond_t *cvp);
```

Use `cond_signal`() to unblock one thread that is blocked on the condition variable pointed to by *cvp*. Call `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 `cond_wait`(), which can cause an infinite wait.

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

**Return Values —** `cond_signal`() 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.

EFAULT *cvp* points to an illegal address.

*Code Example 3-8* Example Using `cond_wait`(3T) and `cond_signal`(3T)

```
mutex_t count_lock;
cond_t count_nonzero;
unsigned int count;


decrement_count()
{
    mutex_lock(&count_lock);
    while (count == 0)
        cond_wait(&count_nonzero, &count_lock);
    count = count - 1;
    mutex_unlock(&count_lock);
}
increment_count()
{
    mutex_lock(&count_lock);
    if (count == 0)
        cond_signal(&count_nonzero);
    count = count + 1;
    mutex_unlock(&count_lock);
}
```

## *Block Until a Specified Event*

### `cond_timedwait`*(3T)*

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

int cond_timedwait(cond_t *cvp, mutex_t *mp,
    timestruc_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, as shown in Code Example 3-9.

**Return Values —** `cond_timedwait()` 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 specified number of seconds in *abstime* is greater than the start time of the application plus 50,000,000, or the number of nanoseconds is greater than or equal to 1,000,000,000.

EFAULT *cvp* or *abstime* points to an illegal address.

EINTR The wait was interrupted by a signal or a `fork()`.

ETIME The time specified by *abstime* has passed.

*Code Example 3-9* Timed Condition Wait

```
timestruc_t to;
mutex_t m;
cond_t c;
...
mutex_lock(&m);
to.tv_sec = time(NULL) + TIMEOUT;
to.tv_nsec = 0;
while (cond == FALSE) {
    err = cond_timedwait(&c, &m, &to);
    if (err == ETIME) {
        /* timeout, do something */
        break;
    }
}
mutex_unlock(&m);
```

## *Unblock All Threads*

### `cond_broadcast`*(3T)*

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

int cond_broadcast(cond_t *cvp);
```

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

This function wakes all the threads blocked in `cond_wait`(). Since `cond_broadcast`() causes all threads blocked on the condition to contend again for the mutex lock, use it with care.

For example, use `cond_broadcast`() to allow threads to contend for variable resource amounts when resources are freed, as shown in Code Example 3-10.

*Code Example 3-10* Condition Variable Broadcast

```
mutex_t rsrc_lock;
cond_t rsrc_add;
unsigned int resources;

get_resources(int amount)
{
    mutex_lock(&rsrc_lock);
    while (resources < amount) {
        cond_wait(&rsrc_add, &rsrc_lock);
    }
    resources -= amount;
    mutex_unlock(&rsrc_lock);
}

add_resources(int amount)
{
    mutex_lock(&rsrc_lock);
    resources += amount;
    cond_broadcast(&rsrc_add);
    mutex_unlock(&rsrc_lock);
}
```

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

**Return Values —** `cond_broadcast`() 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.

EFAULT *cvp* points to an illegal address.

Call `cond_broadcast`() 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 `cond_wait`(), which can cause an infinite wait.

## *Destroy Condition Variable State*

### `cond_destroy`*(3T)*

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

int cond_destroy(cond_t *cvp);
```

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

**Return Values —** `cond_destroy`() 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.

EFAULT *cvp* points to an illegal address.

## The Lost Wake-Up Problem

Calling `cond_signal`() or `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 signal or broadcast has been sent but a thread is waiting on the condition variable even though the condition is true. This happens when the thread that calls `cond_signal`() does not hold the mutex locally.

If the thread calls `cond_signal`() when another thread is between the test of the condition and the call to `cond_wait`(), there are no waiting threads and the signal has no effect.

## 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 3-11 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 3-11* 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 3-12 on page 61 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 `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 `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 3-12). When the condition is signaled, the first thread waiting on `less` is awakened. However, before the thread can return from `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 `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 3-12* The Producer/Consumer Problem—the Producer

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

    while (b->occupied >= BSIZE)
        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) */

    cond_signal(&b->more);

    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 code comment a few lines later could better be expressed as an assertion, but it is too complicated to say as a Boolean-valued expression and so is said here 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 when 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 3-13 shows the code for the consumer. Its flow is symmetric with that of the producer.

*Code Example 3-13* The Producer/Consumer Problem—the Consumer

```
char consumer(buffer_t *b) {
    char item;
    mutex_lock(&b->mutex);
    while(b->occupied <= 0)
        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) */

    cond_signal(&b->less);
    mutex_unlock(&b->mutex);

    return(item);
}
```

# *Multiple-Readers, Single-Writer Locks*

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

*Table 3-3* Routines for Readers/Writer Locks

| Routine | Operation | Page |
|---|---|---|
| *rwlock_init(3T)* | *Initialize a Readers/Writer Lock* | *page 64* |
| *rw_rdlock(3T)* | *Acquire a Read Lock* | *page 65* |
| *rw_tryrdlock(3T)* | *Try to Acquire a Read Lock* | *page 65* |
| *rw_wrlock(3T)* | *Acquire a Write Lock* | *page 66* |
| *rw_trywrlock(3T)* | *Try to Acquire a Write Lock* | *page 67* |
| *rw_unlock(3T)* | *Unlock a Readers/Writer Lock* | *page 67* |
| *rwlock_destroy(3T)* | *Destroy Readers/Writer Lock State* | *page 68* |

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.

# 3

## *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.

**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.

Code Example 3-14 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 3-14* 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;
    savings_balance = savings_balance + amount;
    rw_unlock(&account_lock);
}
```

## *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*.

`sema_wait`(3T) and `sema_post`(3T) correspond to Dijkstra's P and V operations. `sema_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 `sema_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 until the count becomes greater than zero.

*Table 3-4* Routines for Semaphores

| Routine | Operation | Page |
|---|---|---|
| *sema_init(3T)* | *Initialize a Semaphore* | *page 72* |
| *sema_post(3T)* | *Increment a Semaphore* | *page 73* |
| *sema_wait(3T)* | *Block on a Semaphore Count* | *page 73* |
| *sema_trywait(3T)* | *Decrement a Semaphore Count* | *page 74* |
| *sema_destroy(3T)* | *Destroy the Semaphore State* | *page 75* |

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.

## *Initialize a Semaphore*

### `sema_init`*(3T)*

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

**Return Values —** `sema_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 *sp* or *arg* points to an illegal address.

## *Increment a Semaphore*

### `sema_post`*(3T)*

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

**Return Values —** `sema_post`() 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 *sp* points to an illegal address.

## *Block on a Semaphore Count*

### `sema_wait`*(3T)*

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

**Return Values —** `sema_wait`() 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 *sp* points to an illegal address.

EINTR The wait was interrupted by a signal or a `fork`().

## *Decrement a Semaphore Count*

### `sema_trywait`*(3T)*

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

**Return Values —** `sema_trywait`() 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 *sp* points to an illegal address.

EBUSY The semaphore pointed to by *sp* has a zero count.

## *Destroy the Semaphore State*

### `sema_destroy`*(3T)*

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

**Return Values —** `sema_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 *sp* points to an illegal address.

## The Producer/Consumer Problem, Using Semaphores

The data structure in Code Example 3-15 is similar to that used for the solution with condition variables; 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 3-15* The Producer/Consumer Problem With Semaphores

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

buffer_t buffer;

sema_init(&buffer.occupied, 0, USYNC_THREAD, 0);
sema_init(&buffer.empty, BSIZE, USYNC_THREAD, 0);
sema_init(&buffer.pmut, 1, USYNC_THREAD, 0);
sema_init(&buffer.cmut, 1, USYNC_THREAD, 0);
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 3-16* The Producer/Consumer Problem—the Producer

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

    sema_wait(&b->pmut);

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

    sema_post(&b->pmut);

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

*Code Example 3-17* The Producer/Consumer Problem—the Consumer

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

    sema_wait(&b->occupied);

    sema_wait(&b->cmut);

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

    sema_post(&b->cmut);

    sema_post(&b->empty);

    return(item);
}
```

# 3

## *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);
```

Code Example 3-18 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`.

*Code Example 3-18* 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.

# 3

# *Comparing Primitives*

The most basic synchronization primitive in Solaris 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 Solaris 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 `cond_wait`() and after changing the state of the variable.

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.

The readers/writer lock is the most complex Solaris threads synchronization mechanism. This means that the readers/writer lock is most efficiently used with a much coarser granularity than is effective with the other synchronization primitives. A readers/writer lock is basically used with a resource whose contents are searched more often than they are changed.