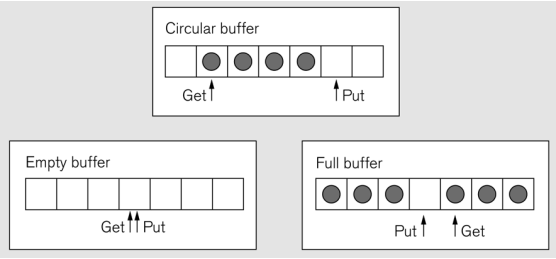
1. Synchronization of behavior between threads using *condition variables.* A condition variable allows threads to wait until some condition becomes true, at which point one of the waiting threads is nondeterministically chosen to stop waiting. Bounded Buffer example:



buffer - array of items treated circularly

Put - index of next empty slot

Get - index of first full (front) slot

a) When would a Producer thread wanting to insert into the bounded buffer need to wait?

b) When would a Consumer thread wanting to remove from the bounded buffer need to wait?

c) When would we want to signal a waiting Producer to "wakeup"?

d) When would we want to signal a waiting Consumer to "wakeup"?

Figure 6.3
Bounded buffer example using condition variables nonempty and nonfull.

```
1  pthread_mutex_t lock=PTHREAD_MUTEX_INITIALIZER;
2  pthread_cond_t nonempty=PTHREAD_COND_INITIALIZER;
3  pthread_cond_t nonfull=PTHREAD_COND_INITIALIZER;
4  Item buffer[SIZE];
5  int put=0;                          // Buff index for next insert
6  int get=0;                          // Buff index for next remove
7
8  void insert(Item x)                 // Producer thread
9  {
10   pthread_mutex_lock(&lock);
11   while((put+1)%SIZE == get)        // While buffer is full
12   {
13     pthread_cond_wait(&nonfull, &lock);
14   }
15   buffer[put]=x;
16   put=(put+1)%SIZE;
17   pthread_cond_signal(&nonempty);
18   pthread_mutex_unlock(&lock);
19 }
20
21 Item remove()                       // Consumer thread
22 {
23   Item x;
24   pthread_mutex_lock(&lock);
25   while(put==get)                   // While buffer is empty
26   {
27     pthread_cond_wait(&nonempty, &lock);
28   }
29   x=buffer[get];
30   get=(get+1)%SIZE;
31   pthread_cond_signal(&nonfull);
32   pthread_mutex_unlock(&lock);
33   return x;
34 }
```

Notes on code:

- single mutex lock used to protect bounded buffer manipulation
- pthread_cond_wait has lock mutex as a parameter so it can release the mutex if the thread must wait (to avoid deadlock). A waiting thread is awoke only after the system reacquires the mutex
- If a signal is performed and no waiting threads exist, then the signal is lost (ignored).
- while-loop needed since a signal can wake multiple threads with only one holding the mutex

pthread_cond_wait()

```
int pthread_cond_wait(
  pthread_cond_t *cond,                 // Condition to wait on
  pthread_mutex_t *mutex);              // Protecting mutex
int pthread_cond_timedwait(
  pthread_cond_t *cond,
  pthread_mutex_t *mutex,
  const struct timespec *abstime);      // Time-out value
```

Arguments:

- A condition variable to wait on.
- A mutex that protects access to the condition variable. The mutex is released before the thread blocks, and these two actions occur atomically. When this thread is later unblocked, the mutex is reacquired on behalf of this thread.

Return value:

0 if successful. Error code from `<errno.h>` otherwise.

pthread_cond_signal()

```
int pthread_cond_signal(
  pthread_cond_t *cond);                // Condition to signal
int pthread_cond_broadcast(
  pthread_cond_t *cond);                // Condition to signal
```

Arguments:

A condition variable to signal.

Return value:

0 if successful. Error code from `<errno.h>` otherwise.

Notes:

- These routines have no effect if there are no threads waiting on `cond`. In particular, there is no memory of the signal when a later call is made to `pthread_cond_wait()`.
- The `pthread_cond_signal()` routine may wake up more than one thread, but only one of these threads will hold the protecting mutex.
- The `pthread_cond_broadcast()` routine wakes up all waiting threads. Only one awakened thread will hold the protecting mutex.

Waiting on Multiple Condition Variables: - needs to test all condition variables simultaneously. Example:

```
 1  EatJuicyFruit()
 2  {
 3    pthread_mutex_lock(&lock);
 4    while(apples==0||oranges==0)
 5    {
 6      pthread_cond_wait(&more_apples, &lock);
6.5    if (oranges==0)
 7        pthread_cond_wait(&more_oranges, &lock);
 8    }
 9    /* Critical Section: Eat both an apple and an orange */
10    pthread_mutex_unlock(&lock);
11  }
```

Thread-Specific Data - useful if you want a global variable in the scope of all code, but can have different values for each thread. Example: Want any function to have access to a thread's ID by the global "index" (my_index):

Thread-Specific Data

```
pthread_key_t *my_index;
#define index(pthread_getspecific(my_index))
main()
{
   ...
   pthread_key_create(&my_index, 0);
   ...
}
void start_routine(int id)
{
   pthread_setspecific(my_index, id);
   ...
}
```

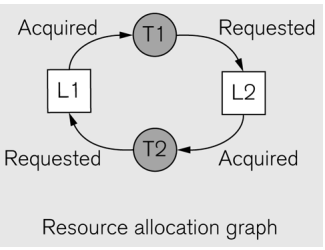
Notes:

Avoid accessing `index` in a tight inner loop because each access requires a procedure call.

Deadlock - Necessary Conditions for Deadlock: ALL MUST HOLD!

- mutual exclusion: resource(s) used in mutually exclusive fashion
- hold and wait: threads currently holding granted resources can request new resources
- no preemption: resources are released voluntarily by the thread holding it
- circular wait: a circular chain of two or more threads that are each waiting for a resource held by the next member of the chain.

Chapter 6 has a narrow view on the problem since it is dealing with threads locking mutexes.



Resource allocation graph

Thread T1 has acquired mutex lock L1 and is attempting to acquire the lock L2

Thread T2 has acquired mutex lock L2 and is attempting to acquire the lock L1

Strategies used for dealing with deadlock:

1) Ignore the problem - "Ostrich algorithm" - Is it a rare occurrence with minor consequences?
2) Detection and recovery - Periodically run an algorithm that checks for cycles in a resource-allocation graph. Rollback or kill a thread/process to break the cycle.
3) Deadlock Prevention - apply rules that prevent one of the 4 necessary conditions for deadlock
4) Deadlock Avoidance - dynamically avoid deadlock by careful resource allocation - make sure that there is some order that threads can finish execution without deadlock.

Pthread programmers typically utilize prevention or avoidance.

Lock hierarchies - deadlock prevention scheme which imposes an ordering on the locks (L1, L2, L3, etc.) and require that all threads acquire their locks in order.

a) How does that prevent a cycle?

b) If a thread did not know a priori what locks it needed, then it could be holding locks L3 and L5, and now wants to acquire L1. What are its options?

Acquiring and Releasing Mutexes

```
int pthread_mutex_lock(              // Lock a mutex
  pthread_mutex_t *mutex);
int pthread_mutex_unlock(            // Unlock a mutex
  pthread_mutex_t *mutex);
int pthread_mutex_trylock(           // Nonblocking lock
  pthread_mutex_t *mutex);
```

Arguments:
Each function takes the address of a mutex variable.

Return value:
0 if successful. Error code from `<errno.h>` otherwise.

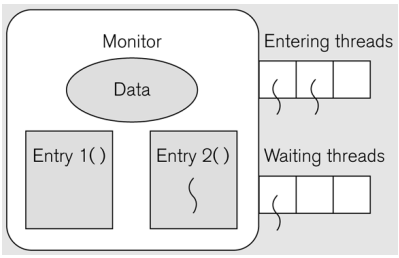
Notes:
The `pthread_mutex_trylock()` routine attempts to acquire a mutex but will not block. This routine returns the POSIX Threads constant `EBUSY` if the mutex is locked.

c) How might a pthread_mutex_trylock function help reduce this overhead?

Programming with mutexes and condition variable is very error prone. Some programming languages provide higher-level synchronization tools, e.g., monitors. A *monitor* is a mutual exclusion interface based on ADTs (abstract data types/objects):

- encapsulated state variables with accessible only by its public interface (methods, entry points)
- only one thread can execute the monitor's code at any time

```
 1  class BoundedBuffer
 2  {                                    // Implement a monitor
 3    private:
 4      pthread_mutex_t lock;            // Synchronization variables
 5      pthread_cond_t nonempty, nonfull;
 6      Item *buffer;                    // Shared data
 7      int in, out;                     // Cursors
 8      CheckInvariant();
 9
10    public:
11      BoundedBuffer(int size);         // Constructor
12      ~BoundedBuffer();                // Destructor
13      void put(Item x);
14      Item get();
15  }
16
17  // Constructor and Destructor
18  BoundedBuffer::Bounded(int size)
19  {
20    // Initialize synchronization variables
21    pthread_mutex_init(&lock, NULL);
22    pthread_cond_init(&nonempty, NULL);
23    pthread_cond_init(&nonfull, NULL);
24
25    // Initialize the buffer
26    buffer=new Item[size];
27    in=out=0;
28  }
29
30  BoundedBuffer::~BoundedBuffer()
31  {
32    pthread_mutex_destroy(&lock);
33    pthread_cond_destroy(&nonempty);
34    pthread_cond_destroy(&nonfull);
35    delete buffer;
36  }
37
38  // Member functions
39  BoundedBuffer::Put(Item x)
40  {
41    pthread_mutex_lock(&lock);
42    while(in-out==size)                // while buffer is full
43    {
44      pthread_cond_wait(&nonfull, &lock);
45    }
46    buffer[in%size]=x;
47    in++;
48    pthread_cond_signal(&nonempty);
49    pthread_mutex_unlock(&lock);
50  }
51
52  Item BoundedBuffer::Get()
53  {
54    pthread_mutex_lock(&lock);
55    while(in==out)                       // While buffer is empty
56    {
57      pthread_cond_wait(&nonempty, &lock);
58    }
59    x=buffer[out%size];
60    out++;
61    pthread_cond_signal(&nonfull);
62    pthread_mutex_unlock(&lock);
63    return x;
64  }
```



C++ monitor implementation of a Bounded Buffer

a) How does the Put method enforce mutex exclusion?

b) How does the Put method prevent adding to a full Bounded Buffer?

c) What invariants should the bounded buffer have?

d) What happens if a monitor method calls another monitor method?

Readers and Writers Example - Granularity Issues.
Multiple concurrent readers, but exclusive access for writers.

Original Textbook code with ERRORS - What are they?

```
 1  int readers;                              // Neg value=> active writer
 2  pthread_mutex_t lock;
 3  pthread_cond_t rBusy, wBusy;              // Use separate conditional vars
 4                                            // for readers and writers
 5  AcquireExclusive()
 6  {
 7    pthread_mutex_lock(&lock);
 8    while(readers !=0)
 9    {
10      pthread_cond_wait(&wBusy, &lock);
11    }
12    readers=-1;
13    pthread_mutex_unlock(&lock);
14  }
15
16  AcquireShared()
17  {
18    pthread_mutex_lock(&lock);
19    readWaiters++;
20    while(readers<0)
21    {
22      pthread_cond_wait(&rBusy, &lock);
23    }
24    readWaiters--;
25    pthread_mutex_unlock(&lock);
26  }
27
28  ReleaseExclusive()
29  {
30    pthread_mutex_lock(&lock);
31    readers=0;
32    pthread_cond_broadcast(&rBusy);         // Only wake up readers
33    pthread_mutex_unlock(&lock);
34  }
35
36  ReleaseShared(
37  {
38    int doSignal;
39
40    pthread_mutex_lock(&lock);
41    readers--;
42    doSignal=(readers==0)
43    pthread_mutex_unlock(&lock);
44    if(doSignal)                            // Signal executes outside
45    {                                       // of critical section
46      pthread_cond_signal(&wBusy);          // Wake up a writer
47    }
48  }
```

Corrected Textbook code????

```
1  int readers;                    // Negative value => active writer
2  pthread_mutex_t lock;
3  pthread_cond_t busy;            // Use one condition variable to
4                                  // indicate whether data if busy
5  AcquireExclusive()
6  {
7    pthread_mutex_lock(&lock);    // This code suffers from spurious
8    while(readers!=0)             // wake-ups!!!
9    {
10     pthread_cond_wait(&busy, &lock);
11   }
12   readers=-1;
13   pthread_mutex_unlock(&lock);
14 }
15
16 AcquireShared()
17 {
18   pthread_mutex_lock(&lock);
19   while(readers<0)
20   {
21     pthread_cond_wait(&busy, &lock);
22   }
23   readers++;
24   pthread_mutex_unlock(&lock);
25 }
26
27 ReleaseExclusive()
28 {
29   pthread_mutex_lock(&lock);
30   readers=0;
31   pthread_cond_broadcast(&busy);
32   pthread_mutex_unlock(&lock);
33 }
34
35 ReleaseShared(
36 {
37   pthread_mutex_lock(&lock);
38   readers--;
39   if(readers==0)
40   {
41     pthread_cond_signal(&busy);
42   }
43   pthread_mutex_unlock(&lock);
44 }
```

Thread Scheduling - how kernel threads are scheduled by the OS. Two Pthread approaches:

- *bounded threads* - each pthread is mapped to its own kernel thread with the OS scheduler assigning them to processors (default which we typically want)
- *unbounded threads* - multiple pthreads (user-level threads) are mapped to a single kernel thread which the OS schedulers. When a user-level thread is blocked by I/O, the whole kernel thread would be put into a wait state, so no threads in that group are running. Only advantages is they have less creating and destroying overhead.