Table of Contents

1 Controlling Access and Synchronization
   • Critical Sections
   • Busy-Waiting
   • Pthreads - Mutexes

2 Next Time
Today we’ll talk about coordinating and synchronizing Pthreads.

Main thread forks and joins two threads
Critical Sections

- Matrix-vector multiplication was straightforward to code:
  - Shared-memory locations were accessed in a simple manner.
  - After initialization, all of the variables but $y$ are read only.
  - After initialization, shared variables not changed.
- Threads make changes to $y$: but elements are owned by a thread.
- There are *no* attempts by multiple threads to modify the *same* element.
- What happens if this is not the case? What happens when multiple threads update a single memory location?
Estimating $\pi$: Serial Code

$$\pi = 4 \left( 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \cdots + (-1)^n \frac{1}{2n+1} + \cdots \right)$$

double factor = 1.0;
double sum = 0.0;
for (i = 0; i < n; i++, factor = -factor) {
    sum += factor/(2*i+1);
}
pi = 4.0*sum;
Thread function for calculating $\pi$ (Pacheco)

```c
void* Thread_sum(void* rank) {
    long my_rank = (long) rank;
    double factor;
    long long i;
    long long my_n = n/thread_count;
    long long my_first_i = my_n*my_rank;
    long long my_last_i = my_first_i + my_n;

    if (my_first_i % 2 == 0) /* my_first_i is even */
        factor = 1.0;
    else /* my_first_i is odd */
        factor = -1.0;

    for (i = my_first_i; i < my_last_i; i++, factor = -factor) {
        sum += factor/(2*i+1);
    }

    return NULL;
} /* Thread_sum */
Using a dual core processor

<table>
<thead>
<tr>
<th></th>
<th>$10^5$</th>
<th>$10^6$</th>
<th>$10^7$</th>
<th>$10^8$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\pi$</td>
<td>3.14159</td>
<td>3.141593</td>
<td>3.1415927</td>
<td>3.14159265</td>
</tr>
<tr>
<td>1 Thread</td>
<td>3.14158</td>
<td>3.141592</td>
<td>3.1415926</td>
<td>3.14159264</td>
</tr>
<tr>
<td>2 Threads</td>
<td>3.14158</td>
<td>3.141480</td>
<td>3.1413692</td>
<td>3.14164686</td>
</tr>
</tbody>
</table>

Note that as we increase $n$, the estimate with one thread gets better and better.

Program run with 2 threads, dual core processor
/* File: pth_pi.c */
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <pthread.h>

const int MAX_THREADS = 1024;
long thread_count;
long long n;
double sum;


void* Thread_sum(void* rank);
double Serial_pi(long long n);

int main(int argc, char* argv[]) {
    long thread; /* Use long in case of a 64-bit system */
    pthread_t* thread_handles;

    /* Get number of threads from command line */
    Get_args(argc, argv);

    thread_handles = (pthread_t*) malloc (thread_count*sizeof(pthread_t));
    sum = 0.0;
}
for (thread = 0; thread < thread_count; thread++)
    pthread_create(&thread_handles[thread], NULL,
                   Thread_sum, (void*)thread);

for (thread = 0; thread < thread_count; thread++)
    pthread_join(thread_handles[thread], NULL);

sum = 4.0*sum;
printf("With n = %lld terms,\n", n);
printf(" Our estimate of pi = %.15f\n", sum);
sum = Serial_pi(n);
printf(" Single thread est = %.15f\n", sum);
printf(" pi = %.15f\n", 4.0*atan(1.0));

free(thread_handles);
return 0;
} /* main */
Controlling Access and Synchronization

Critical Sections

POSIX Threads: Pacheco `pthd_pi.c` (3)

```c

* Function:  Thread_sum
* Purpose:   Add in the terms computed by the thread running this
* In arg:   rank
* Ret val:  ignored
* Globals in:  n, thread_count
* Global in/out:  sum
*

void* Thread_sum(void* rank) {
    long my_rank = (long) rank;
    double factor;
    long long i;
    long long my_n = n/thread_count;
    long long my_first_i = my_n*my_rank;
    long long my_last_i = my_first_i + my_n;

    if (my_first_i % 2 == 0)
        factor = 1.0;
    else
        factor = -1.0;

    for (i = my_first_i; i < my_last_i; i++, factor = -factor) {
        sum += factor/(2*i+1);
    }

    return NULL;
} /* Thread_sum */
```
POSIX Threads: Pacheco `pthd_pi.c` (4)

```c
/*
 * Function: Serial_pi
 * Purpose: Estimate pi using 1 thread
 * In arg: n
 * Return val: Estimate of pi using n terms of Maclaurin series
 */
double Serial_pi(long long n) {
    double sum = 0.0;
    long long i;
    double factor = 1.0;

    for (i = 0; i < n; i++, factor = -factor) {
        sum += factor/(2*i+1);
    }
    return 4.0*sum;
} /* Serial_pi */
```
Controlling Access and Synchronization

Critical Sections

[tuckoo] mthomas% ./pth_pi 1 10000000
With n = 10000000 terms,
  Our estimate of pi = 3.141592553589792
  Single thread est = 3.141592553589792
  pi = 3.141592653589793
[tuckoo] mthomas% ./pth_pi 2 10000000
With n = 10000000 terms,
  Our estimate of pi = 3.142023243006218
  Single thread est = 3.141592553589792
  pi = 3.141592653589793
[tuckoo] mthomas% ./pth_pi 8 10000000
With n = 10000000 terms,
  Our estimate of pi = 3.139519366596481
  Single thread est = 3.141592553589792
  pi = 3.141592653589793
[tuckoo] mthomas% ./pth_pi 32 10000000
With n = 10000000 terms,
  Our estimate of pi = 3.141841506956732
  Single thread est = 3.141592553589792
  pi = 3.141592653589793
[tuckoo] mthomas% ./pth_pi 64 10000000
With n = 10000000 terms,
  Our estimate of pi = 3.141815732350456
  Single thread est = 3.141592553589792
  pi = 3.141592653589793
[tuckoo] mthomas% ./pth_pi 128 10000000
With n = 10000000 terms,
  Our estimate of pi = 3.141396902178384
  Single thread est = 3.141592553589792
  pi = 3.141592653589793
Possible race condition

<table>
<thead>
<tr>
<th>Time</th>
<th>Thread 0</th>
<th>Thread 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Started by main thread</td>
<td>Started by main thread</td>
</tr>
<tr>
<td>2</td>
<td>Call Compute ()</td>
<td>Call Compute ()</td>
</tr>
<tr>
<td>3</td>
<td>Assign y = 1</td>
<td>Assign y = 2</td>
</tr>
<tr>
<td>4</td>
<td>Put x=0 and y=1 into registers</td>
<td>Put x=0 and y=2 into registers</td>
</tr>
<tr>
<td>5</td>
<td>Add 0 and 1</td>
<td>Add 0 and 2</td>
</tr>
<tr>
<td>6</td>
<td>Store 1 in memory location x</td>
<td>Store 2 in memory location x</td>
</tr>
</tbody>
</table>

Fundamental problem with Pthreads: when multiple threads try to access/update the same resource, the result can be unpredictable.
Busy-Waiting

- A thread repeatedly tests a condition
- Beware of optimizing compilers:

```c
y = Compute(my_rank);
while (flag != my_rank);
x = x + y;
flag++;
```

- Thread 1 cannot enter critical section until Thread 0 has finished.
  Warning: compilers can optimize code and rearrange order of code affecting busy-wait cycle.
Pthreads: global sum with busy-waiting

```c
/*
 * Function: Thread_sum
 * Purpose: Add in the terms computed by the thread running this
 * In arg: rank
 * Ret val: ignored
 * Globals in: n, thread_count
 * Global in/out: sum
 */
void* Thread_sum(void* rank) {
    long my_rank = (long) rank;
double factor;
    long long i;
    long long my_n = n/thread_count;
    long long my_first_i = my_n*my_rank;
    long long my_last_i = my_first_i + my_n;

    if (my_first_i % 2 == 0)
        factor = 1.0;
    else
        factor = -1.0;

    for (i = my_first_i; i < my_last_i; i++, factor = -factor) {
        while (flag != my_rank);
        sum += factor/(2*i+1);
        flag = (flag+1) % thread_count;
    }
    return NULL;
} /* Thread_sum */
```

Thread 1 spins until Thread 0 finishes - could waste resources. Add in logic for last thread to reset flag.
Controlling Access and Synchronization

Busy-Waiting

[tuckoo] mthomas% ./pth_pi_busy1 8 100000
With n = 100000 terms,
  Multi-threaded estimate of pi  = 3.141582653589717
  Elapsed time = 1.306486e-02 seconds
  Single-threaded estimate of pi = 3.141582653589720
  Elapsed time = 4.179478e-04 seconds
  Math library estimate of pi    = 3.141592653589793

[tuckoo] mthomas% ./pth_pi_busy1 8 10000000
With n = 10000000 terms,
  Multi-threaded estimate of pi  = 3.141592553589788
  Elapsed time = 9.265280e-01 seconds
  Single-threaded estimate of pi = 3.141592553589792
  Elapsed time = 4.049492e-02 seconds
  Math library estimate of pi    = 3.141592653589793

Note: Serial version is faster than threaded version!
Pthreads: define locate sum, and update global sum in critical section after loop

```
* Function: Thread_sum
* Purpose: Add in the terms computed by the thread running this
* In arg: rank
* Ret val: ignored
* Globals in: n, thread_count
* Global in/out: sum
*/
void* Thread_sum(void* rank) {
    long my_rank = (long) rank;
    double factor;
    long long i;
    long long my_n = n/thread_count;
    long long my_first_i = my_n*my_rank;
    long long my_last_i = my_first_i + my_n;

    if (my_first_i % 2 == 0)
        factor = 1.0;
    else
        factor = -1.0;

    for (i = my_first_i; i < my_last_i; i++, factor = -factor)
        my_sum += factor/(2*i+1);

    while (flag != my_rank);
    sum += my_sum;
    flag = (flag+1) % thread_count;

    return NULL;
} /* Thread_sum */
```
Output after using local sum var; moving critical section to after loop.

[mthomas@tuckoo ch4]$ ./pth_pi_busy2 8 1000000
With n = 1000000 terms,
   Multi-threaded estimate of pi  = 3.141591653589728
   Elapsed time = 1.039195e-02 seconds
   Single-threaded estimate of pi = 3.141591653589774
   Elapsed time = 1.185608e-02 seconds
   Math library estimate of pi   = 3.141592653589793

[mthomas@tuckoo ch4]$

[mthomas@tuckoo ch4]$ ./pth_pi_busy2 8 10000000
With n = 10000000 terms,
   Multi-threaded estimate of pi  = 3.141592553589832
   Elapsed time = 3.278208e-02 seconds
   Single-threaded estimate of pi = 3.141592553589792
   Elapsed time = 1.130030e-01 seconds
   Math library estimate of pi   = 3.141592653589793

Note: Serial and threaded timings are closer
Mutexes

- A thread that is busy-waiting may continually use the CPU accomplishing nothing.
- Mutex (mutual exclusion) is a special type of variable that can be used to restrict access to a critical section to a single thread at a time.
- Used to guarantee that one thread "excluded" all other threads while it executes the critical section.
- The Pthreads standard includes a special type for mutexes: `pthread_mutex_t`

Initialization:

```c
int pthread_mutex_init (
    pthread_mutex_t *mutex_p, /* out */
    pthread_mutexattr_t *attr_p, /* out */
);```
When a thread is finished executing the code in a critical section, it should call:

```c
int pthread_mutex_unlock (pthread_mutex_t*);
```

calling thread waits until no other thread is in critical section

program pseudocode:

- declare global mutex variable
- main thread init variable
- call `pthread_mutex_lock`
  - do critical section work ]
- call `pthread_mutex_unlock`

this is a **blocking** call: it locks and unlocks access to critical section.

repeats until all threads are done.
main defines global mutex variable, inits and destroys

```c
pthread_mutex_t mutex;  /* declare global mutex variable */

int main(int argc, char* argv[]) {
    long thread;  /* Use long in case of a 64-bit system */
    pthread_t* thread_handles;
    double start, finish, elapsed;

    /* Get number of threads from command line */
    Get_args(argc, argv);
    thread_handles = (pthread_t*) malloc (thread_count*sizeof(pthread_t));

    pthread_mutex_init(&mutex, NULL);
    sum = 0.0;
    GET_TIME(start);
    for (thread = 0; thread < thread_count; thread++)
        pthread_create(&thread_handles[thread], NULL, Thread_sum, (void*)thread);
    for (thread = 0; thread < thread_count; thread++)
        pthread_join(thread_handles[thread], NULL);
    GET_TIME(finish);
    elapsed = finish - start;
    sum = 4.0*sum;
    GET_TIME(start); sum = Serial_pi(n); GET_TIME(finish);
    elapsed = finish - start;

    pthread_mutex_destroy(&mutex);
    free(thread_handles);
    return 0;  }  /* end main */
```
function computes local my_sum, then uses mutex_lock for control

/*------------------------------------------------------------------*/

void* Thread_sum(void* rank) {
    long my_rank = (long) rank;
    double factor;
    long long i;
    long long my_n = n/thread_count;
    long long my_first_i = my_n*my_rank;
    long long my_last_i = my_first_i + my_n;
    double my_sum = 0.0;

    if (my_first_i % 2 == 0)
        factor = 1.0;
    else
        factor = -1.0;

    for (i = my_first_i; i < my_last_i; i++, factor = -factor) {
        my_sum += factor/(2*i+1);
    }
    pthread_mutex_lock(&mutex);
    sum += my_sum;
    pthread_mutex_unlock(&mutex);
    return NULL;
} /* Thread_sum */
Controlling Access and Synchronization

Pthreads - Mutexes

<table>
<thead>
<tr>
<th>Threads</th>
<th>Busy-Wait</th>
<th>Mutex</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2.90</td>
<td>2.90</td>
</tr>
<tr>
<td>2</td>
<td>1.45</td>
<td>1.45</td>
</tr>
<tr>
<td>4</td>
<td>0.73</td>
<td>0.73</td>
</tr>
<tr>
<td>8</td>
<td>0.38</td>
<td>0.38</td>
</tr>
<tr>
<td>16</td>
<td>0.50</td>
<td>0.38</td>
</tr>
<tr>
<td>32</td>
<td>0.80</td>
<td>0.40</td>
</tr>
<tr>
<td>64</td>
<td>3.56</td>
<td>0.38</td>
</tr>
</tbody>
</table>

\[ \frac{T_{\text{serial}}}{T_{\text{parallel}}} \approx \text{thread\_count} \]

Run-times (in seconds) of π programs using \( n = 108 \) terms on a system with two four-core processors.
Run-times (in seconds) for calculation of $\pi$ using $n = 10^8$ points.
### Possible sequence of events with busy-waiting and more threads than cores.

<table>
<thead>
<tr>
<th>Time</th>
<th>flag</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>crit sect</td>
<td>busy wait</td>
<td>susp</td>
<td>susp</td>
<td>susp</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>terminate</td>
<td>crit sect</td>
<td>susp</td>
<td>busy wait</td>
<td>susp</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>—</td>
<td>terminate</td>
<td>susp</td>
<td>busy wait</td>
<td>busy wait</td>
</tr>
<tr>
<td>:</td>
<td>:</td>
<td>:</td>
<td>:</td>
<td>:</td>
<td>:</td>
<td>:</td>
</tr>
<tr>
<td>?</td>
<td>2</td>
<td>—</td>
<td>—</td>
<td>crit sect</td>
<td>susp</td>
<td>busy wait</td>
</tr>
</tbody>
</table>
Next Time

- Next class: 10/23/14
- HW #3 Due: 10/28/14
- Quiz 2 (MPI) and Quiz 3 (Pthreads) will be held on 10/30/14.
- Quiz 2: two parts:
  - Part 1: take home assignment on 10/30/14, due 11/04/14
    [was 10/23/14, due 10/28]
    (50% of the grade)
  - Part 2: in class quiz on TBA 10/30/14
    [was 10/28]
    (50% of the grade)