Lecture 5: Parallel Programming
with Thread (Part2)
Controlling Thread and Synchronization Attributes

*Attributes object:* data structure that describes entity (e.g. thread, mutex, condition variable) properties.

*Example:* for threads user can specify the scheduling policy, stack size, etc.

*Advantages:*

- Separates the issues of program semantics and implementation.
- Improves modularity and readability of the programs.
- Allows the user to modify the program easily.
Attributes Objects for Threads

Creating an attributes object for threads:

```c
int pthread_attr_init( pthread_attr_t *attr);
```

The attributes of object `attr` will be initialized to default values.

Destroying the attributes object:

```c
int pthread_attr_destroy( pthread_attr_t *attr);
```

Changing individual properties associated with attributes:

- Set detach state: `pthread_attr_setdettachstate()`
- Set the stack size: `pthread_attr_setstacksize()`
- Set scheduling policy inheritance: `pthread_attr_setinheritsched()`
- Set scheduling policy: `pthread_attr_setschedpolicy()`
- Set scheduling parameters: `pthread_attr_setschedparam()`
Attributes Objects for Mutexes

Three types of mutex-locks:

• **Normal mutex**: only one thread is allowed to lock a normal mutex once at any point in time.

• **Recursive mutex**: allows a single thread to lock a mutex multiple times (a lock counter is incremented at each mutex lock).
  
  Another thread can lock the mutex if the counter becomes 0.
  
  Useful when a thread function needs to call itself recursively.

• **Errorcheck mutex**: similar to a normal mutex. But when a thread attempts a lock on a mutex it has already locked, instead of deadlocking, it returns an error.
Attributes Objects for Mutexes

Creating an attributes object for mutexes:

```c
int pthread_mutexattr_init(pthread_mutexattr_t *attr);
```
– The attributes of object `attr` will be initialized to default values.
– The default type of mutex is a normal mutex.

Setting the type of a mutex:

```c
int pthread_mutexattr_settype_np(pthread_mutexattr_t *attr, int type);
```
- `type` can take one of the values:
  - PTHREAD_MUTEX_NORMAL_NP
  - PTHREAD_MUTEX_RECURSIVE_NP
  - PTHREAD_MUTEX_ERRORCHECK_NP
Example: Searching a Binary Tree

```c
search_tree(void *tree_ptr)
{
    struct node *node_pointer;
    node_pointer = (struct node *) tree_ptr;
    pthread_mutex_lock(&tree_lock);
    if (is_search_node(node_pointer) == 1) {
        /* solution is found here */
        print_node(node_pointer);
        pthread_mutex_unlock(&tree_lock);
        return(1);
    }
    else {
        if (tree_ptr -> left != NULL)
            search_tree((void *) tree_ptr -> left);
        if (tree_ptr -> right != NULL)
            search_tree((void *) tree_ptr -> right);
    }
    printf("Search unsuccessful\n");
    pthread_mutex_unlock(&tree_lock);
}
```
Thread Cancellation

A thread may cancel (terminate) itself or cancel other threads:

```c
int pthread_cancel(pthread_t thread);
```

It is not guaranteed that the specified thread will act on the cancellation. Threads can protect themselves against cancellation.
Composite Synchronization Constructs

• By design, Pthreads provide support for a basic set of operations.

• Higher level constructs can be built using basic synchronization constructs.

• We discuss two such constructs - read-write locks and barriers.
Implementation of Read-Write Locks

- In many applications, a data structure is read frequently but written infrequently. For such applications, we should use read-write locks.
- A read lock is granted when there are other threads that may already have read locks.
- If there is a write lock on the data (or if there are queued write locks), the thread performs a condition wait.
- With this description, we can design functions for read locks `mylib_rwlock_rlock`, write locks `mylib_rwlock_wlock`, and unlocking `mylib_rwlock_unlock`.
Read-Write Locks

The lock data type `mylib_rwlock_t` holds the following:

- a count of the number of readers,
- the writer (a 0/1 integer specifying whether a writer is present),
- a condition variable `readers_proceed` that is signaled when readers can proceed,
- a condition variable `writer_proceed` that is signaled when one of the writers can proceed,
- a count `pending_writers` of pending writers, and
- a mutex `read_write_lock` associated with the shared data structure

```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;
```
Read-Write Locks

```c
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));
}
```

What does ‘l->readers’ record?
Read-Write Locks (cont’d)

```c
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);
}

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: Computing the Minimum (using r/w mutex lock)

```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);
}
```
Example: Hash Tables (using regular mutex lock)

```c
manipulate_hash_table(int entry) {
    int table_index, found;
    struct list_entry *node, *new_node;

    table_index = hash(entry);
    pthread_mutex_lock(&hash_table[table_index].list_lock);
    found = 0;
    node = hash_table[table_index].next;
    while ((node != NULL) && (!found)) {
        if (node -> value == entry)
            found = 1;
        else
            node = node -> next;
    }
    pthread_mutex_unlock(&hash_table[table_index].list_lock);
    if (found)
        return(1);
    else
        insert_into_hash_table(entry);
}

Use mutex lock in the function and search the list again
```
Example: Hash Tables (using read-write lock)

```c
manipulate_hash_table(int entry)
{
    int table_index, found;
    struct list_entry *node, *new_node;

    table_index = hash(entry);
    mylib_rwlock_rlock(&hash_table[table_index].list_lock);
    found = 0;
    node = hash_table[table_index].next;
    while ((node != NULL) && (!found)) {
        if (node -> value == entry)
            found = 1;
        else
            node = node -> next;
    }
    mylib_rwlockunlock(&hash_table[table_index].list_lock);
    if (found)
        return(1);
    else
        insert_into_hash_table(entry);
}
```

Use write lock in the function
Barrier Implementation

• Implementing counter-based barriers with two phases:

  ➢ A process enters arrival phase and does not leave this phase until all processes have arrived in this phase.

  ➢ Then processes move to departure phase and are released.
Sequential Barrier Implementation

Centralized counter implementation (a linear barrier):

- Increment and check for $p$
- $P_0$: Counter, $C$
- $P_1$: Barrier();
- $P_{p-1}$: Barrier();

Processes

Diagram shows the sequential barrier implementation with processes $P_0$, $P_1$, and $P_{p-1}$ and a centralized counter $C$. The counter is incremented and checked by each process before proceeding. The diagram illustrates the linear barrier concept where processes wait in sequence to synchronize their execution.
Sequential Barrier

```c
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));
}
```

Lower bound on execution: $O(n)$
Tree Barrier Implementation

Idea: for n threads use n/2 condition variable-mutex pairs.
Tree Barrier

typedef struct barrier_node {
    pthread_mutex_t count_lock;
    pthread_cond_t ok_to_proceed_up;
    pthread_cond_t ok_to_proceed_down;
    int count;
} mylib_barrier_t_internal;

typedef struct barrier_node mylog_logbarrier_t[MAX_THREADS];
pthread_t p_threads[MAX_THREADS];
pthread_attr_t attr;

void mylib_init_barrier(mylog_logbarrier_t b) {
    int i;
    for (i = 0; i < MAX_THREADS; i++) {
        b[i].count = 0;
        pthread_mutex_init(&b[i].count_lock, NULL);
        pthread_cond_init(&b[i].ok_to_proceed_up, NULL);
        pthread_cond_init(&b[i].ok_to_proceed_down, NULL);
    }
}
void mylib_logbarrier (mylog_logbarrier_t b, int num_threads,
    int thread_id) {
    int i, base, index;
    i = 2;
    base = 0;
    do {
        index = base + thread_id / i;
        if (thread_id % i == 0) {
            pthread_mutex_lock(&b[index].count_lock);
            b[index].count ++;
            while (b[index].count < 2)
                pthread_cond_wait(&b[index].ok_to_proceed_up),
                    &b[index].count_lock);
            pthread_mutex_unlock(&b[index].count_lock);
        }
        else {
            pthread_mutex_lock(&b[index].count_lock);
            b[index].count ++;
            if (b[index].count == 2)
                pthread_cond_signal(&b[index].ok_to_proceed_up);
            while (pthread_cond_wait(&b[index].ok_to_proceed_down),
                    &b[index].count_lock) != 0);
            pthread_mutex_unlock(&b[index].count_lock);
            break;
        }
        base = base + num_threads/i;
        i = i * 2;
    } while (i <= num_threads);
    i = i / 2;
    for (; i > 1; i = i / 2) {
        base = base - num_threads/i;
        index = base + thread_id / i;
        pthread_mutex_lock(&b[index].count_lock);
        b[index].count = 0;
        pthread_cond_signal(&b[index].ok_to_proceed_down);
        pthread_mutex_unlock(&b[index].count_lock);
    }
}
Comparison of Barrier Performance

Execution time of 1000 sequential and logarithmic barriers as a function of number of threads on a 32 processor SGI Origin 2000.