

CS410: Parallel Computing

Spring 2024

Nikhil Hegde
Milind Chabbi

Shared-Memory Programming (using POSIX Threads)
contd..

Overheads of Locking

- Locks enforce serialization
 - threads must execute critical sections one at a time
- Large critical sections can seriously degrade performance
- Reduce overhead by overlapping computation with waiting

```
int pthread_mutex_trylock(pthread_mutex_t *mutex_lock)
```

- acquire lock if available
- return EBUSY if not available
- enables a thread to do something else if lock unavailable

Pthreads API

- Subroutines grouped into four major categories:

- 1. Thread Management –**

Routines that deal with thread creation, join, detach, etc. Also, query thread attributes

- 2. Mutexes –**

Routines that deal with synchronization. Creating, destroying, locking, and unlocking a mutex (abbr. for “mutual exclusion”). Also, set mutex attributes

- 3. Condition Variables –**

Routines addressing communication between threads that share a mutex. Based upon programmer specified conditions. Includes functions to create, destroy, wait and signal based upon specified variable values. Also, set/query condition variable attributes.

- 4. Synchronization**

Routines that manage read-write locks and barriers



Condition Variables

- Allows one thread to signal to another thread that the condition is true
- Prevents programmer from looping on a mutex call.
 - Allows to poll if the condition is true

Condition Variables for Synchronization

Condition variable: associated with a **predicate** and a **mutex**

- Using a condition variable

- thread can block itself until a condition becomes true

- thread locks a mutex

- tests a predicate defined on a shared variable

- if predicate is false, then wait on the condition variable

- waiting on condition variable unlocks associated mutex

- when some thread makes a predicate true

- that thread can signal the condition variable to either

- wake one waiting thread

- wake all waiting threads

- when thread releases the mutex, it is passed to first waiter

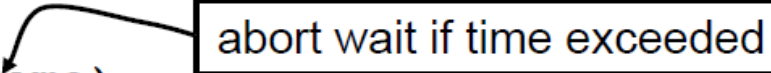
Pthread Condition Variable API

/ initialize or destroy a condition variable */*

```
int pthread_cond_init(pthread_cond_t *cond,  
    const pthread_condattr_t *attr);  
int pthread_cond_destroy(pthread_cond_t *cond);
```

/ block until a condition is true */*

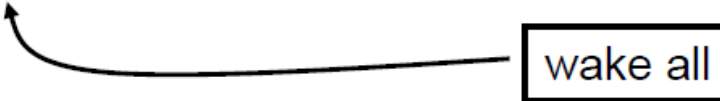
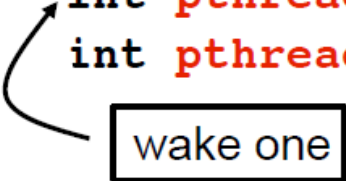
```
int pthread_cond_wait(pthread_cond_t *cond,  
    pthread_mutex_t *mutex);  
int pthread_cond_timedwait(pthread_cond_t *cond,  
    pthread_mutex_t *mutex,  
    const struct timespec *wtime);
```



abort wait if time exceeded

/ signal one or all waiting threads that condition is true */*

```
int pthread_cond_signal(pthread_cond_t *cond);  
int pthread_cond_broadcast(pthread_cond_t *cond);
```



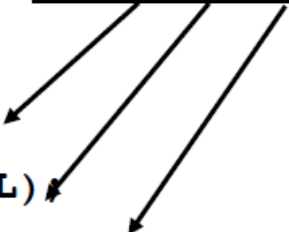
wake one

wake all

Condition Variable Producer-Consumer (main)

```
pthread_cond_t cond_queue_empty, cond_queue_full;
pthread_mutex_t task_queue_cond_lock;
int task_available;
/* other data structures here */
main() {
    /* declarations and initializations */
    task_available = 0;
    pthread_init();
    pthread_cond_init(&cond_queue_empty, NULL);
    pthread_cond_init(&cond_queue_full, NULL);
    pthread_mutex_init(&task_queue_cond_lock, NULL);
    /* create and join producer and consumer threads */
}
```

default
initializations



Producer Using Condition Variables

```
void *producer(void *producer_thread_data) {  
    int inserted;  
    while (!done()) {  
        create_task();  
        pthread_mutex_lock(&task_queue_cond_lock);  
        while (task_available == 1)  
            pthread_cond_wait(&cond_queue_empty,  
                             &task_queue_cond_lock);  
        insert_into_queue();  
        task_available = 1;  
        pthread_cond_signal(&cond_queue_full);  
        pthread_mutex_unlock(&task_queue_cond_lock);  
    }  
}
```

note
loop {

releases mutex on wait

reacquires mutex when woken

Consumer Using Condition Variables

```
void *consumer(void *consumer_thread_data) {  
    while (!done()) {  
        pthread_mutex_lock(&task_queue_cond_lock);  
        while (task_available == 0)  
            pthread_cond_wait(&cond_queue_full,  
                             &task_queue_cond_lock);  
        my_task = extract_from_queue();  
        task_available = 0;  
        pthread_cond_signal(&cond_queue_empty);  
        pthread_mutex_unlock(&task_queue_cond_lock);  
        process_task(my_task);  
    }  
}
```

releases mutex on wait

note loop {

reacquires mutex when woken

Pthreads API

- Subroutines grouped into four major categories:
 1. **Thread Management** –
Routines that deal with thread creation, join, detach, etc. Also, query thread attributes
 2. **Mutexes** –
Routines that deal with synchronization. Creating, destroying, locking, and unlocking a mutex (abbr. for “mutual exclusion”). Also, set mutex attributes
 3. **Condition Variables** –
Routines addressing communication between threads that share a mutex. Based upon programmer specified conditions. Includes functions to create, destroy, wait and signal based upon specified variable values. Also, set/query condition variable attributes.
 4. **Composite Synchronization Constructs**
Routines that manage read-write locks and barriers

Reader-Writer Locks – pthread_rwlock_t

- Given an array of elements arr
- Operations permitted on arr:
 - Search: `search(arr, x)` //checks if `arr[i]==x?`
 - Update: `update(arr, i, y)` //updates `arr[i]=y`
- There are frequent number of search operations
- A mutex to coordinate access to arr is inefficient
 - OK to grant access to arr when two threads are calling search simultaneously
 - NOT OK to grant access to arr when at least one of the thread is calling update simultaneously

Reader-Writer Locks – pthread_rwlock_t

- WriteLock: NOT OK to grant access to arr when at least one of the thread is calling update simultaneously
 - If multiple writers are trying to simultaneously update the array, queue them.

```
int pthread_rwlock_wrlock(pthread_rwlock_t *rwlock );
```

- ReadLock: OK to grant access to arr when two threads are calling search simultaneously
 - Do not grant access if WriteLock is held OR there are some writers waiting.

```
int pthread_rwlock_rdlock(pthread_rwlock_t *rwlock );
```

Read-Write Lock Sketch

- Use a data type with the following components
 - a count of the number of active readers
 - 0/1 integer specifying whether a writer is active
 - a condition variable `readers_proceed`
 - signaled when readers can proceed
 - a condition variable `writer_proceed`
 - signaled when one of the writers can proceed
 - a count `pending_writers` of pending writers
 - a mutex `read_write_lock`
 - controls access to the reader/writer data structure

Barriers – pthread_barrier_t

- Use barrier synchronization When waiting for several independent tasks is required before proceeding with an overall task

```
pthread_barrier_t mybarrier;
```

```
pthread_barrierattr_t attr;
```

```
unsigned int count;
```

- `pthread_barrier_init(&mybarrier, &attr, count)`
- `pthread_barrier_wait(&mybarrier);`
- `pthread_barrier_destroy(&mybarrier)`
- `pthread_barrierattr_init(attr)`
- `pthread_barrierattr_destroy(attr)`
- `pthread_barrier_setpshared(&mybarrier, int)`
- `pthread_barrier_getpshared(&mybarrier)`


Suggested Reading

- [POSIX Threads Programming | LLNL HPC Tutorials](#)
- Chapter 7. “Introduction to Parallel Computing” by Ananth Grama, Anshul Gupta, George Karypis, and Vipin Kumar. Addison Wesley, 2003

Thanks to: [LLNL HPC Tutorials](#), Samuel Midkiff (ECE563), and John Mellor-Crummy (COMP422) for the slides.

Shared Address Space Programming Models

A brief taxonomy

- Lightweight processes and threads
 - all memory is global
 - examples: Pthreads, Cilk (lazy, lightweight threads)
- Process-based models
 - each process's data is private, unless otherwise specified
 - example: Linux shget/shmat/shmdt
- Directive-based models, e.g., OpenMP  Next
 - shared and private data
 - facilitate thread creation and synchronization
- Global address space programming languages
 - shared and private data
 - hardware typically distributed memory, perhaps not shared
 - examples: Unified Parallel C, Co-array Fortran

CS410: Parallel Computing

Spring 2024

Nikhil Hegde
Milind Chabbi

Shared-Memory Programming OpenMP - I

What is OpenMP

- An open standard for **shared-memory programming** in C, C++, and Fortran
- Supported by IBM, Intel, GNU and others
- *Directive-based* programming approach removes the need for explicitly setting up initialization, mutexes, and condition variables.

What is OpenMP?

Open specifications for Multi Processing

- An API for explicit multi-threaded, shared memory parallelism
- Three components
 - compiler directives
 - runtime library routines
 - environment variables
- Higher-level programming model than Pthreads
 - implicit mapping and load balancing of work
- Portable
 - API is specified for C/C++ and Fortran
 - implementations on almost all platforms
- Standardized

OpenMP Is Not

- An automatic parallel programming model
 - parallelism is explicit
 - programmer full control (and responsibility) over parallelization
- Meant for distributed-memory parallel systems (by itself)
 - designed for shared address spaced machines
- Necessarily implemented identically by all vendors
- Guaranteed to make the most efficient use of shared memory
 - no data locality control

OpenMP at a Glance

