Announcements

- **Reading**
  - Today: Pthreads book - Chapters 2 & 3
- **Photos were taken of the class**
Pthreads

- Allows multiple threads of control on a process
- Basic operations:
  - `pthread_create(&threadId, attr, func, arg)`
    - creates a new thread
    - `threadId` is the id of the new thread
    - `attr` are special attributes of the thread (pass NULL)
    - `func` is a pointer to a function to run
    - `arg` is an argument to that function
  
  - first thread of control must not exit (will kill other threads)
    - `pthread_join(threadId, status)`
      - wait for a specific thread to terminate
Using Locks for the Critical Section

- **Lock:**
  - if no thread has the lock mark it locked and return
  - if another thread has the lock, wait

- **Unlock:**
  - release the lock
  - if other threads waiting, notify one or all of them

- **Called mutexs in pthreads**
  - pthread_mutex is the data type
  - pthread_mutex_init used to initialize it
  - pthread_mutex_lock locks it
  - pthread_mutex_unlock releases it

- **Lock Grainularity**
  - want to lock enough to protect accesses
  - don’t want to lock too much to slow down the program
Condition Variables

• Allow threads to wait on the value of a variable
  - wait until the list is non-empty for example
  - allows one thread to signal to another thread that something has changed
    • threads may sleep waiting to be notified of this change

• Can unlock and re-lock a mutex before/after suspend

wait for count to be >= 1
  pthread_mutex_lock(&count_mutex);
  while (count <= 0) {
    pthread_cond_wait(&count_condvar, &count_mutex);
  }
  pthread_unlcok(&count_mutex);
update count:
  pthread_mutex_lock(&count_mutex);
  count++;
  pthread_mutex_unlock(&count_mutex);
  pthread_cond_signal(&count_condvar);
Consider the following program

**T1:**
\[
\text{count++ -- in C one statement, but really multiple instructions}
\]
\[
\text{load r1, count}
\]
\[
\text{add r1, 1, r1}
\]
\[
\text{store r1, count}
\]

**T2:**
\[
\text{count++ -- in C one statement, but really multiple instructions}
\]
\[
\text{load r2, count}
\]
\[
\text{add r2, 1, r2}
\]
\[
\text{store r2, count}
\]

**What happens when T1 is preempted right after the load**
With Synchronization

T1:
```c
pthread_mutex_lock(&mylock)
count++
pthread_mutex_unlock(&mylock)
```

T2:
```c
pthread_mutex_lock(&mylock)
count++
pthread_mutex_unlock(&mylock)
```

Only one thread at a time gets to update the count
Queue Project

- Need to coordinate access to shared resources
  - use mutex to guard access to a shared data structure
- Queue abstraction is **very** useful
  - enqueue: add item to queue
  - dequeue: remove item, **block** if not ready
  - head: return head of queue without dequeue
  - probe: test if the queue is empty

- must use a mutex to protect access to queue
- build a producer/consumer test program

- **Multiple application threads**
  - our test application is multi-threaded
  - must be able to support multiple threads trying to en-queue
Link State Routing

- Used on the ARPANET after 1979
- Each Router:
  - computes metric to neighbors and sends to every other router
  - each router computes the shortest path based on received data
- Needs to estimate time to neighbor
  - best approach is send an ECHO packet and time response
- Distributing Info to other routers
  - each router may have a different view of the topology
  - simple idea: use flooding
  - refinements
    - use age sequence number to damp old packets
    - use acks to permit reliable delivery of routing info