CS410: Parallel Computing

Spring 2024

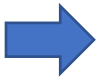
Nikhil Hegde
Milind Chabbi

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

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

Mutexes – Protecting Shared Data

- Mutex ("mutual exclusion") – primary means of protecting data when write occurs

An example of a race condition

Thread 1	Thread 2	Balance
Read balance: $1000		$1000
	Read balance: $1000	$1000
	Deposit $200	$1000
Deposit $200		$1000
Update balance $1000+$200		$1200
	Update balance $1000+$200	$1200

Mutex Variables Overview | LLNL HPC Tutorials

- `Balance` must be protected

Mutexes – Protecting Shared Data

- Mutex variable acts like a lock on the shared data
  - Only one Pthread can lock (or acquire / own) a mutex at a time. All other threads wanting to lock must wait till the thread owning the lock unlocks (or releases)
  - What happens when the thread owning the mutex tries to lock the mutex again?
    - Normal – thread deadlocks
    - Recursive – increment a count on the number of times locked, unlock when count reaches zero
    - Errorcheck –
      - report error when thread tries to lock a mutex it has already locked
      - Report error when thread tries to unlock a mutex that some other thread has locked

Pthread Mutex APIs

- Variables are of type `pthread_mutex_t`
- Initialized as:
  - `pthread_mutex_t mymutex = PTHREAD_MUTEX_INITIALIZER;`
  - `pthread_mutex_init(mymutex, attr)`
  - Mutex is unlocked initially
- `pthread_mutex_init(mutex,attr)`
- `pthread_mutex_destroy(mutex)`
- `pthread_mutexattr_init(attr)`
- `pthread_mutexattr_destroy(attr)`
- `pthread_mutex_lock(mutex)`
- `pthread_mutex_unlock(mutex)`
- `pthread_mutexattr_trylock(mutex)`

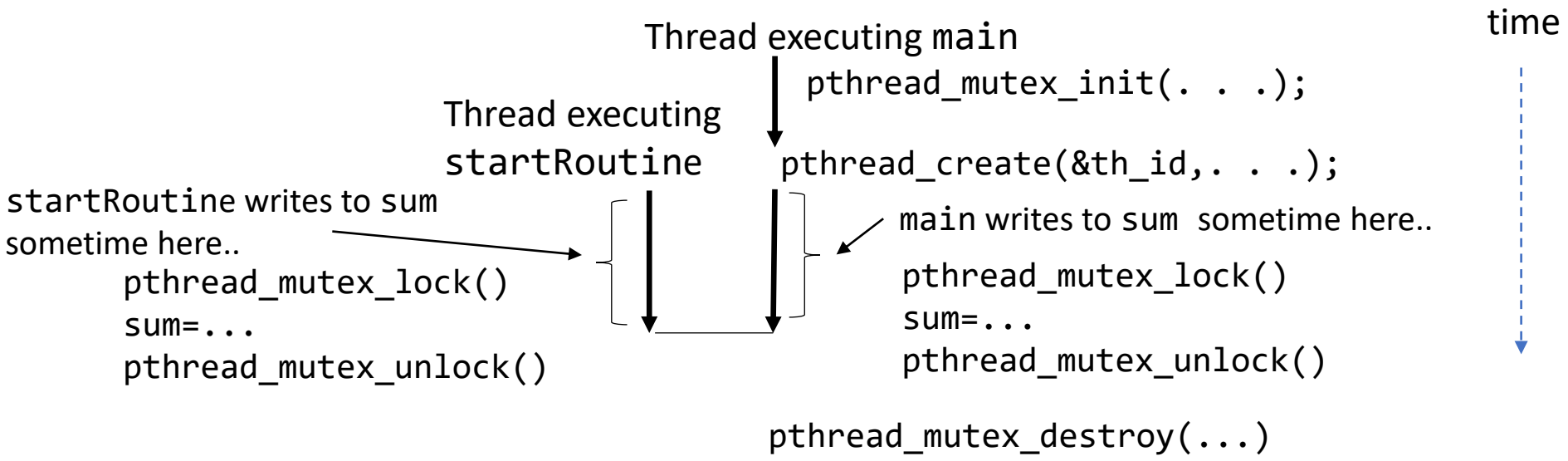
Exercise: When more than one thread is waiting for a locked mutex, which thread will be granted the lock first after the mutex is released?

Synchronization: pthread_mutex_t

```
pthread_mutex_t mymutex;
int sum;
int main() {
 //some code here... (optional)
 pthread_mutex_init(&mymutex, NULL);
 //some code here... (optional)
 pthread_create(&th_id, &myattr, startRoutine, NULL);
//some code here that writes to sum (OR at least
one of startRoutine or main writing to sum)
 pthread_mutex_lock(&mymutex);
 sum=...
 pthread_mutex_unlock(&mymutex);
 //some code here (optional)
 pthread_mutex_destroy(&mymutex);

}
```

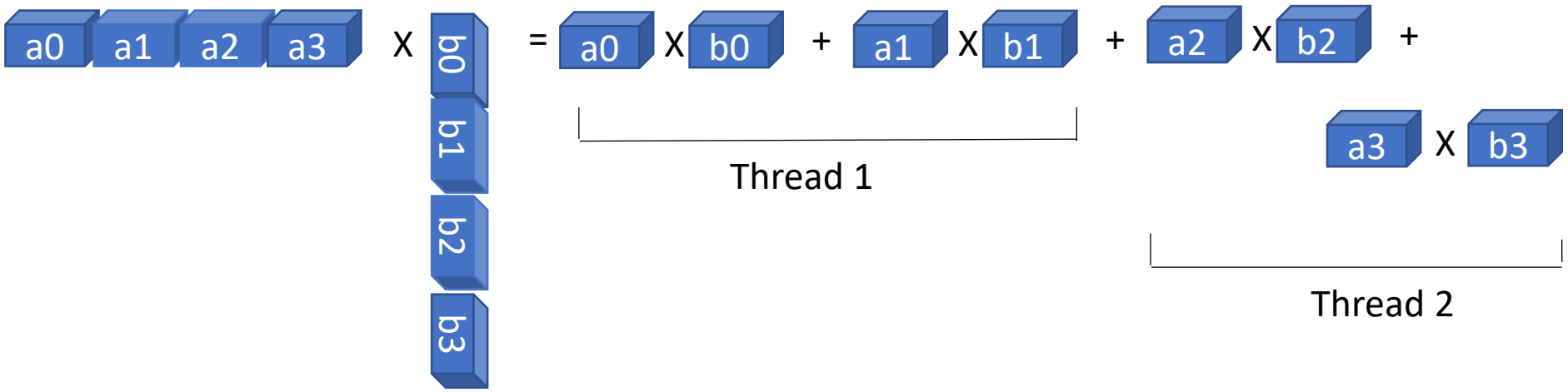
```
int startRoutine() {
...
pthread_mutex_lock(&mymutex);
sum=...
pthread_mutex_unlock(&mymutex);
...

}
```

Synchronization: pthread_mutex_t



Mutex - Example

- Compute Dot Product in Parallel



- Partial result is computed by each thread. Value of the dot product is sum of partial results.
- Demo

Producer-Consumer Using Mutex Locks

Constraints

- **Producer thread**
  - —must not overwrite the shared buffer until previous task has picked up by a consumer
- **Consumer thread**
  - —must not pick up a task until one is available in the queue
  - —must pick up tasks one at a time

Producer-Consumer Using Mutex Locks

```
pthread_mutex_t task_queue_lock;
int task_available;
...
main() {
    ...
    task_available = 0;
    pthread_mutex_init(&task_queue_lock, NULL);
    ...
}
void *producer(void *producer_thread_data) {
    ...
    while (!done()) {
        inserted = 0;
        create_task(&my_task);
        while (inserted == 0) {
            pthread_mutex_lock(&task_queue_lock);
            if (task_available == 0) {
               insert_into_queue(my_task); task_available = 1;
               inserted = 1;
            }
            pthread_mutex_unlock(&task_queue_lock);
        }
    }
}
```

critical section

Producer-Consumer Using Locks

```
void *consumer(void *consumer_thread_data) {
    int extracted;
    struct task my_task;
    /* local data structure declarations */
    while (!done()) {
       extracted = 0;
       while (extracted == 0) {
          pthread_mutex_lock(&task_queue_lock);
          if (task_available == 1) {
             extract_from_queue(&my_task);
             task_available = 0;
             extracted = 1;
          }
          pthread_mutex_unlock(&task_queue_lock);
       }
       process_task(my_task);
    }
```