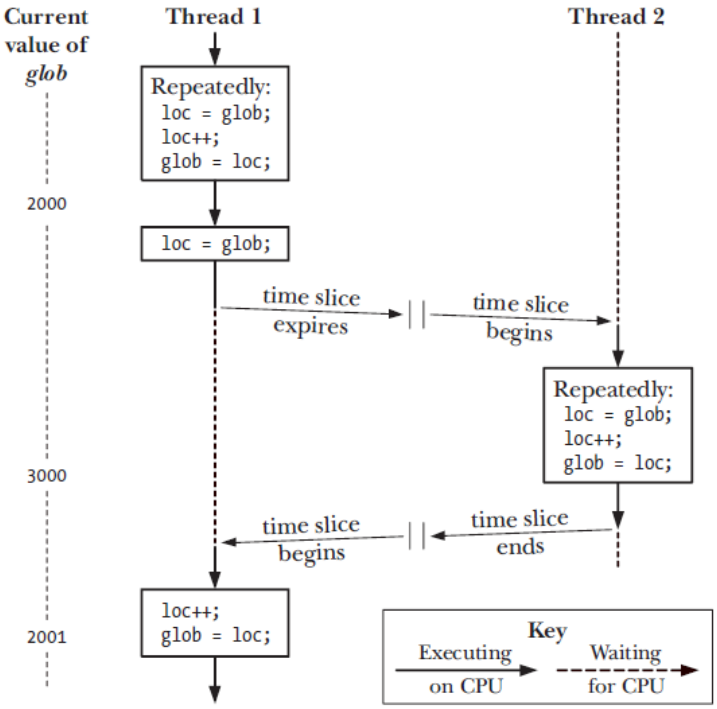
```
/* Two threads update a global shared variable without synchronization*/
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
int max;
volatile int counter = 0; // shared global variable
void * mythread(void *arg)
{   char *letter = arg;
    int i; // stack (private per thread)
    printf("%s: begin [addr of i: %p]\n", letter, &i);
    for (i = 0; i < max; i++) {
       counter = counter + 1; // shared: only one
    }
    printf("%s: done\n", letter);
    return NULL;
}
int main(int argc, char *argv[])
{      if (argc != 2) {
       fprintf(stderr, "usage: main-first <loopcount>\n");
       exit(1);
    }
    max = atoi(argv[1]);
    pthread_t p1, p2;
    printf("main: begin [counter = %d] [%x]\n", counter,(unsigned int) &counter);
    pthread_create(&p1, NULL, mythread, "A");
    pthread_create(&p2, NULL, mythread, "B");
    // join waits for the threads to finish
    pthread_join(p1, NULL);
    pthread_join(p2, NULL);
    printf("main: done\n [counter: %d]\n [should: %d]\n", counter, max*2);
    return 0;
}/* [vishnu@localhost ~]$ cc t1.c -lpthread
[vishnu@localhost ~]$ ./a.out 10
main: begin [counter = 0] [60106c]
B: begin [addr of i: 0x7f7da8bf6f44]
B: done
A: begin [addr of i: 0x7f7da93f7f44]
A: done
main: done
 [counter: 20]
 [should: 20]                                    */
```

Incorrectly incrementing a global variable from two threads: Refer Increment Problem in Synchronization - I

Two threads incrementing a global variable without synchronization



The problem here results from execution sequences such as the following:

1. Thread 1 fetches the current value of glob into its local variable loc. Let's assume that the current value of glob is 2000.

2. The scheduler time slice for thread 1 expires, and thread 2 commences execution.

3. Thread 2 performs multiple loops in which it fetches the current value of glob into its local variable loc, increments loc, and assigns the result to glob. In the first of these loops, the value fetched from glob will be 2000. Let's suppose that by the time the time slice for thread 2 has expired, glob has been increased to 3000.

4. Thread 1 receives another time slice and resumes execution where it left off. Having previously (step 1) copied the value of glob (2000) into its loc, it now increments loc and assigns the result (2001) to glob. At this point, the effect of the increment operations performed by thread 2 is lost.

Mutex Deadlocks

Sometimes, a thread needs to simultaneously access two or more different shared resources, each of which is governed by a separate mutex. When more than one thread is locking the same set of mutexes, deadlock situations can arise. Figure shows an example of a deadlock in which each thread successfully locks one mutex, and then tries to lock the mutex that the other thread has already locked. Both threads will remain blocked indefinitely.

Thread A	**Thread B**
1. pthread_mutex_lock(mutex1); 2. pthread_mutex_lock(mutex2);	1. pthread_mutex_lock(mutex2); 2. pthread_mutex_lock(mutex1);

blocks	blocks

Figure: A deadlock when two threads lock two mutexes

The simplest way to avoid such deadlocks is to define a mutex hierarchy. When threads can lock the same set of mutexes, they should always lock them in the same order. For example, in the scenario in Figure, the deadlock could be avoided if the two threads always lock the mutexes in the order mutex1 followed by mutex2. Sometimes, there is a logically obvious hierarchy of mutexes. However, even if there isn't, it may be possible to devise an arbitrary hierarchical order that all threads should follow.

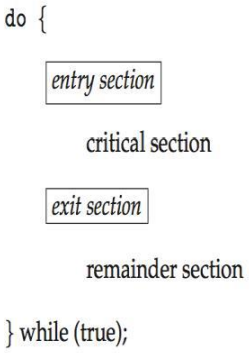
Critical section problem

Each process must ask permission to enter critical section in **entry section**, may follow critical section with **exit section**, then **remainder section**

General structure of process ***Pi***

Algorithm for Process Pi

```
do {
        while (turn == j);
                critical section
                        turn = j;
                remainder section
        } while (true);
```



Solution to Critical-Section Problem

1. **Mutual Exclusion** - If process ***Pi*** is executing in its critical section, then no other processes can be executing in their critical sections
2. **Progress** - If no process is executing in its critical section and there exist some processes that wish to enter their critical section, then the selection of the processes that will enter the critical section next cannot be postponed indefinitely
3. **Bounded Waiting** - A bound must exist on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted

- Assume that each process executes at a nonzero speed
- No assumption concerning **relative speed** of the ***n*** processes

Semaphores

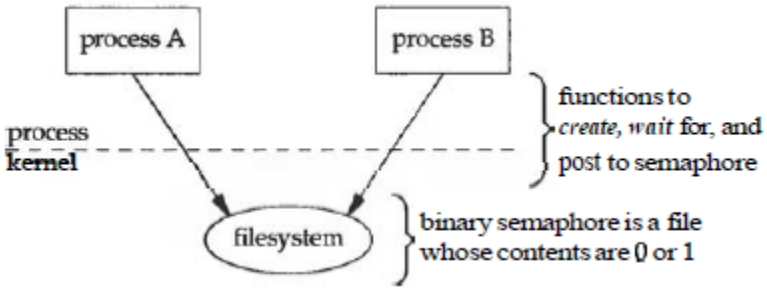
A semaphore is a primitive used to provide synchronization between various processes or between the various threads in a given process. We look at three types of semaphores in this section.

- POSIX named semaphores are identified by POSIX **IPC** names and can be used to synchronize processes or threads. By calling sem_open() with the same name, unrelated processes can access the same semaphore.
- POSIX memory-based semaphores are stored in shared memory and can be used to synchronize processes or threads. This type of semaphore doesn't have a name; instead, it resides at an agreed-upon location in memory. Unnamed semaphores can be shared between processes or between a group of threads. When shared between processes, the semaphore must reside in a region of (System V, POSIX, or mmap()) shared memory. When shared between threads, the semaphore may reside in an area of memory shared by the threads (e.g., on the heap or in a global variable).
- System V semaphores are maintained in the kernel and can be used to synchronize processes or threads.

For now, we concern ourselves with synchronization between different processes. We first consider a *binary semaphore:* a semaphore that can assume only the values 0 or 1.

We show that the semaphore is maintained by the kernel (which is true for System V semaphores) and that its value can be 0 or 1.

POSIX semaphores need not be maintained in the kernel. Also, POSIX semaphores are identified by names that might correspond to pathnames in the files system. Therefore, Figure is a more realistic picture of what is termed a *POSIX named semaphore.*



A Posix named binary semaphore being used by two processes

We must make one qualification with regard to Figure: although POSIX named semaphores are identified by names that might correspond to pathnames in the filesystem, nothing requires that they actually be stored in a file in the filesystem.

Three operations that a process can perform on a semaphore:

1. ***Create*** a semaphore. This also requires the caller to specify the initial value, which for a binary semaphore is often 1, but can be 0.
2. ***Wait*** for a semaphore. This tests the value of the semaphore, waits (blocks) if the value is less than or equal to 0, and then decrements the semaphore value once it is greater than 0. This can be summarized by the pseudocode

```
while (semaphore-value <= 0)
        ; /* wait; i-e., block the thread or process */
semaphore-value--;
        /* we have the semaphore */
```

The fundamental requirement here is that the test of the value in the while statement, and its subsequent decrement (if its value was greater than 0), must be done as an *atomic* operation with respect to other threads or processes accessing this semaphore.

3. ***Post*** to a semaphore. This increments the value of the semaphore and can be summarized by the pseudocode

If any processes are blocked, waiting for this semaphore's value to be greater than 0, one of those processes can now be awoken. As with the wait code just shown, this post operation must also be atomic with regard to other processes accessing the semaphore. Obviously, the actual semaphore code has more details than we show in the pseudocode for the wait and post operations: namely how to queue all the processes that are waiting for a given semaphore and then how to wake up one (of the possibly many processes) that is waiting for a given semaphore to be posted to. Fortunately, these details are handled by the implementation. Notice that the pseudocode shown does not assume a binary semaphore with the values 0 and 1. The code works with semaphores that are initialized to any nonnegative value. These are called ***counting semaphores.*** These are normally initialized to some value *N,* which indicates the number of resources (say buffers) available. We show examples of both binary semaphores and counting semaphores throughout the handout

We often differentiate between a binary semaphore and a counting semaphore, and we do so for our own edification. No difference exists between the two in the system code that implements a semaphore.

A binary semaphore can be used for mutual exclusion, just like a mutex.

initialize mutex;
`pthread_mutex_lock(&mutex);`
critical region
`pthread_mutex_unlock(&mutex);`
	initialize to 1;semaphore
	`Sem_wait(&sem);`
	critical region
	`sem_post (&sem);`

Comparison of mutex and semaphore to solve mutual exclusion problem.

We initialize the semaphore to 1. The call to **`sem_wait`** waits for the value to be greater than 0 and then decrements the value. The call to **`sem_post`** increments the value (from 0 to 1) and wakes up any threads blocked in a call to **`sem_wait`** for this semaphore.

Although semaphores can be used like a mutex, semaphores have a feature not provided by mutexes: a mutex must always be unlocked by the thread that locked the mutex, while a semaphore post need not be performed by the same thread that did the semaphore wait.

We can show an example of this feature using two binary semaphores and a **simplified version of the producer-consumer problem**: a producer that places an item into a shared buffer and a consumer that removes the item. For simplicity, assume that the buffer holds one item.

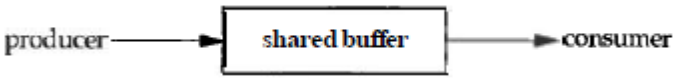


Figure: Simple producer-consumer problem with a shared buffer.

Shows the pseudocode for the producer and consumer.

Producer	**Consumer**

initialize semaphore `get` to 0;
initialize semaphore `put` to 1;
`sem_wait (&put) ;`
put data into buffer
`sem_post (&get) ;`
	`for(;;)(`
	`sem_wait(&get);`
	process data in buffer
	`sem_post (&put) ;`

Figure: Pseudocode for simple producer+mnsumer.

The semaphore **`put`** controls whether the producer can place an item into the shared buffer, and the semaphore **`get`** controls whether the consumer can remove an item from the shared buffer. The steps that occur over time are as follows:

1. The producer initializes the buffer and the two semaphores.
2. Assume that the consumer then runs. It blocks in its call to **`sem_wait`** because the value of **`get`** is 0.
3. Sometime later, the producer starts. When it calls **`sem_wait,`** the value of **`put`** is decremented from 1 to 0, and the producer places an item into the buffer. It then calls **`sem_post`** to increment the value of **`get`** from 0 to 1. Since a thread is blocked on this semaphore (the consumer), waiting for its value to become positive, that thread is marked as ready-to-run. But assume that the producer continues to run. The producer then blocks in its call to **`sem_wait`** at the top of the **`for`** loop, because the value of **`put`** is 0. The producer must wait until the consumer empties the buffer.
4. The consumer returns from its call to **`sem_wait,`** which decrements the value of the **`get`** semaphore from 1 to 0. It processes the data in the buffer, and calls **`sem_post,`** which increments the value of **`put`** from 0 to 1. Since a thread is blocked on this semaphore (the producer), waiting for its value to become positive, that thread is marked as ready-to-run. But assume that the consumer continues to run. The consumer then blocks in its call to **`sem_wait,`** at the top of the **`for`** loop, because the value of **`get`** is 0.
5. The producer returns from its call to sem_wait, places data into the buffer, and this scenario just continues.

We assumed that each time **`sem_post`** was called, even though a process was waiting and was then marked as ready-to-run, the caller continued. Whether the caller continues or whether the thread that just became ready runs does not matter (you should assume the other scenario and convince yourself of this fact).

Named Semaphores

To work with a named semaphore, we employ the following functions:

- The sem_open() function opens or creates a semaphore, initializes the semaphore if it is created by the call, and returns a handle for use in later calls.

```
#include <semaphore.h>
sem_t *sem_open(const char *name, int oflag, ... /* mode_t mode, unsigned int value */ );
Returns pointer to semaphore on success, or SEM_FAILED on error
```

- The sem_post(sem) and sem_wait(sem) functions respectively increment and decrement a semaphore's value.

```
int sem_wait(sem_t *sem);
Returns 0 on success, or –1 on error
int sem_trywait(sem_t *sem);
```

```
Returns 0 on success, or –1 on error
int sem_post(sem_t *sem);
Returns 0 on success, or –1 on error
```

- The sem_getvalue() function retrieves a semaphore's current value.

```
int sem_getvalue(sem_t *sem, int *sval);
Returns 0 on success, or –1 on error
```

- The sem_close() function removes the calling process's association with a semaphore that it previously opened.

```
int sem_close(sem_t *sem);
Returns 0 on success, or –1 on error
```

- The sem_unlink() function removes a semaphore name and marks the semaphore for deletion when all processes have closed it.

```
int sem_unlink(const char *name);
Returns 0 on success, or –1 on error
```

Unnamed Semaphores

Unnamed semaphores (also known as memory-based semaphores) are variables of type sem_t that are stored in memory allocated by the application. The semaphore is made available to the processes or threads that use it by placing it in an area of memory that they share. Operations on unnamed semaphores use the same functions (sem_wait(), sem_post(), sem_getvalue(), and so on) that are used to operate on named semaphores.
In addition, two further functions are required:

- The sem_init() function initializes a semaphore and informs the system of whether the semaphore will be shared between processes or between the threads of a single process.

```
int sem_init(sem_t *sem, int pshared, unsigned int value);
Returns 0 on success, or –1 on error
```

- The sem_destroy(sem) function destroys a semaphore. These functions should not be used with named semaphores.

```
int sem_destroy(sem_t *sem);
Returns 0 on success, or –1 on error
```

The POSIX semaphore interface is simpler than the System V semaphore interface. Semaphores are allocated and operated on individually, and the wait and post operations adjust a semaphore's value by one.
POSIX semaphores have a number of advantages over System V semaphores, but they are somewhat less portable. For synchronization within multithreaded applications, mutexes are generally preferred over semaphores.

The Bounded-Buffer Problem or Producer-Consumer Problem

producer-consurner problem solutions in which multple producer threads fill an array that is processed by one consumer thread.
1. In our first solution, the consumer started only after the producers were finished, and we were able to solve this synchronization problem using a single mutex (to synchronize the producers). **See code at page number: 12**
2. In our next solution (the consumer started before the producers were finished, and this required a mutex (to synchronize the producers) along with a condition variable and its mutex (to synchronize the consumer with the producers). **See code at page number: 13**
We now extend the producer-consumer problem by using the shared buffer as a circular buffer: after the producer fills the final entry (buff [NBUFF-1]), it goes back and fills the first entry (buff [0])**,** and the consumer does the same. This adds another synchronization problem in that the producer must not get ahead of the consumer. We still assume that the producer and consumer are threads, but they could also be processes, assuming that some way existed to share the buffer between the processes (e.g., shared memory). **See code at page number: 12**
Three conditions must be maintained by the code when the shared buffer is considered as a circular buffer:
1. The consumer cannot try to remove an item from the buffer when the buffer is empty.
2. The producer cannot try to place an item into the buffer when the buffer is full.
3. Shared variables may describe the current state of the buffer (indexes, counts, linked list pointers, etc.), so all buffer manipulations by the producer and consumer must be protected to avoid any race conditions.
Our solution using semaphores demonstrates three different types of semaphores:
1. A binary semaphore named mutex protects the critical regions: inserting a data item into the buffer (for the producer) and removing a data item from the buffer (for the consumer). A binary semaphore that is used as a mutex is initialized to 1. (Obviously we could use a real mutex for this, instead of a binary semaphore.)
2. A counting semaphore named nempty counts the number of empty slots in the buffer. This semaphore is initialized to the number of slots in the buffer (NBUFF).
3. A counting semaphore named nstored counts the number of filled slots in the buffer. This semaphore is initialized to 0, since the buffer is initially empty.

The Readers-Writers Problem

In the readers-writers problem there are some processes (termed readers) who only read the shared data, and never change it, and there are other processes (termed writers) who may change the data in addition to or instead of reading it. There is no limit to how many readers can access the data simultaneously, but when a writer accesses the data, it needs exclusive access.

- There are several variations to the readers-writers problem, most centered around relative priorities of readers versus writers.
    - The *first* readers-writers problem gives priority to readers. In this problem, if a reader wants access to the data, and there is not already a writer accessing it, then access is granted to the reader. A solution to this problem can lead to starvation of the writers, as there could always be more readers coming along to access the data. (A steady stream of readers will jump ahead of waiting writers as long as there is currently already another reader accessing the data, because the writer is forced to wait until the data is idle, which may never happen if there are enough readers.)

```
do {
   wait(rw_mutex);
      . . .
   /* writing is performed */
      . . .
   signal(rw_mutex);
} while (true);
```

Figure 5.11 The structure of a writer process.

```
do {
   wait(mutex);
   read_count++;
   if (read_count == 1)
      wait(rw_mutex);
   signal(mutex);
      . . .
   /* reading is performed */
      . . .
   wait(mutex);
   read_count--;
   if (read_count == 0)
      signal(rw_mutex);
   signal(mutex);
} while (true);
```

Figure 5.12 The structure of a reader process.

- The *second* readers-writers problem gives priority to the writers. In this problem, when a writer wants access to the data it jumps to the head of the queue - All waiting readers are blocked, and the writer gets access to the data as soon as it becomes available. In this solution the readers may be starved by a steady stream of writers.
- The following code is an example of the first readers-writers problem, and involves an important counter and two binary semaphores:
  - readcount is used by the reader processes, to count the number of readers currently accessing the data.
  - mutex is a semaphore used only by the readers for controlled access to readcount.
  - rw_mutex is a semaphore used to block and release the writers. The first reader to access the data will set this lock and the last reader to exit will release it; The remaining readers do not touch rw_mutex.
  - Note that the first reader to come along will block on rw_mutex if there is currently a writer accessing the data, and that all following readers will only block on mutex for their turn to increment readcount.

Condition Variables

A mutex prevents multiple threads from accessing a shared variable at the same time. A condition variable allows one thread to inform other threads about changes in the state of a shared variable (or other shared resource) and allows the other threads to wait (block) for such notification.

```
static pthread_mutex_t mtx = PTHREAD_MUTEX_INITIALIZER;
static int avail = 0;
```

A simple example that doesn't use condition variables serves to demonstrate why they are useful. Suppose that we have a number of threads that produce some "result units" that are consumed by the main thread, and that we use a mutex-protected variable, avail, to represent the number of produced units awaiting consumption:

```
/* prod_no_condvar.c -- A simple POSIX threads producer-consumer example that doesn't use a
condition variable. See also prod_condvar.c. */
#include <time.h>
#include <pthread.h>
static pthread_mutex_t mtx = PTHREAD_MUTEX_INITIALIZER;
static int avail = 0;
static void *
threadFunc(void *arg)
{   int cnt = atoi((char *) arg);
    int s, j;
    for (j = 0; j < cnt; j++) {
        sleep(1);
        /* Code to produce a unit omitted */
        s = pthread_mutex_lock(&mtx);
        if (s != 0)
            quit("pthread_mutex_lock",s);
        avail++;        /* Let consumer know another unit is available */
        s = pthread_mutex_unlock(&mtx);
        if (s != 0)
            quit("pthread_mutex_unlock",s);
    }
    return NULL;
}
void quit (char *message, int exit_status) {
perror(message);
exit(exit_status);
}
int main(int argc, char *argv[])
{   pthread_t tid;
    typedef enum { FALSE, TRUE } Boolean;
    int s, j;
    int totRequired;            /* Total number of units that all threads will produce */
    int numConsumed;            /* Total units so far consumed */
    Boolean done;
    time_t t;
    t = time(NULL);
    /* Create all threads */
    totRequired = 0;
    for (j = 1; j < argc; j++) {
        totRequired += atoi(argv[j]);
        s = pthread_create(&tid, NULL, threadFunc, argv[j]);
        if (s != 0)
            quit("pthread_create",s);
    } /* Use a polling loop to check for available units */
```

```
    numConsumed = 0;
    done = FALSE;
    for (;;) {
        s = pthread_mutex_lock(&mtx);
        if (s != 0)
            quit("pthread_mutex_lock",s);
        while (avail > 0) {             /* Consume all available units */
            /* Do something with produced unit */
            numConsumed ++;
            avail--;
            printf("T=%ld: numConsumed=%d\n", (long) (time(NULL) - t), numConsumed);
            done = numConsumed >= totRequired;
        }
        s = pthread_mutex_unlock(&mtx);
        if (s != 0)
            quit("pthread_mutex_unlock",s);
        if (done)
            break;
        /* Perhaps do other work here that does not require mutex lock */
    }
    exit(0);
}/* [vishnu@localhost ~]$ ./a.out 4 5 1
T=1: numConsumed=1
T=1: numConsumed=2
T=1: numConsumed=3
T=2: numConsumed=4
T=2: numConsumed=5
T=3: numConsumed=6
T=3: numConsumed=7
T=4: numConsumed=8
T=4: numConsumed=9
T=5: numConsumed=10                                 */
```

The above code works, but it wastes CPU time, because the main thread continually loops, checking the state of the variable avail. A condition variable remedies this problem. It allows a thread to sleep (wait) until another thread notifies (signals) it that it must do something (i.e., that some “condition” has arisen that the sleeper must now respond to).

A condition variable is always used in conjunction with a mutex. The mutex provides mutual exclusion for accessing the shared variable, while the condition variable is used to signal changes in the variable’s state.

Statically Allocated Condition Variables

As with mutexes, condition variables can be allocated statically or dynamically.

A condition variable has the type pthread_cond_t. As with a mutex, a condition variable must be initialized before use. For a statically allocated condition variable, this is done by assigning it the value PTHREAD_COND_INITIALIZER, as in the following example:

```
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
```

Signaling and Waiting on Condition Variables

The principal condition variable operations are signal and wait. The signal operation is a notification to one or more waiting threads that a shared variable’s state has changed. The wait operation is the means of blocking until such a notification is received.

The pthread_cond_signal() and pthread_cond_broadcast() functions both signal the condition variable specified by cond. The pthread_cond_wait() function blocks a thread until the condition variable cond is signaled.

```
#include <pthread.h>
int pthread_cond_signal(pthread_cond_t *cond);
int pthread_cond_broadcast(pthread_cond_t *cond);
int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);
```

All return 0 on success, or a positive error number on error

The difference between pthread_cond_signal() and pthread_cond_broadcast() lies in what happens if multiple threads are blocked in pthread_cond_wait(). With pthread_cond_signal(), we are simply guaranteed that at least one of the blocked threads is woken up; with pthread_cond_broadcast(), all blocked threads are woken up.

Using pthread_cond_broadcast() always yields correct results (since all threads should be programmed to handle redundant and spurious wake-ups), but pthread_cond_signal() can be more efficient. However, pthread_cond_signal() should be used only if just one of the waiting threads needs to be woken up to handle the change in state of the shared variable, and it doesn’t matter which one of the waiting threads is woken up. This scenario typically applies when all of the waiting threads are designed to perform the exactly same task. Given these assumptions, pthread_cond_signal() can be more efficient than pthread_cond_broadcast(), because it avoids the following possibility:

1. All waiting threads are awoken.

2. One thread is scheduled first. This thread checks the state of the shared variable(s) (under protection of the associated mutex) and sees that there is work to be done. The thread performs the required work, changes the state of the shared variable(s) to indicate that the work has been done, and unlocks the associated mutex.

3. Each of the remaining threads in turn locks the mutex and tests the state of the shared variable. However, because of the change made by the first thread, these threads see that there is no work to be done, and so unlock the mutex and go back to sleep (i.e., call pthread_cond_wait() once more). By contrast, pthread_cond_broadcast() handles the case where the waiting threads are designed to perform different tasks (in which case they probably have different predicates associated with the condition variable).

A condition variable holds no state information. It is simply a mechanism for communicating information about the application’s state. If no thread is waiting on the condition variable at the time that it is signaled, then the signal is lost. A thread that later waits on the condition variable will unblock only when the variable is signaled once more.

pthread_cond_wait() Steps

We noted earlier that a condition variable always has an associated mutex. Both of these objects are passed as arguments to pthread_cond_wait(), which performs the following steps:

- unlock the mutex specified by mutex;
- block the calling thread until another thread signals the condition variable cond; and
- relock mutex.

Using a condition variable in the producer-consumer example

```
/* prod_condvar.c -- A simple POSIX threads producer-consumer example using a condition
variable. */
#include <time.h>
#include <pthread.h>
static pthread_mutex_t mtx = PTHREAD_MUTEX_INITIALIZER;
static pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
static int avail = 0;
static void *
threadFunc(void *arg)
{   int cnt = atoi((char *) arg);
    int s, j;
    for (j = 0; j < cnt; j++) {
        sleep(1);
        /* Code to produce a unit omitted */
        s = pthread_mutex_lock(&mtx);
        if (s != 0)
            quit("pthread_mutex_lock",s);
        avail++;        /* Let consumer know another unit is available */
        s = pthread_mutex_unlock(&mtx);
        if (s != 0)
            quit("pthread_mutex_unlock",s);
        s = pthread_cond_signal(&cond);         /* Wake sleeping consumer */
        if (s != 0)
            quit("pthread_cond_signal",s);
    }
    return NULL;
}
void quit (char *message, int exit_status) {
perror(message);
exit(exit_status);
}
int main(int argc, char *argv[])
{   pthread_t tid;
    typedef enum { FALSE, TRUE } Boolean;
    int s, j;
    int totRequired;            /* Total number of units that all threads will produce */
    int numConsumed;            /* Total units so far consumed */
    Boolean done;
    time_t t;
    t = time(NULL);
    /* Create all threads */
    totRequired = 0;
    for (j = 1; j < argc; j++) {
        totRequired += atoi(argv[j]);
        s = pthread_create(&tid, NULL, threadFunc, argv[j]);
        if (s != 0)
            quit("pthread_create",s);
    }
    /* Loop to consume available units */
    numConsumed = 0;
    done = FALSE;
    for (;;) {
        s = pthread_mutex_lock(&mtx);
        if (s != 0)
            quit("pthread_mutex_lock",s);
        while (avail == 0) {            /* Wait for something to consume */
            s = pthread_cond_wait(&cond, &mtx);
            if (s != 0)
                quit("pthread_cond_wait",s);
        }
        /* At this po,int, 'mtx' is locked... */
        while (avail > 0) {             /* Consume all available units */
            /* Do something with produced unit */
            numConsumed ++;
            avail--;
            printf("T=%ld: numConsumed=%d\n", (long) (time(NULL) - t), numConsumed);
            done = numConsumed >= totRequired;
        }
        s = pthread_mutex_unlock(&mtx);
```

```
        if (s != 0)
            quit("pthread_mutex_unlock",s);
        if (done)
            break;
        /* Perhaps do other work here that does not require mutex lock */
    }
    exit(0);
}/* [vishnu@localhost ~]$ cc prod_condvar.c -lpthread
[vishnu@localhost ~]$ ./a.out 4 5
T=1: numConsumed=1
T=1: numConsumed=2
T=2: numConsumed=3
T=2: numConsumed=4
T=3: numConsumed=5
T=3: numConsumed=6
T=4: numConsumed=7
T=4: numConsumed=8
T=5: numConsumed=9                                          */
```

In the above code, both accesses to the shared variable must be mutex-protected for the reasons that we explained earlier. In other words, there is a natural association of a mutex with a condition variable:

1. The thread locks the mutex in preparation for checking the state of the shared variable.
2. The state of the shared variable is checked.
3. If the shared variable is not in the desired state, then the thread must unlock the mutex (so that other threads can access the shared variable) before it goes to sleep on the condition variable.
4. When the thread is reawakened because the condition variable has been signaled, the mutex must once more be locked, since, typically, the thread then immediately accesses the shared variable.

The pthread_cond_wait() function automatically performs the mutex unlocking and locking required in the last two of these steps. In the third step, releasing the mutex and blocking on the condition variable are performed atomically. In other words, it is not possible for some other thread to acquire the mutex and signal the condition variable before the thread calling pthread_cond_wait() has blocked on the condition variable.

In the producer code shown earlier, we called pthread_mutex_unlock(), and then called pthread_cond_signal(); that is, we first unlocked the mutex associated with the shared variable, and then signaled the corresponding condition variable.

Testing a Condition Variable's Predicate

Each condition variable has an associated predicate involving one or more shared variables. For example, in the code segment in the preceding section, the predicate associated with cond is (avail == 0). This code segment demonstrates a general design principle: a pthread_cond_wait() call must be governed by a while loop rather than an if statement. This is so because, on return from pthread_cond_wait(), there are no guarantees about the state of the predicate; therefore, we should immediately recheck the predicate and resume sleeping if it is not in the desired state.

Semaphores	*Condition Variables*
Can be used anywhere in a program, but should not be used in a monitor	**Can only be used in monitors**
Wait() does not always block the caller (*i.e.*, when the semaphore counter is greater than zero).	**Wait() always blocks the caller.**
Signal() either releases a blocked thread, if there is one, or increases the semaphore counter.	**Signal() either releases a blocked thread, if there is one, or the signal is lost as if it never happens.**
If Signal() releases a blocked thread, the caller and the released thread *both* continue.	**If Signal() releases a blocked thread, the caller yields the monitor (Hoare type) or continues (Mesa Type). Only one of the caller or the released thread can continue, but not both.**

```
/* prodcons-mutex.c - Producer Consumer problem using mutex and pthreads */
#include    <stdio.h>
#include    <unistd.h>
#include    <fcntl.h>
#include    <pthread.h>
#include    <sys/types.h>
#define     MAXNITEMS         1000000
#define     MAXNTHREADS       100
int         nitems;                 /* read-only by producer and consumer */
struct {
  pthread_mutex_t mutex;
  int buff[MAXNITEMS];
  int nput;
  int nval;
} shared ={PTHREAD_MUTEX_INITIALIZER};
void  *produce(void *), *consume(void *);
int main(int argc, char **argv)
{
      shared.nput=0;
      shared.nval=0;
      int               i, nthreads, count[MAXNTHREADS];
      pthread_t   tid_produce[MAXNTHREADS], tid_consume;
      if (argc != 3)
      {
            printf("usage: prodcons1 <#items> <#threads>");
            exit(1);
      }
      nitems = atoi(argv[1]);
      nthreads = atoi(argv[2]);
      pthread_setconcurrency(nthreads);
            /* 4start all the producer threads */
      for (i = 0; i < nthreads; i++) {
            count[i] = 0;
            pthread_create(&tid_produce[i], NULL, produce, &count[i]);
      }
            /* 4wait for all the producer threads */
      for (i = 0; i < nthreads; i++) {
            pthread_join(tid_produce[i], NULL);
            printf("count[%d] = %d\n", i, count[i]);
      }
            /* 4start, then wait for the consumer thread */
      pthread_create(&tid_consume, NULL, consume, NULL);
      pthread_join(tid_consume, NULL);
      exit(0);
}
/* end main */
/* include producer */
void * produce(void *arg)
{
pthread_t tid;
int i=*((int *) arg);
      for ( ; ; ) {
            pthread_mutex_lock(&shared.mutex);
            tid=pthread_self();
            printf("threadid=%u\n", (unsigned int) tid);
            if (shared.nput >= nitems) {
                  pthread_mutex_unlock(&shared.mutex);
                  return(NULL);           /* array is full, we're done */
            }
            shared.buff[shared.nput] = shared.nval;
            printf("buff[%d] = %d\n", shared.nput, shared.buff[shared.nput]);
            shared.nput++;
            shared.nval++;
            *((int *) arg) += 1;
            pthread_mutex_unlock(&shared.mutex);
            printf("shared.nput=%d, shared.nval=%d,count[%u] =
```

```
            %d\n",shared.nput,shared.nval, i, *((int *) arg));
      }
}
void * consume(void *arg)
{
      int   i;
      for (i = 0; i <nitems; i++) {
            if (shared.buff[i]! = i)
                  printf("buff[%d] = %d\n", i, shared.buff[i]);
      }
      return(NULL);
}
/* end producer */
/*
[vishnu@mannava mutex]$ ./a.out 4 5
threadid=3078474608
buff[0] = 0
shared.nput=1, shared.nval=1,count[0] = 1
threadid=3078474608
buff[1] = 1
shared.nput=2, shared.nval=2,count[0] = 2
threadid=3078474608
buff[2] = 2
shared.nput=3, shared.nval=3,count[0] = 3
threadid=3078474608
buff[3] = 3
shared.nput=4, shared.nval=4,count[0] = 4
threadid=3078474608
count[0] = 4
threadid=3067984752
threadid=3057494896
threadid=3047005040
count[1] = 0
count[2] = 0
count[3] = 0
threadid=3036515184
count[4] = 0
*/
```

/* prodcons-semaphores.c Producer Consumer problem using semaphores and pthreads */

```
#include    <stdio.h>
#include    <unistd.h>
#include    <fcntl.h>
#include    <pthread.h>
#include    <semaphore.h>
#include    <sys/types.h>
#define     NBUFF       10
#define     SEM_MUTEX   "mutex"           /* these are args to px_ipc_name() */
#define     SEM_NEMPTY  "nempty"
#define     SEM_NSTORED "nstored"
int nitems; /* read-only by producer and consumer */
struct {    /* data shared by producer and consumer */
  int buff[NBUFF];
  sem_t     *mutex;
  sem_t     *nempty;
  sem_t     *nstored;
} shared;
void *produce(void *), *consume(void *);
int main(int argc, char **argv)
{
      pthread_t   tid_produce, tid_consume;
      if (argc != 2)
      {
            printf("usage: prodcons1 <#items>");
            exit(1);
      }
      nitems = atoi(argv[1]);
```

```
        /* 4create three semaphores */
    shared.mutex = sem_open(SEM_MUTEX, O_CREAT | O_EXCL, 0644, 1);
    shared.nempty = sem_open(SEM_NEMPTY, O_CREAT | O_EXCL, 0644, NBUFF);
    shared.nstored = sem_open(SEM_NSTORED, O_CREAT | O_EXCL, 0644, 0);
        /* 4create one producer thread and one consumer thread */
    pthread_setconcurrency(2);
    pthread_create(&tid_produce, NULL, produce, NULL);
    pthread_create(&tid_consume, NULL, consume, NULL);
        /* 4wait for the two threads */
    pthread_join(tid_produce, NULL);
    pthread_join(tid_consume, NULL);
        /* 4remove the semaphores */
    sem_unlink(SEM_MUTEX);
    sem_unlink(SEM_NEMPTY);
    sem_unlink(SEM_NSTORED);
    exit(0);
}
/* end main */
/* include prodcons */
void *produce(void *arg)
{
    int         i;
    for (i = 0; i < nitems; i++) {
        sem_wait(shared.nempty);      /* wait for at least 1 empty slot */
        sem_wait(shared.mutex);
        shared.buff[i % NBUFF] = i;   /* store i into circular buffer */
        sem_post(shared.mutex);
        sem_post(shared.nstored);     /* 1 more stored item */
    }
    return(NULL);
}
void *consume(void *arg)
{
    int         i;
    for (i = 0; i < nitems; i++) {
        sem_wait(shared.nstored);           /* wait for at least 1 stored item */
        sem_wait(shared.mutex);
        if (shared.buff[i % NBUFF] == i)
            printf("buff[%d] = %d\n", i, shared.buff[i % NBUFF]);
        sem_post(shared.mutex);
        sem_post(shared.nempty);            /* 1 more empty slot */
    }
    return(NULL);
}
/* end prodcons */
/*[vishnu@mannava pxsem]$ ./a.out 5
buff[0] = 0
buff[1] = 1
buff[2] = 2
buff[3] = 3
buff[4] = 4
*/
```

```
/* Implementation of Producer consumer problem using condition
variables */
/* include globals */
#include    <stdio.h>
#include    <unistd.h>
#include    <fcntl.h>
#include    <pthread.h>
#include    <sys/types.h>
#define     MAXNITEMS           1000000
#define     MAXNTHREADS         100
            /* globals shared by threads */
int         nitems;                         /* read-only by producer and consumer
*/
int         buff[MAXNITEMS];
struct {
  pthread_mutex_t   mutex;
  int                   nput;  /* next index to store */
  int                   nval;  /* next value to store */
} put = { PTHREAD_MUTEX_INITIALIZER };
struct {
  pthread_mutex_t   mutex;
  pthread_cond_t    cond;
  int                   nready;       /* number ready for consumer */
} nready = { PTHREAD_MUTEX_INITIALIZER, PTHREAD_COND_INITIALIZER };
/* end globals */
```

```
void   *produce(void *), *consume(void *);
/* include main */
int main(int argc, char **argv)
{
	int			i, nthreads, count[MAXNTHREADS];
	pthread_t	tid_produce[MAXNTHREADS], tid_consume;
	if (argc != 3)
	{
		printf("usage: prodcons6 <#items> <#threads>");
	}
	nitems = atoi(argv[1]);
	nthreads = atoi(argv[2]);
	pthread_setconcurrency(nthreads + 1);
		/* 4create all producers and one consumer */
	for (i = 0; i < nthreads; i++) {
		count[i] = 0;
		pthread_create(&tid_produce[i], NULL, produce, &count[i]);
	}
	pthread_create(&tid_consume, NULL, consume, NULL);
		/* wait for all producers and the consumer */
	for (i = 0; i < nthreads; i++) {
		pthread_join(tid_produce[i], NULL);
		printf("count[%d] = %d\n", i, count[i]);
	}
	pthread_join(tid_consume, NULL);
	exit(0);
}
/* end main */
/* include prodcons */
void *produce(void *arg)
{
	for ( ; ; ) {
		pthread_mutex_lock(&put.mutex);
		if (put.nput >= nitems) {
			pthread_mutex_unlock(&put.mutex);
			return(NULL);		/* array is full, we're done */
		}
		buff[put.nput] = put.nval;
		put.nput++;
		put.nval++;
		pthread_mutex_unlock(&put.mutex);
		pthread_mutex_lock(&nready.mutex);
		if (nready.nready == 0)
			pthread_cond_signal(&nready.cond);
		nready.nready++;
		pthread_mutex_unlock(&nready.mutex);
		*((int *) arg) += 1;
	}
}
void *consume(void *arg)
{
	int			i;
	for (i = 0; i < nitems; i++) {
		pthread_mutex_lock(&nready.mutex);
		while (nready.nready == 0)
			pthread_cond_wait(&nready.cond, &nready.mutex);
		nready.nready--;
		pthread_mutex_unlock(&nready.mutex);

		if (buff[i] == i)
			printf("buff[%d] = %d\n", i, buff[i]);
	}
	return(NULL);
}/* end prodcons */
/*[vishnu@mannava mutex]$ ./a.out 5 4
count[0] = 4
count[1] = 1
count[2] = 0
buff[0] = 0
buff[1] = 1
buff[2] = 2
buff[3] = 3
buff[4] = 4
count[3] = 0
*/
```