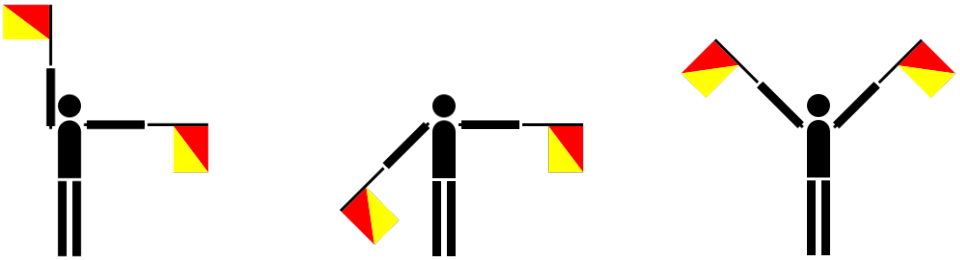
CS 470
Spring 2017

Mike Lam, Professor



Semaphores and Conditions

Synchronization mechanisms

- Busy-waiting (wasteful!)
- Atomic instructions (e.g., `LOCK` prefix in x86)
- Pthreads
  - Mutex: simple mutual exclusion (“lock”)
  - Condition variable: lock + wait set (wait/signal/broadcast)
  - Semaphore: access to limited resources
    - Not technically part of Pthreads library (just the POSIX standard)
  - Barrier: ensure all threads are at the same point
    - Not present in all implementations (requires `--std=gnu99` on cluster)
- Java threads
  - Synchronized keyword: implicit mutex
  - Monitor: lock on object (wait/notify/notifyAll)

Semaphores

- `sem_`**`init`** `(sem_t*, pshared, int value)`
  - Initialize a semaphore to *value*
- `sem_`**`wait`** `(sem_t*)`
  - If *value* > 0, decrement *value* and return
  - Else, block until signaled
- `sem_`**`post`** `(sem_t*)`
  - Increment *value* and signal a blocked thread
  - Use a loop to signal multiple blocked threads
- `sem_`**`getvalue`** `(sem_t*, int*)`
  - Return current *value*
- `sem_`**`destroy`** `(sem_t*)`
  - Clean up a semaphore

Condition variables

- `pthread_cond_`**`init`** `(pthread_cond_t*, attrs)`
  - Initialize a condition variable
- `pthread_cond_`**`wait`** `(pthread_cond_t*, pthread_mutex_t*)`
  - Release mutex and block until signaled
  - Re-acquires mutex after waking up
  - A variant also exists that times out after a certain period
- `pthread_cond_`**`signal`** `(pthread_cond_t*)`
  - Wake a single blocked thread
- `pthread_cond_`**`broadcast`** `(pthread_cond_t*)`
  - Wake all blocked threads
- `pthread_cond_`**`destroy`** `(pthread_cond_t*)`
  - Clean up a condition variable

Barrier w/ semaphores

Setup:

```
sem_t count_sem;     // initialize to 1
sem_t barrier_sem;   // initialize to 0
volatile int waiting_threads = 0;
```

Threads:

```
sem_wait(&count_sem);
waiting_threads++;
if (waiting_threads == thread_count) {
    waiting_threads = 0;
    sem_post(&count_sem);
    for (int i = 0; i < thread_count-1; i++) {
        sem_post(&barrier_sem);
    }
} else {
    sem_post(&count_sem);
    sem_wait(&barrier_sem);
}
```

Barrier w/ condition variable

Setup:

```
mutex_t count_mut;
cond_t done_waiting;
volatile int waiting_threads = 0;
```

Threads:

```
mutex_lock(&count_mut);
waiting_threads++;
if (waiting_threads == thread_count) {
    waiting_threads = 0;
    cond_broadcast(&done_waiting);
} else {
    cond_wait(&done_waiting, &count_mut);
}
mutex_unlock(&count_mut);
```

Barrier comparison

Semaphores

Setup:

```
sem_t count_sem;      // initialize to 1
sem_t barrier_sem;    // initialize to 0
volatile int waiting_threads = 0;
```

Threads:

```
sem_wait(&count_sem);
waiting_threads++;
if (waiting_threads == thread_count) {
    waiting_threads = 0;
    sem_post(&count_sem);
    for (int i = 0; i < thread_count-1; i++) {
        sem_post(&barrier_sem);
    }
} else {
    sem_post(&count_sem);
    sem_wait(&barrier_sem);
}
```

Condition

Setup:

```
mutex_t count_mut;
cond_t done_waiting;
volatile int waiting_threads = 0;
```

Threads:

```
mutex_lock(&count_mut);
waiting_threads++;
if (waiting_threads == thread_count) {
    waiting_threads = 0;
    cond_broadcast(&done_waiting);
} else {
    cond_wait(&done_waiting, &count_mut);
}
mutex_unlock(&count_mut);
```

Barrier

Setup:

```
barrier_t barrier;    // initialize to nthreads
```

Threads:

```
barrier_wait(&barrier);
```

Condition variables

- Issue: POSIX standard says that `pthread_cond_wait` might experience spurious wakeups
  - Goal: encourage programmers to write correct code
  - Every condition should have an associated boolean predicate
  - The predicate should be true before condition is signaled

    e.g., "`waiting_threads == nthreads`"
  - Waiting thread should re-check predicate after waiting
    - Another thread may have invalidated it!
  - Waiting thread should also check for errors (non-zero return value)
  - Best practice: use a predicate loop and zero check

```
while (!predicate) {
    while (pthread_cond_wait(&cond, &mut) != 0);
}
```

Condition variables

Setup (static):

```
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
volatile boolean status = false;    // protected by mutex
```

Thread 1:

```
pthread_mutex_lock(&mutex);
while (!status) {
    while (pthread_cond_wait(&cond, &mutex) != 0);
}
// at this point, status == true and mutex is locked
```

Thread 2:

```
// do something that triggers status
pthread_mutex_lock(&mutex);
status = true;
pthread_cond_signal(&cond);    // or pthread_cond_broadcast
pthread_mutex_unlock(&mutex);
```

Condition variables

Setup (static):

```
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
volatile boolean status = false;    // protected by mutex
```

initializer macros; can be used if you don't need attributes

C keyword meaning "don't optimize this variable; it could change at any time"

Thread 1:

```
pthread_mutex_lock(&mutex);
while (!status) {
    while (pthread_cond_wait(&cond, &mutex) != 0);
}
// at this point, status == true and mutex is locked
```

check predicate to avoid spurious wakeups

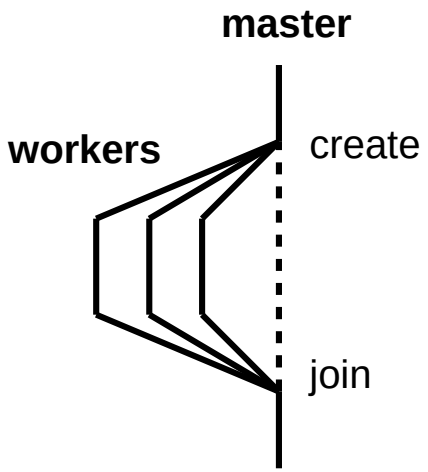
Thread 2:

```
// do something that triggers status
pthread_mutex_lock(&mutex);
status = true;
pthread_cond_signal(&cond);    // or pthread_cond_broadcast
pthread_mutex_unlock(&mutex);
```

set predicate

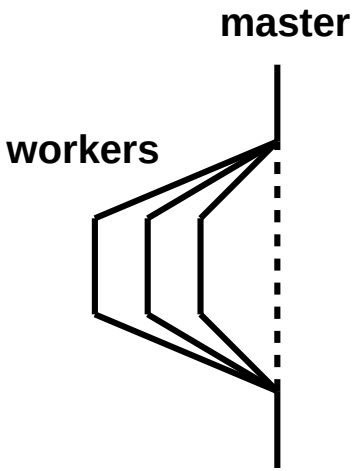
Master/worker model

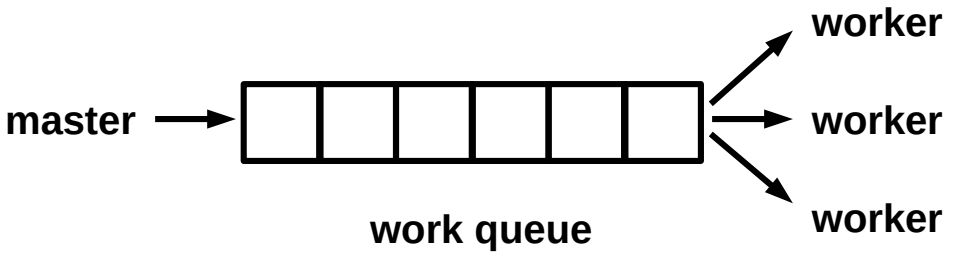
- Common pattern: master/worker threads
  - Original “master” thread creates multiple “worker” threads
  - Each worker thread does a chunk of the work
    - Coordinate via shared global data structure w/ locking
  - Main thread waits for workers, then aggregates results



Thread pool model (P1)

- Minor tweak on master/worker: thread pool model
  - Master thread creates multiple worker threads
  - Work queue tracks chunks of work to be done
    - Producer/consumer: master enqueues, workers dequeue
    - Synchronization required
    - Workers idle while queue is empty





P1 pseudocode

```
master:
```

done = false
initialize work queue and sync variables
spawn workers and wait for all threads to initialize

for each (**action**, **num**) *pair in input:*
    if **action** == *'p':*
        add **num** *to work queue*
        wake an idle worker thread
    else if **action** == *'w':*
        wait **num** *seconds*

wait until all work has been processed

done = true
wake all worker threads and wait for them to terminate

print results, clean up, and exit

```
worker:
```

while not **done***:*
    if queue is not empty:
        extract **num** *from work queue*
        `update(num)`
    else:
        block until signaled

ONE POSSIBILITY; NOT THE ONLY SOLUTION!