# Table of Contents

1. Today
2. Shared Memory Programming with PThreads
   - Shared Memory Systems
   - Threads and Processes
   - Basic Pthreads Program: Hello World
   - POSIX Threads API
   - Matrix-Vector Multiplication with Pthreads
3. Controlling Access and Synchronization
   - Critical Sections
   - Busy-Waiting
4. Next Time
Today:
- Introduction to Pthreads
- We’ll cover Pthreads in Lectures L17-L20

Source code for HW4 (MPI Communications) has been put into the tuckoo in /COMP605/hw4

Source code for MPI Communication Topology examples have been put into the class directory on tuckoo in /COMP605/comm.src
Shared Memory System

Best candidates:

- can be organized into discrete, independent tasks which can execute concurrently
- routines can be interchanged, interleaved and/or overlapped in real time
Shared Memory System
What is a Process?

- A process is an instance of a running (or suspended) program.
- Can be "muti-threaded," created by OS, requires a fair amount of "overhead"
- Process ID, process group ID, user ID, and group ID, Environment
- program instructions, registers, stack, heap, signals, libraries
- working directory, file descriptors
- Inter-process communication tools (such as message queues, pipes, semaphores, or shared memory).

Ref: http://www.bottomupcs.com/elements_of_a_process.html
What is a Thread?

- Threads are analogous to a light-weight process.
- Shared memory program: single process may have multiple threads of control.
- Independent stream of instructions, run inside processes
- Programs/procedures: runs independently from main program (e.g. multiple functions running concurrently)
- Example: main program (a.out) that contains a number of procedures that can be scheduled to run simultaneously and/or independently

Thread models:
- **Manager/worker**: a single thread, manager assigns work to other threads (workers).
- **Pipeline**: task is broken into series of subops; each handled in series, but concurrently by another thread.
- **Peer**: After the main thread (manager) creates other threads, it participates in the work.
Shared Memory Programming with PThreads

Threads and Processes

UNIX PROCESS

THREADS WITHIN A UNIX PROCESS
PORTABLE OPERATING SYSTEM INTERFACE

IEEE’s POSIX Threads Model (Pthreads):
- programming models for threads in a UNIX platform
- Pthreads are included in the international standards ISO/IEC9945-1

A standard for Unix-like operating systems.
A library that can be linked with C programs.
Specifies an application programming interface (API) for multi-threaded programming.

The Pthreads API is only available on POSIXR systems such as: Linux, MacOS X, Solaris, HPUX,
Phreads: Hello World

/* File: pth_hello.c
 * Purpose:
 * Illustrate basic use of pthreads: create some threads, each of which prints a message.
 * Input: none
 * Output: message from each thread
 * Compile: gcc -g -Wall -o pth_hello pth_hello.c -lpthread
 * Usage: ./pth_hello <thread_count>
 * IPP: Section 4.2 (p. 153 and ff.)
 */
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
const int MAX_THDS = 64;  
/* Global variable: accessible to all threads */
int thread_count;

void Usage(char* prog_name);
void *Hello(void* rank); /* Thread function */

int main(int argc, char* argv[]) {
    /* Use long in case of a 64-bit system */
    long thread;
    pthread_t* thread_handles;
    /* Get number of threads from command line */
    if (argc != 2) Usage(argv[0]);
    thread_count = strtol(argv[1], NULL, 10);
    if (thread_count <= 0 || thread_count > MAX_THDS)
        Usage(argv[0]);
    thread_handles = malloc (thread_count*sizeof(pthread_t));
    for (thread = 0; thread < thread_count; thread++)
        pthread_create(&thread_handles[thread], NULL, Hello, (void*) thread);
    printf("Hello from the main thread\n");
    for (thread = 0; thread < thread_count; thread++)
        pthread_join(thread_handles[thread], NULL);
    free(thread_handles);
    return 0;
} /* main */

void *Hello(void* rank) {
    /* Use long in case of a 64-bit system */
    long my_rank = (long) rank;
    printf("Hello from thread %ld of %d\n", my_rank, thread_count);
    return NULL;
} /* Hello */

void Usage(char* prog_name) {
    fprintf(stderr, "usage: %s <number of threads>\n", prog_name);
    fprintf(stderr, "0 < number of threads <= %d\n", MAX_THREADS);
    exit(0);
} /* Usage */
Compiling and running a Pthreads program

- Pthreads is a standard C library
- Compile like standard C code:

  [gidget] % gcc -g -Wall -o pth_hello pth_hello.c -lpthread

[gidget] % ./pth_hello 1
Hello from the main thread
Hello from thread 0 of 1

[gidget:dev/ipp.ch4/hello] mthomas% ./pth_hello 4
Hello from thread 0 of 4
Hello from thread 2 of 4
Hello from thread 1 of 4
Hello from the main thread
Hello from thread 3 of 4
Running a Pthreads program on tuckoo

[mthomas@tuckoo ch4] gcc -g -Wall -o pth_hello pth_hello.c -lpthread

[mthomas@tuckoo ch4] ./pth_hello 8

Hello from thread 0 of 8
Hello from thread 1 of 8
Hello from thread 2 of 8
Hello from thread 3 of 8
Hello from thread 4 of 8
Hello from thread 5 of 8
Hello from thread 6 of 8
Hello from the main thread
Hello from thread 7 of 8
Warning about global variables

- All threads have access to the same global, shared memory
- Threads also have their own private data
- Limit use of global variables to situations where they are really needed:
  - Shared variables.
- Programmers are responsible for synchronizing access (protecting) globally shared data.
  - Can introduce subtle and confusing bugs
POSIX Threads API: Four Main Groups

- **Thread management:** Routines that work directly on threads - creating, detaching, joining, etc.
- **Mutexes:** Routines that deal with synchronization, called a "mutex", which is an abbreviation for "mutual exclusion".
- **Condition variables:** Routines that address communications between threads that share a mutex. Includes functions to create, destroy, wait and signal based upon specified variable values.
- **Synchronization:** Routines that manage read/write locks and barriers.
Processes in MPI are usually started by a script.

In Pthreads the threads are started by the program executable.

```c
int pthread_create ( pthread_t* thread_p /*out*/ ,
const pthread_attr_t* attr_p /*in*/ ,
void* (*start_routine) ( void ) /*in*/ ,
void* arg_p /*in*/ ,
);
```
pthread_t objects

- Opaque
- The actual data that they store is system-specific.
- Their data members aren’t directly accessible to user code.
- However, the Pthreads standard guarantees that a pthread_t object does store enough information to uniquely identify the thread with which it’s associated.
A closer look (1)

```c
int pthread_create ( 
    pthread_t* thread_p /* out */ ,
    const pthread_attr_t* attr_p /* in */ ,
    void* (*start_routine) ( void ) /* in */ ,
    void* arg_p /* in */ ) ;
```

We won’t be using, so we just pass NULL.

Allocate before calling.
A closer look (2)

```c
int pthread_create ( 
    pthread_t* thread_p /* out */ ,
    const pthread_attr_t* attr_p /* in */ ,
    void* (*start_routine ) ( void ) /* in */ ,
    void* arg_p /* in */ ) ;
```

- Pointer to the argument that should be passed to the function `start_routine`.
- The function that the thread is to run.
Function started by pthread_create

- Prototype:
  ```c
  void* thread_function ( void* args_p ) ;
  ```

- `void*` can be cast to any pointer type in C.

- So `args_p` can point to a list containing one or more values needed by `thread_function`.

- Similarly, the return value of `thread_function` can point to a list of one or more values.
Main thread forks and joins two threads
Stopping the Threads

- We call the function `pthread_join` once for each thread.
- A single call to `pthread_join` will wait for the thread associated with the `pthread_t` object to complete.
Shared Memory Programming with PThreads

POSIX Threads API

Source: https://computing.llnl.gov/tutorials/pthreads/
**Matrix-Vector Multiplication with Pthreads**

**Solve:** \( A x = y \)

\[
\begin{bmatrix}
  a_{0,0} & a_{0,j} & \cdots & a_{0,n-1} \\
  \vdots & \ddots & \ddots & \vdots \\
  a_{i,0} & a_{i,j} & \cdots & a_{i,n-1} \\
  \vdots & \ddots & \ddots & \vdots \\
  a_{m-1,0} & a_{m-1,j} & \cdots & a_{m-1,n-1}
\end{bmatrix} \begin{bmatrix} x_0 \\ \vdots \\ x_i \\ \vdots \\ x_{n-1} \end{bmatrix} = \begin{bmatrix} y_0 \\ \vdots \\ y_i = a_{i,0}x_0 + \cdots + a_{i,j}x_i + \cdots + a_{i,n-1}x_{n-1} - 1 \\ \vdots \\ y_{m-1} \end{bmatrix}
\]
Serial Pseudo-code

/* For each row of A */
for (i = 0; i < m; i++) {
    y[i] = 0.0;
    /* For each element of the row and each element of x */
    for (j = 0; j < n; j++)
        y[i] += A[i][j] * x[j];
}

\[ y_i = \sum_{j=0}^{n-1} a_{ij} x_j \]
Using 3 Pthreads, 6 elements

<table>
<thead>
<tr>
<th>Thread</th>
<th>Components of y</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>y[0], y[1]</td>
</tr>
<tr>
<td>1</td>
<td>y[2], y[3]</td>
</tr>
<tr>
<td>2</td>
<td>y[4], y[5]</td>
</tr>
</tbody>
</table>

\[
y[0] = 0.0;
\text{for } (j = 0; j < n; j++)
\]
\[
y[0] += A[0][j] \times x[j];
\]

\[
y[i] = 0.0;
\text{for } (j = 0; j < n; j++)
\]
\[
y[i] += A[i][j] \times x[j];
\]
Pthreads matrix-vector multiplication

/* File:
 *  pth_mat_vect.c
 * *
 * Compile: gcc -g -Wall -o pth_mat_vect pth_mat_vect.c -lpthread
 * Usage:
 *  pth_mat_vect <thread_count>
 * *
 * IPP: Section 4.3 (pp. 159 and ff.). Also Section 4.10 (pp. 191)
 */
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

/* Global variables */
int    thread_count;
int    m, n;
double* A;
double* x;
double* y;

/* Serial functions */
void Usage(char* prog_name);
void Read_matrix(char* prompt, double A[], int m, int n);
void Read_vector(char* prompt, double x[], int n);
void Print_matrix(char* title, double A[], int m, int n);
void Print_vector(char* title, double y[], double m);

/* Parallel function */
void *Pth_mat_vect(void* rank);
Pthreads matrix-vector multiplication

int main(int argc, char* argv[]) {
    long thread;
    pthread_t* thread_handles;

    if (argc != 2) Usage(argv[0]);
    thread_count = atoi(argv[1]);
    thread_handles = malloc(thread_count*sizeof(pthread_t));

    printf("Enter m and n\n"); scanf("%d%d", &m, &n);

    A = malloc(m*n*sizeof(double));
    x = malloc(n*sizeof(double));
    y = malloc(m*sizeof(double));

    Read_matrix("Enter the matrix", A, m, n); Print_matrix("We read", A, m, n);
    Read_vector("Enter the vector", x, n); Print_vector("We read", x, n);

    for (thread = 0; thread < thread_count; thread++)
        pthread_create(&thread_handles[thread], NULL, Pth_mat_vect, (void*) thread);

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

    Print_vector("The product is", y, m);

    free(A); free(x); free(y);
    return 0;
Pthreads matrix-vector multiplication

/**************************************************************************
 * Function: Usage
 * Purpose: print a message showing what the command line should be, and terminate
 * In arg : prog_name
 */
void Usage (char* prog_name) {
fprintf(stderr, "usage: %s <thread_count>\n", prog_name);
exit(0);
} /* Usage */

/**************************************************************************
 * Function: Read_matrix
 * Purpose: Read in the matrix
 * In args: prompt, m, n
 * Out arg: A
 */
void Read_matrix(char* prompt, double A[], int m, int n) {
    int i, j;

    printf("%s\n", prompt);
    for (i = 0; i < m; i++)
        for (j = 0; j < n; j++)
            scanf("%lf", &A[i*n+j]);
} /* Read_matrix */

/**************************************************************************
 * Function: Read_vector
 * Purpose: Read in the vector x
 * In arg: prompt, n
 * Out arg: x
 */
void Read_vector(char* prompt, double x[], int n) {
    int i;

    printf("%s\n", prompt);
    for (i = 0; i < n; i++)
        scanf("%lf", &x[i]);
} /* Read_vector */
Pthreads matrix-vector multiplication

/*----------------------------------------*
 * Function:    Pth_mat_vect
 * Purpose:      Multiply an mxn matrix by an nx1 column vector
 * In arg:       rank
 * Global in vars: A, x, m, n, thread_count
 * Global out var: y
 */

void *Pth_mat_vect(void* rank) {
    long my_rank = (long) rank;
    int i, j;
    int local_m = m/thread_count;
    int my_first_row = my_rank*local_m;
    int my_last_row = (my_rank+1)*local_m - 1;

    for (i = my_first_row; i <= my_last_row; i++) {
        y[i] = 0.0;
        for (j = 0; j < n; j++)
            y[i] += A[i*n+j]*x[j];
    }

    return NULL;
} /* Pth_mat_vect */

/*----------------------------------------*
 * Function:    Print_matrix
 * Purpose:      Print the matrix
 * In args:      title, A, m, n
 */

void Print_matrix(char* title, double A[], int m, int n) {
    int i, j;

    printf("%s\n", title);
    for (i = 0; i < m; i++) {
        for (j = 0; j < n; j++)
            printf("%4.1f ", A[i*n+j]);
        printf("\n");
    }
} /* Print_matrix */
Compiling and Running Pth_Mat_Vec on tuckoo

```
[mthomas@tuckoo pacheco/ch4] mthomas% gcc -g -Wall -o pth_matVect pth_mat_vect.c -lpthread
[mthomas@tuckoo pacheco/ch4] mthomas% ./pth_matVect 4
Enter m and n
4 4
Enter the matrix
1 2 3 4
5 6 7 8
9 10 11 12
1 2 3 4
We read
1.0 2.0 3.0 4.0
5.0 6.0 7.0 8.0
9.0 10.0 11.0 12.0
1.0 2.0 3.0 4.0
Enter the vector
9 7 6 3
We read
9.0 7.0 6.0 3.0
The product is
53.0 153.0 253.0 53.0
```
Matrix Mult Example

- More Straightforward because of shared memory
- Code only *reads* shared arrays \((A, x)\), so no contention associated with shared updates of same memory location
- No thread communication
- Small jobs, small memory

Next we’ll look at what happens when multiple threads need to update same memory location
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;

See: https://www.math.hmc.edu/funfacts/ffiles/30001.1-3.shtml
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
Controlling Access and Synchronization

Critical Sections

POSIX Threads: Pacheco *pthd_pi.c* (1)

```c
/* File: pth_pi.c
 * Purpose: Try to estimate pi using the formula:
 * pi = 4*[1 - 1/3 + 1/5 - 1/7 + 1/9 - . . . ]
 * Compile: gcc -g -Wall -o pth_pi pth_pi.c -lm -lpthread
 * Run: ./pth_pi <number of threads> <n>
 * n is the number of terms of the series to use.
 * n should be evenly divisible by the number of threads
 * Input: none
 * Output: Estimate of pi as computed by multiple threads, estimate
 * as computed by one thread, and 4*arctan(1).
 * Notes:
 * 1. The radius of convergence for the series is only 1. So the series converges quite slowly.
 * 2. This version will not get right answer bcs all threads are trying to update sum!!!!
 * Function needs a critical section to control update.
 * IPP: Section 4.4 (pp. 162 and ff.)
 */
#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);

/* Only executed by main thread */
void Get_args(int argc, char* argv[]);
void Usage(char* prog_name);
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;
    double sersum;
    double piref=3.14159265358979323846264;

    /* 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(" Reference value for pi = %.15f\n", piref);
    printf(" Pthread estimate of pi = %.15f\n", sum);
    printf(" Pthread error for pi = %.15f\n", fabs(piref - sum));
    sersum = Serial_pi(n);
    printf(" Single thread est = %.15f\n", sersum);
    printf(" Single Thd err for pi = %.15f\n", fabs(piref - sersum));

    free(thread_handles);
    return 0;
} /* main */
POSIX Threads: Pacheco *pthd_pi.c* (3)

*-----------------------------------------------
* 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, my_n = n/thread_count, my_first_i = my_n*my_rank, 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 */

/*-----------------------------------------------
* 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, factor = 1.0;
  long long i;

  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

[mthomas@tuckoo ch4]$ ./pth_pi 100 1000
[With n = 1000 terms,
  Reference value for pi = 3.141592653589793
  Pthread estimate of pi = 3.140592653839794
  Pthread error for pi = 0.0009999997499999
  Single thread est = 3.140592653839794
  Single Thd err for pi = 0.0009999997499999
]
[mthomas@tuckoo ch4]$ ./pth_pi 100 10000
With n = 10000 terms,
  Reference value for pi = 3.141592653589793
  Pthread estimate of pi = 3.141492653590034
  Pthread error for pi = 0.000099999999759
  Single thread est = 3.141492653590034
  Single Thd err for pi = 0.000099999999759
[mthomas@tuckoo ch4]$ ./pth_pi 100 100000
With n = 100000 terms,
  Reference value for pi = 3.141592653589793
  Pthread estimate of pi = 3.142916601214706
  Pthread error for pi = 0.001323947624913
  Single thread est = 3.141582653589720
  Single Thd err for pi = 0.000010000000073
[mthomas@tuckoo ch4]$ ./pth_pi 100 1000000
With n = 1000000 terms,
  Reference value for pi = 3.141592653589793
  Pthread estimate of pi = 3.004711135456170
  Pthread error for pi = 3.169991845683063
  Single thread est = 3.141592553589792
  Single Thd err for pi = 0.000000100000002
[mthomas@tuckoo ch4]$ ./pth_pi 100 10000000
With n = 10000000 terms,
  Reference value for pi = 3.141592653589793
  Pthread estimate of pi = -0.028399192093270
  Pthread error for pi = 3.169991845683063
  Single thread est = 3.141592553589792
  Single Thd err for pi = 0.000000100000002

------------------------------------------------------------- error increasing
### 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++;
```

- *flag initialized to 0 by main thread*

- 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
[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

```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)
        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
Next class: 03/24/15: Pthreads: Mutexes
HW #4 (MPI): 03/26/15: Characterizing 1D MPI Communication
HW #5 (MPI): 04/09/15: MPI: Calculating Bessel Functions Using Matrix-Matrix Multiplication
Spring Break: 03/30/15 - 04/05/15