• **Concept of thread**

• **Posix thread creation and termination**

• **Synchronization**
  • **Mutual exclusion**
  • Condition variables

• Broadcast

• Composite synchronization constructs
Broadcast

• Wake all threads that are waiting on the condition variable as opposed to a single thread.

  int pthread_cond_broadcast(pthread_cond_t *cond) ;

• A thread performs a wait on a condition variable until a specified time expires.

  int pthread_cond_timedwait(pthread_cond_t * cond,
                              pthread_mutex_t *mutex,
                              const struct timespec *abstime) ;
Customization

- Pthread allows the customization of threads and synch. through *attribute objects*

- **Attribute object**
  - a data-structure that describes entity (thread, mutex, condition variable) properties

- **Creation process**
  - step 1: create an object with default properties
  - step 2: modify the object as required
Attr. of Threads

- Modify the attr object through a set of functions
  - `pthread_attr_setinheritsched()`: whether the thread inherits the scheduling policy of its parent
  - `pthread_attr_setschedpolicy()`: what scheduling policy the thread uses
  - `pthread_attr_setstacksize()`: the stack size of the thread
  - `pthread_attr_setdetachstate()`, `pthread_attr_setguardsize_np()`, `pthread_attr_setschedparam()`
Attr. of Mutex

Three types of mutex locks:

normal (default): `PTHREAD_MUTEX_NORMAL_NP`
when a thread who has acquired a lock tries to acquire the same lock again, deadlock.

error check: `PTHREAD_MUTEX_ERRORCHECK_NP`
when a thread with a lock tries to lock the lock again, return error.

recursive: `PTHREAD_MUTEX_RECURSIVE_NP`
allows relock to happen. Use a counter to keep track.
Compose Synch Constructs

- Built using the basic constructs
- Offer conveniences to certain scenarios

- construct 1: read-write locks
- construct 2: barriers
Construct 1: Read-Write Locks

- Useful when there are frequent reads but infrequent writes

Motivating example:

```c
void *find_min(void *list_ptr) {
    int *partial_list_pointer, my_min, i;
    my_min = MIN_INT;
    partial_list_pointer = (int *) list_ptr;
    for (i = 0; i < partial_list_size; i++)
        if (partial_list_pointer[i] < my_min)
            my_min = partial_list_pointer[i];
    /* lock the mutex associated with minimum_value and
     update the variable as required */
    pthread_mutex_lock(&minimum_value_lock);
    if (my_min < minimum_value)
        minimum_value = my_min;
    /* and unlock the mutex */
    pthread_mutex_unlock(&minimum_value_lock);
    pthread_exit(0);
}
Implementation

• Allow multiple readers
• No readers or other writers are allowed if there is a writer
• Uses a counter, a mutex, and two condition variables

```c
typedef struct {
    int readers;
    int writer;
    pthread_cond_t readers_proceed;
    pthread_cond_t writer_proceed;
    int pending_writers;
    pthread_mutex_t read_write_lock;
} mylib_rwlock_t;
```
void mylib_rwlock_init (mylib_rwlock_t *l) {
    l -> readers = l -> writer = l -> pending_writers = 0;
    pthread_mutex_init(&(l -> read_write_lock), NULL);
    pthread_cond_init(&(l -> readers_proceed), NULL);
    pthread_cond_init(&(l -> writer_proceed), NULL);
}

void mylib_rwlock_rlock(mylib_rwlock_t *l) {
    /* if there is a write lock or pending writers, perform condition
     * wait... else increment count of readers and grant read lock */
    pthread_mutex_lock(&(l -> read_write_lock));
    while (((l -> pending_writers > 0) || (l -> writer > 0))
        pthread_cond_wait(&(l -> readers_proceed),
            &(l -> read_write_lock));
    l -> readers ++;
    pthread_mutex_unlock(&(l -> read_write_lock));
}
void mylib_rwlock_wlock(mylib_rwlock_t *l) {
    /* if there are readers or writers, increment pending writers
    count and wait. On being woken, decrement pending writers
    count and increment writer count */

    pthread_mutex_lock(&(l -> read_write_lock));
    while ((l -> writer > 0) || (l -> readers > 0)) {
        l -> pending_writers ++;
        pthread_cond_wait(&(l -> writer_proceed),
                         &(l -> read_write_lock));
    }
    l -> pending_writers --;
    l -> writer ++
    pthread_mutex_unlock(&(l -> read_write_lock));
}

Read-Write Lock

```c
void mylib_rwlock_unlock(mylib_rwlock_t *l) {
    /* if there is a write lock then unlock, else if there are
    read locks, decrement count of read locks. If the count
    is 0 and there is a pending writer, let it through, else
    if there are pending readers, let them all go through */

    pthread_mutex_lock(&l->read_write_lock);
    if (l->writer > 0)
        l->writer = 0;
    else if (l->readers > 0)
        l->readers --;
    pthread_mutex_unlock(&l->read_write_lock);
    if ((l->readers == 0) && (l->pending_writers > 0))
        pthread_cond_signal(&l->writer_proceed);
    else if (l->readers > 0)
        pthread_cond_broadcast(&l->readers_proceed);
}
```
Example of Usage

Find Minimum Number

```c
void *find_min_rw(void *list_ptr) {
    int *partial_list_pointer, my_min, i;
    my_min = MIN_INT;
    partial_list_pointer = (int *) list_ptr;
    for (i = 0; i < partial_list_size; i++)
        if (partial_list_pointer[i] < my_min)
            my_min = partial_list_pointer[i];
    /* lock the mutex associated with minimum_value and
     * update the variable as required */
    mylib_rwlock_rlock(&read_write_lock);
    if (my_min < minimum_value) {
        mylib_rwlock_unlock(&read_write_lock);
        mylib_rwlock_wlock(&read_write_lock);
        minimum_value = my_min;
    }
    /* and unlock the mutex */
    mylib_rwlock_unlock(&read_write_lock);
    pthread_exit(0);
}
```
Construct 2: Barriers

- Recall that a barrier call holds a thread until all other threads participating in the barrier have reached the barrier
- Can also be implemented with a counter, a mutex, and a condition variable
typedef struct {
    pthread_mutex_t count_lock;
    pthread_cond_t ok_to_proceed;
    int count;
} mylib_barrier_t;

void mylib_init_barrier(mylib_barrier_t *b) {
    b -> count = 0;
    pthread_mutex_init(&(b -> count_lock), NULL);
    pthread_cond_init(&(b -> ok_to_proceed), NULL);
}

void mylib_barrier (mylib_barrier_t *b, int num_threads) {
    pthread_mutex_lock(&(b -> count_lock));
    b -> count ++;
    if (b -> count == num_threads) {
        b -> count = 0;
        pthread_cond_broadcast(&(b -> ok_to_proceed));
    } else {
        while (pthread_cond_wait(&(b -> ok_to_proceed),
                &(b -> count_lock)) != 0);
        pthread_mutex_unlock(&(b -> count_lock));
    }
}
Performance Issue

- Linear to the number of threads
  - Threads are released one by one
- Can you reduce this overhead to sub-linear?

Use \( \frac{n}{2} \) condition vars.

“Introduction to Parallel Computing” 2nd Edition
see ch. 7 for details.
Tips for Thread Programming

• Do not assume any order of threads; use synch. if order is needed.
• Set up all the requirements for a thread before actually creating the thread.
• For producer-consumer, make sure the producer places the data before it is consumed and that intermediate buffers are guaranteed to not overflow.
• Where possible, define and use group synchronizations and data replication for performance.
Pthread

- Concept of thread
- Posix thread creation and termination
- Synchronization
  - Mutual exclusion
  - Condition variables
- Broadcast
- Composite synchronization constructs
Pthread

• Concept of thread
• Posix thread creation and termination
• Synchronization
  • Mutual exclusion
  • Condition variables
• Broadcast
• Composite synchronization constructs
Locality Analysis for Hardware Cache
Hardware Cache Line (Block)

• The unit of memory in a cache is a line (block)
• A line may have 1 or more words
  • A word is typically 4 or 8 bytes

• Accessing any member of the line brings the entire line into cache (replacing whatever was there previously)
• Interference - multiple lines that map to the same cache location and need to be in the cache at the same time
Temporal Reuse/Locality

- Reuse of the same memory location.

```
Do I = 2, N
A(I) = A(I-1)
ENDDO
```

Access the same element one iteration apart
Spatial Reuse/Locality

• Reuse of a cache block with a nearby memory reference

Do I = 1, N

A(I) = Access the same cache line on successive iterations

ENDDO
Software Adaptation for Hardware Cache

- Two main ways
  - Improving cache utilization
    - reduce both latency and bandwidth requirement
    - enhance two kinds of locality
      - temporal locality
      - spatial locality
  - Prefetching
    - reduce latency but not bandwidth requirement
- Need to understand program locality
  - when do data accesses happen
Locality Analysis Classification
(based on time)

- Compile time
  - effective for scalars or affine indices
  - for loop nests with linear index expressions
  - need to approximate branches, recursion, and indirect data access
- Profiling
  - accurate for one input
  - need to predict behavior in other inputs
- Run-time
  - adapt to different access patterns
  - need to be efficient
Locality Analysis Classification (based on features)

• Stream  
  • frequent sequences of data access in a trace
• Distance  
  • patterns of the reuse distance of data access in a trace
• Stride  
  • patterns of the stride of data access in a trace
• Dependence  
  • dependences among data references in a program  
  • static measurement of frequency, distance, and stride