CS 470
Spring 2019
Mike Lam, Professor

Multithreading & Pthreads
MIMD system architectures

- Shared memory

- Distributed memory
Multithreading

- A **process** is an instance of a running program
  - Private address space, shared files/sockets
- A **thread** is a single unit of execution in a process
  - Private stack/registers, shared address space
- **Multithreading** libraries provide thread management
  - Spawn/kill capabilities
  - Synchronization mechanisms
  - POSIX threads: **Pthreads**
POSIX threads

- **Pthreads** – POSIX standard interface for threads in C
  - `pthread_create`: spawn a new thread
    - `pthread_t` struct for storing thread info
    - attributes (or NULL)
    - thread work routine (function pointer)
    - thread routine parameter (void*)
  - `pthread_self`: get current thread ID
  - `pthread_exit`: terminate current thread
    - can also terminate implicitly by returning from the thread routine
  - `pthread_join`: wait for another thread to terminate
#include <stdio.h>
#include <pthread.h>

void* work (void* arg)
{
    printf("Hello from new thread!
");
    return NULL;
}

int main ()
{
    printf("Spawning new thread ...
");

    pthread_t peer;
    pthread_create(&peer, NULL, work, NULL);
    pthread_join(peer, NULL);

    printf("Done!
");

    return 0;
}
Shared memory

• Some data is shared in threaded programs
  – Global variables (shared, single static copy)
  – Local variables (multiple copies, one on each stack)
    • Technically still shared if in memory, but harder to access
    • Not shared if cached in register
    • Safer to assume they're private
  – Local static variables (shared, single static copy)
Issues with shared memory

- Nondeterminism
- Data races and deadlock

```assembly
foo:
  irmovq x, %rcx
  irmovq 7, %rax
  mrmovq (%rcx), %rdx
  addq %rax, %rdx
  rmmovq %rdx, (%rcx)
  ret

x:
  .quad 0```

thread1

thread2

foo()
Issues with shared memory

- Nondeterminism
- Data races and deadlock

```assembly
foo:
    irmovq x, %rcx
    irmovq 7, %rax
    mrmovq (%rcx), %rdx
    addq %rax, %rdx
    rmmovq %rdx, (%rcx)
    ret

x:
    .quad 0
```

This interleaving is ok.
Issues with shared memory

- Nondeterminism
- Data races and deadlock

```
foo:
  irmovq x, %rcx
  irmovq 7, %rax
  mrmovq (%rcx), %rdx
  addq %rax, %rdx
  rmmovq %rdx, (%rcx)
  ret

x:
  .quad 0
```

```
PROBLEM!
```
Issues with shared memory

- Nondeterminism
  - Incorrect code can produce “correct” results
  - Test suites cannot guarantee correctness!
- Data races
- Deadlock
- Starvation
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 associated w/ an object (wait/notify/notifyAll)
Mutexes

- `pthread_mutex_init` (pthread_mutex_t*, attrs)
  - Initialize a mutex
- `pthread_mutex_lock` (pthread_mutex_t*)
  - Acquire mutex (block if unavailable)
- `pthread_mutex_unlock` (pthread_mutex_t*)
  - Release mutex
- `pthread_mutex_destroy` (pthread_mutex_t*)
  - Clean up a mutex
Semaphores

- **sem_init** (sem_t*, pshared, int value)
  - Initialize a semaphore to \textit{value}
- **sem_wait** (sem_t*)
  - If \textit{value} > 0, decrement \textit{value} and return
  - Else, block until signaled
- **sem_post** (sem_t*)
  - Increment \textit{value} and signal a blocked thread
  - Use a loop to signal multiple blocked threads
- **sem_getvalue** (sem_t*, int*)
  - Return current \textit{value}
- **sem_destroy** (sem_t*)
  - Clean up a semaphore
Barrier w/ semaphores

Setup:

```c
sem_t count_sem;          // initialize to 1 (access to waiting_threads)
sem_t barrier_sem;        // initialize to 0
volatile int waiting_threads = 0;
```

Threads:

```c
sem_wait(&count_sem);
waiting_threads++;
if (waiting_threads < thread_count) {
    sem_post(&count_sem);
    sem_wait(&barrier_sem);
} else {   // last thread to the barrier
    waiting_threads--;
    sem_post(&count_sem);
    sem_post(&barrier_sem);
    while (waiting_threads-- > 0) {
        sem_post(&barrier_sem);
    }
}

Issue: barrier_sem can’t be re-used later
```
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/ condition variable

Setup:

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

Threads:

```c
mutex_lock(&count_mut);
waiting_threads++;
if (waiting_threads < thread_count) {
    cond_wait(&done_waiting, &count_mut);
} else { // last thread to the barrier
    waiting_threads = 0;
    cond_broadcast(&done_waiting);
}
mutex_unlock(&count_mut);
```
Barrier comparison

Semaphores

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

Threads:
```c
sem_wait(&count_sem);
waiting_threads++;
if (waiting_threads < thread_count) {
    sem_post(&count_sem);
    sem_wait(&barrier_sem);
} else {  // last thread to the barrier
    waiting_threads--;
    sem_post(&count_sem);
    while (waiting_threads-- > 0) {
        sem_post(&barrier_sem);
    }
}
```

Condition

Setup:
```c
mutex_t count_mut;
cond_t done_waiting;
volatile int waiting_threads = 0;
```

Threads:
```c
mutex_lock(&count_mut);
waiting_threads++;
if (waiting_threads < thread_count) {
    cond_wait(&done_waiting, &count_mut);
} else {  // last thread to the barrier
    waiting_threads = 0;
    cond_broadcast(&done_waiting);
}
mutex_unlock(&count_mut);
```

Barrier

Setup:
```c
barrier_t barrier;  // initialize to nthreads
```

Threads:
```c
barrier_wait(&barrier);
```
Condition variables

- Issue: POSIX standard says that `pthread_cond_wait` might experience spurious wakeups from sources other than signal/broadcast calls
  - Goal: optimize runtime and force programmers to write correct code
    ```c
    while (pthread_cond_wait(&cond, &mut) != 0);
    ```

- Issue: non-determinism!
  - Every condition should have an associated boolean predicate
  - The predicate should be true before condition is signaled
    ```c
    e.g., “waiting_threads == nthreads”
    ```
  - Waiting thread should re-check predicate after waking up
    ```c
    • Another thread may have invalidated it in the meantime!
    ```
  - Best practice: use a predicate loop
    ```c
    while (!predicate) {
        pthread_cond_wait(&cond, &mut);
    }
    ```
Condition variables

Setup (static):

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

Thread 1:

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

Thread 2:

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

Thread 1:
```c
pthread_mutex_lock(&mutex);
while (!status) {
    pthread_cond_wait(&cond, &mutex);
}
// at this point, status == true and mutex is locked
```

Thread 2:
```c
// do something that triggers status
pthread_mutex_lock(&mutex);
status = true;
pthread_cond_signal(&cond); // or pthread_cond_broadcast
pthread_mutex_unlock(&mutex);
```
Error checking

- All threading calls might return a non-zero value
  - This generally indicates an error (except for cond_wait)
  - Recovering from errors is not our primary concern now
    - Although we’ll talk a bit about fault tolerance later this semester
  - For now, just write a wrapper to abort on error
  - Example:

```c
void lock(pthread_mutex_t *mut) {
    if (pthread_mutex_lock(mut) != 0) {
        printf("ERROR: could not acquire mutex\n");
        exit(EXIT_FAILURE);
    }
}
```
Common synchronization patterns

- Naturally ("embarrassingly") parallel
  - No synchronization!
- Mutual exclusion
  - Use a lock to prevent simultaneous access
- Producer/consumer
  - Protect common buffer w/ lock
- Readers/writers
  - Multiple lock types
- Master/worker
  - One producer, many consumers
- Dining philosophers
  - Atomic acquisition of multiple locks
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 worker threads

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

done = true
exhaust work queue and wait for workers to finish

print results, clean up, and exit

worker:

while not done and queue is not empty:
    if queue is not empty:
        extract num from work queue
        update(num)
    else:
        become idle until awakened

NOT COMPLETE,
AND NOT THE
ONLY SOLUTION!
Synchronization granularity

- **Granularity**: level at which a structure is locked
  - Whole structure vs. individual pieces
  - If individual pieces, which pieces?
  - Simple locks vs. read/write locks
  - Tradeoff: coarse vs. fine-grained locks

---

<table>
<thead>
<tr>
<th>Implementation</th>
<th>Number of Threads</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
</tr>
<tr>
<td>Read-Write Locks</td>
<td>0.213</td>
</tr>
<tr>
<td>One Mutex for Entire List</td>
<td>0.211</td>
</tr>
<tr>
<td>One Mutex per Node</td>
<td>1.680</td>
</tr>
</tbody>
</table>

**Table 4.3** Linked List Times: 1000 Initial Keys, 100,000 ops, 99.9% Member, 0.05% Insert, 0.05% Delete

---

<table>
<thead>
<tr>
<th>Implementation</th>
<th>Number of Threads</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
</tr>
<tr>
<td>Read-Write Locks</td>
<td>2.48</td>
</tr>
<tr>
<td>One Mutex for Entire List</td>
<td>2.50</td>
</tr>
<tr>
<td>One Mutex per Node</td>
<td>12.00</td>
</tr>
</tbody>
</table>

**Table 4.4** Linked List Times: 1000 Initial Keys, 100,000 ops, 80% Member, 10% Insert, 10% Delete
Locality

- **Temporal locality**: frequently-accessed items will continue to be accessed in the future
  - Theme: *repetition is common*
- **Spatial locality**: nearby addresses are more likely to be accessed soon
  - Theme: *sequential access is common*
- **Why do we care?**
  - *Shared-memory programs with good locality run faster than programs with poor locality*
Caching effects

• Caching
  – Keep frequently-used stuff in faster memory

• Cache line
  – Single unit of cached data

• Cache hits/misses
  – Was data in cache? (if so, hit; if not, miss)

• Cache invalidation
  – Writes to one cache can render another cache out-of-date

• False sharing
  – Unnecessary cache invalidation
Multithreading summary

• Shared memory parallelism has a lot of benefits
  - Low overhead for thread creation/switching
  - Uniform memory access times (*symmetric* multiprocessing)

• It also has significant issues
  - Limited scaling (# of cores)
  - Requires explicit thread management
  - Requires explicit synchronization (*HARD!*)
  - Caching problems can be difficult to diagnose

• Core design tradeoff: synchronization *granularity*
  - Higher granularity: simpler but slower
  - Lower granularity: more complex but faster