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
  - Must `#include <pthread.h>` and link using `-lpthread`
  - `pthread_create`: spawn a new thread
    - `pthread_t opaque struct for storing thread info`
    - attributes (or NULL)
    - thread work routine (function pointer)
    - work routine parameter (void*)
  - `pthread_self`: get current thread ID
  - `pthread_exit`: terminate current thread current thread
    - can also terminate implicitly by returning from the thread routine
  - `pthread_join`: wait for another thread to terminate
```c
#include <stdio.h>
#include <pthread.h>

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

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

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

    printf("Done!\n");

    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)
• Also shared:
  – Heap_allocated memory (if the threads have pointers)
  – Open files, sockets, pipes, etc.
int x = 0;

void foo()
{
    x += 7;
}

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

x:
  .quad 0
Example (from CS 261)

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

x:
  .quad 0

This interleaving is ok.
Example (from CS 261)

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

x:
  .quad 0

thread1
  foo()
  irmovq x, %rcx
  irmovq 7, %rax
  mrmovq (%rcx), %rdx
  addq %rax, %rdx
  rmmovq %rdx, (%rcx)
  ret

thread2
  foo()
  irmovq x, %rcx
  irmovq 7, %rax
  mrmovq (%rcx), %rdx
  addq %rax, %rdx
  rmmovq %rdx, (%rcx)
  ret

PROBLEM!
Issues with shared memory

- **Nondeterminism**
  - Incorrect code can produce “correct” results
  - Test suites cannot guarantee correctness!
- **Data race**
  - Multiple threads attempting to access a shared resource simultaneously
  - Different interleavings may produce different outputs
- **Deadlock**
  - All threads waiting such that none can make progress
- **Starvation**
  - A particular thread never gets access to a shared resource
Tools for detecting thread issues

- **Helgrind**: Valgrind-based thread issue detector
  - Available on the cluster! (use it for P1!)
  - Usage: `valgrind --tool=helgrind <YOUR PROGRAM>`
  - Detects data races, deadlock, and other Pthread misuses
  - Helgrind documentation

- **Other tools**:
  - Intel Inspector
  - Arm DDT
  - Google ASan
```c
#include <stdio.h>
#include <pthread.h>

int count = 0;

int increment(int x) {
    return x + 1;
}

void* work (void* arg) {
    for (int i = 0; i < 10000; i++) {
        count = increment(count);
    }
    return NULL;
}

int main () {
    pthread_t peer;
    pthread_create(&peer, NULL, work, NULL);

    for (int i = 0; i < 10000; i++) {
        count = increment(count);
    }

    pthread_join(peer, NULL);
    printf("count = %d\n", count);
    return 0;
}
```

```c
#include <stdio.h>
#include <pthread.h>

int count = 0;

int increment(int x) {
    return x + 1;
}

void* work (void* arg) {
    for (int i = 0; i < 10000; i++) {
        pthread_mutex_lock(&count_mut);
        count = increment(count);
        pthread_mutex_unlock(&count_mut);
    }
    return NULL;
}

int main () {
    pthread_t peer;
    pthread_create(&peer, NULL, work, NULL);

    for (int i = 0; i < 10000; i++) {
        pthread_mutex_lock(&count_mut);
        count = increment(count);
        pthread_mutex_unlock(&count_mut);
    }

    pthread_join(peer, NULL);
    printf("count = %d\n", count);
    return 0;
}
```
Synchronization mechanisms

- Busy-waiting *(wasteful!)*
- Atomic instructions (e.g., \texttt{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 \texttt{--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_INITIALIZER macro for defaults
- `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
Barrier w/ mutex

Setup:

```c
int counter = 0;               // number of threads waiting
int thread_count;              // number of total threads
pthread_mutex_t barrier_mutex;
```

Threads:

```c
pthread_mutex_lock(&barrier_mutex);
counter++;
pthread_mutex_unlock(&barrier_mutex);
while (counter < thread_count);  // busy wait
```

**Issue:** wasted CPU cycles!
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
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);
    while (waiting_threads--> 0) {
        sem_post(&barrier_sem);
    }
}
```

**Issue:** barrier_sem can’t be re-used later (race condition if one thread hits the second barrier while another thread is still waiting to be posted on the first)
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 (should be holding the mutex)
- **pthread_cond_broadcast** (pthread_cond_t*)
  - Wake all blocked threads (should be holding the mutex)
- **pthread_cond_destroy** (pthread_cond_t*)
  - Clean up a condition variable
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) {
    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
    
    ```
    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
    
    ```
    e.g., “task_queue_size > 0”
    ```
  - Waiting thread should **re-check predicate** after waking up
    - Another thread may have invalidated it in the meantime!
  - Best practice: use a predicate loop
    ```
    pthread_mutex_lock(&mut);
    while (!predicate) {
        pthread_cond_wait(&cond, &mut);
    }
    pthread_mutex_unlock(&mut);
    ```
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) {
     pthread_cond_wait(&cond, &mutex);
 }
 // 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):
```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; // set predicate
- pthread_cond_signal(&cond); // or pthread_cond_broadcast
pthread_mutex_unlock(&mutex);
```
Error checking

- All Pthreads 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
- Supervisor/worker
  - One producer, many consumers
- Dining philosophers
  - Atomic acquisition of multiple locks
Supervisor/worker model

- Common pattern: supervisor/worker threads
  - Original “supervisor” thread creates multiple “worker” threads
  - Each worker thread does a chunk of the work
    - Coordinate via shared global data structure w/ locking
  - Main/supervisor thread waits for workers, then aggregates results
Thread pool model (P1)

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

\[ \text{done} = \text{false} \]
initialize work queue and sync variables
spawn worker threads

\[ \text{for each (action, num) pair in input:} \]

\[ \text{if action == 'p':} \]
\[ \begin{align*}
\text{add num to work queue} \\
\text{wake an idle worker thread}
\end{align*} \]

\[ \text{else if action == 'w':} \]
\[ \begin{align*}
\text{wait num seconds}
\end{align*} \]

\[ \text{done} = \text{true} \]
wake any idle workers
wait for all workers to finish

print results, clean up, and exit

worker:

\[ \text{while not done or queue is not empty:} \]

\[ \text{if queue is not empty:} \]
\[ \begin{align*}
\text{extract num from work queue} \\
\text{update(num)}
\end{align*} \]

\[ \text{else:} \]
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 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>
<th>1</th>
<th>2</th>
<th>4</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read-Write Locks</td>
<td></td>
<td>0.213</td>
<td>0.123</td>
<td>0.098</td>
<td>0.115</td>
</tr>
<tr>
<td>One Mutex for Entire List</td>
<td></td>
<td>0.211</td>
<td>0.450</td>
<td>0.385</td>
<td>0.457</td>
</tr>
<tr>
<td>One Mutex per Node</td>
<td></td>
<td>1.680</td>
<td>5.700</td>
<td>3.450</td>
<td>2.700</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Implementation</th>
<th>Number of Threads</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read-Write Locks</td>
<td></td>
<td>2.48</td>
<td>4.97</td>
<td>4.69</td>
<td>4.71</td>
</tr>
<tr>
<td>One Mutex for Entire List</td>
<td></td>
<td>2.50</td>
<td>5.13</td>
<td>5.04</td>
<td>5.11</td>
</tr>
<tr>
<td>One Mutex per Node</td>
<td></td>
<td>12.00</td>
<td>29.60</td>
<td>17.00</td>
<td>12.00</td>
</tr>
</tbody>
</table>
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