Table of Contents

1. Pthreads: Controlling Access and Synchronization
   - Critical Sections
   - Busy-Waiting
   - Mutexes
   - Pthreads: Producer/Consumer, Synchronization, Semaphores
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 $\pi$ using $n$ terms of a Maclaurin series:

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
First attempt:

- Parallelize similar to the way we did matrix-vector multiplication:
- Divide iterates in the `for` loop and make `sum` a shared variable.

```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, 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 */
```
Program run with 2 threads, 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>

- For two threads, as \( n \uparrow \) accuracy of \( \pi \uparrow \)
- But, as \# threads \( \uparrow \) accuracy of \( \pi \downarrow \)
This Leads to 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.
/* 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 */
Pthreads: Controlling Access and Synchronization

Critical Sections

[mthomas@tuckoo ch4]$ ./pth_pi 100 1000
With n = 1000 terms,
Reference value for pi = 3.14159265389793
Pthread estimate of pi = 3.1409265389794
Pthread error for pi = 0.000099999749999
Single thread est = 3.1409265389794
Single Thd err for pi = 0.000099999749999

[mthomas@tuckoo ch4]$ ./pth_pi 100 10000
With n = 10000 terms,
Reference value for pi = 3.14159265389793
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 = 3.14159265389793
Pthread estimate of pi = 3.142916601214706
Pthread error for pi = 0.001323947624913 -----------------------------------> error increasing
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 = 3.14159265389793
Pthread estimate of pi = -0.028399192093270
Pthread error for pi = 3.169991845683063
Single thread est = 3.141592553589792
Single Thd err for pi = 0.0000000100000002
Busy-Waiting

- A thread repeatedly tests a condition

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

```c
x = x + y;
flag++;
```

Flag initialized to 0 by main thread

- Thread 1 cannot enter critical section until Thread 0 has finished.
- Beware of optimizing compilers:
  They 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
Busy-Waiting

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 $\pi = 3.141592653589793$

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: Controlling Access and Synchronization

Busy-Waiting

---

* 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
* /

```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)
        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 */
```

Define local sum, then update global sum in a critical section after loop
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]$ ./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
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`.

```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* mutex_p /* in/out */);
```

calling thread waits until no other thread is in critical section

steps:
- declare global mutex variable
- have main thread init variable
- use `pthread_mutex_lock` work use `pthread_mutex_unlock` pair
- this is a `blocking` call
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 */
### 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.
**Mutexes**

- **Thread runtimes**
  - **Busy-Wait**
  - **Mutex**

The graph shows the runtime in seconds for different numbers of threads, comparing Busy-Wait and Mutex methods. The runtime increases substantially with the number of threads for Busy-Wait, whereas the Mutex method maintains a relatively constant runtime.
OS X, N=100000, Intel 4 cores. Note: data not always reproducible, fastest times out of 5-10 runs used. Had to increase number of points in order to get difference in run times.
A few observations

- Results on OS X are similar to text. What would happen on tuckoo?
- The order in which threads execute is random
- This is effectively a barrier, so you expect mutex performance to degrade ($N_{threads} > N_{cores}$)
- if $T \frac{T_{serial}}{T_{parallel}} \approx \text{threadcount}$ then you have *Speedup*
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>
Busy-waiting enforces the order in which threads access a critical section.

Using mutexes, the order is left to chance and the system.

There are applications where we need to control the order threads access the critical section.

Trade-off between safety (mutex) and control (busy-wait) and performance.
Global sum function that uses a mutex.

```c
/* n and product_matrix are shared and initialized by the main thread */
/* product_matrix is initialized to be the identity matrix */

void* Thread_work(void* rank) {
    long my_rank = (long) rank;
    matrix_t my_mat = Allocate_matrix(n);
    Generate_matrix(my_mat);
    pthread_mutex_lock(&mutex);
    Multiply_matrix(product_mat, my_mat);
    pthread_mutex_unlock(&mutex);
    Free_matrix(&my_mat);
    return NULL;
} /* Thread_work */
```

Problem: Matrix-Matrix multiplication is not commutative.
First attempt at sending messages using Pthreads

```c
/* messages has type char**. It's allocated in main. */
/* Each entry is set to NULL in main. */

void *Send_msg(void *rank) {
    long my_rank = (long) rank;
    long dest = (my_rank + 1) % thread_count;
    long source = (my_rank + thread_count - 1) % thread_count;
    char* my_msg = malloc(MSG_MAX*sizeof(char));

    sprintf(my_msg, "Hello to %ld from %ld", dest, my_rank);
    messages[dest] = my_msg;

    if (messages[my_rank] != NULL)
        printf("Thread %ld > %s\n", my_rank, messages[my_rank]);
    else
        printf("Thread %ld > No message from %ld\n", my_rank, source);

    return NULL;
}
/* Send_msg */
```

$$[P_{source}] \rightarrow [P_{myrank}] \rightarrow [P_{destination}]$$
/* File: pth_msg.c */

/* Purpose: Illustrate a synchronization problem with pthreads: create
some threads, each of which creates and prints a message. */

/* Input: none */
/* Output: message from each thread */

/* Compile: gcc -g -Wall -o pth_msg pth_msg.c -lpthread */
/* Usage: pth_msg <thread_count> */

/* IPP: Section 4.7 (pp. 172 and ff.) */

#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

const int MAX_THREADS = 1024;
const int MSG_MAX = 100;

/* Global variables: accessible to all threads */
int thread_count;
char** messages;

void Usage(char* prog_name);
void *Send_msg(void* rank); /* Thread function */
/*-----------------------------------------*/
int main(int argc, char* argv[]) {
    long thread;
    pthread_t* thread_handles;
    
    if (argc != 2) Usage(argv[0]);
    thread_count = strtol(argv[1], NULL, 10);
    if (thread_count <= 0 || thread_count > MAX_THREADS) Usage(argv[0]);
    
    thread_handles = (pthread_t*) malloc (thread_count*sizeof(pthread_t));
    messages = (char**) malloc(thread_count*sizeof(char*));
    for (thread = 0; thread < thread_count; thread++)
        messages[thread] = NULL;
    
    for (thread = 0; thread < thread_count; thread++)
        pthread_create(&thread_handles[thread], (pthread_attr_t*) NULL,
                        Send_msg, (void*) thread);
    
    for (thread = 0; thread < thread_count; thread++)
        pthread_join(thread_handles[thread], NULL);
    
    for (thread = 0; thread < thread_count; thread++)
        free(messages[thread]);
    free(messages);
    free(thread_handles);
    return 0;
} /* main */
/* Function: Usage
 * Purpose: Print command line for function and terminate
 * In arg: prog_name
 */
void Usage(char* prog_name) {
    fprintf(stderr, "usage: %s <number of threads>\n", prog_name);
    exit(0);
} /* Usage */

/* Function: Send_msg
 * Purpose: Create a message and ‘send’ it by copying it into the global messages array. Receive a message and print it.
 * In arg: rank
 * Global in: thread_count
 * Global in/out: messages
 * Return val: Ignored
 * Note: The my_msg buffer is freed in main
 */
void *Send_msg(void* rank) {
    long my_rank = (long) rank;
    long dest = (my_rank + 1) % thread_count;
    long source = (my_rank + thread_count - 1) % thread_count;
    char* my_msg = (char*) malloc(MSG_MAX*sizeof(char));
    sprintf(my_msg, "Hello to %ld from %ld", dest, my_rank);
    messages[dest] = my_msg;

    if (messages[my_rank] != NULL)
        printf("Thread %ld > %s\n", my_rank, my_msg);
    else
        printf("Thread %ld > No message from %ld\n", my_rank, source);
    free(my_msg);
    return NULL;
} /* Send_msg */
Sending Messages Using Pthreads: mutex does not control when messages are sent so some get lost.
Possible Solutions

- Try busy-wait, but we will waste cpu time.
  
  ```
  while (messages[my_rank] == NULL)
    printf("Thread %d > %s", my_rank, messages[my_rank])
  ```

- There is no MPI style send/recv pairs

- Find way to notify destination thread, not easy to do with mutexes

  ```
  messages[dest] = my_msg;
  Notify thread [dest] to enter block
  ```

  ```
  Await notification from thread [source]
  printf("Thread %d > %s", my_rank, messages[my_rank])
  ```

- Solution: Semaphores
What is a semaphore?

Ask.com:
semaphore
Noun:
A system of sending messages by holding the arms or two flags or poles positions according to an alphabetic code.
Verb:
Send (a message) by semaphore or by signals resembling semaphore.
Synonyms:
noun. traffic light - traffic lights - signal
verb. signal

Wikipedia:
In computer science, a semaphore is a variable or abstract data type that provides a simple but useful abstraction for controlling access by multiple processes to a common resource in a parallel programming environment.
Possible Solutions

- unsigned int
- binary semaphore = 0,1 == locked, unlocked
- usage:
  1. *init* semaphore to 1 (unlocked)
  2. before critical block, thread places call to *sem_wait*
  3. if *semaphore* > 1, decrement semaphore and enter critical block
  4. when done, call *sem_post*, which increments semaphore for next thread

- semaphores have no ownership: any thread can modify them
- semaphores are not part of Pthreads, so need to include *semaphore.h*
Syntax of the various semaphore functions

#include <semaphore.h>

int sem_init(  
    sem_t* semaphore_p  /* out */,  
    int shared  /* in */,  
    unsigned initial_val  /* in */);  

int sem_destroy(sem_t* semaphore_p /* in/out */);  
int sem_post(sem_t* semaphore_p /* in/out */);  
int sem_wait(sem_t* semaphore_p /* in/out */);  

Semaphores are not part of Pthreads; you need to add this.
Send_msg using semaphore

/*-------------------------------------------------------------------
* Function: Send_msg
* Purpose: Create a message and ‘send’ it by copying it
* into the global messages array. Receive a message
* and print it.
* In arg: rank
* Global in: thread_count
* Global in/out: messages, semaphores
* Return val: Ignored
* Note: The my_msg buffer is freed in main
*/
void *Send_msg(void* rank) {
    long my_rank = (long) rank;
    long dest = (my_rank + 1) \% thread_count;
    char* my_msg = (char*) malloc(MSG_MAX*sizeof(char));

    sprintf(my_msg, "Hello to \%ld from \%ld", dest, my_rank);
    messages[dest] = my_msg;
    sem_post(&semaphores[dest]); /* "Unlock" the semaphore of dest */

    sem_wait(&semaphores[my_rank]); /* Wait for our semaphore to be unlocked */
    printf("Thread \%ld > \%s\n", my_rank, messages[my_rank]);

    return NULL;
} /* Send_msg */
Send_msg output on tuckoo using PBS node

[mthomas@tuckoo ch4]$ cat pth_msg_sem.o63124
Thread 1 > Hello to 1 from 0
Thread 2 > Hello to 2 from 1
Thread 5 > Hello to 5 from 4
Thread 3 > Hello to 3 from 2
Thread 4 > Hello to 4 from 3
Thread 6 > Hello to 6 from 5
Thread 7 > Hello to 7 from 6
Thread 8 > Hello to 8 from 7
Thread 9 > Hello to 9 from 8
Thread 10 > Hello to 10 from 9
Thread 11 > Hello to 11 from 10
Thread 12 > Hello to 12 from 11
Thread 13 > Hello to 13 from 12
Thread 14 > Hello to 14 from 13
Thread 15 > Hello to 15 from 14
Thread 16 > Hello to 16 from 15
Thread 17 > Hello to 17 from 16
Thread 18 > Hello to 18 from 17
Thread 19 > Hello to 19 from 18
Thread 20 > Hello to 20 from 19
Thread 21 > Hello to 21 from 20
Thread 22 > Hello to 22 from 21
Thread 23 > Hello to 23 from 22
Thread 24 > Hello to 24 from 23
Thread 25 > Hello to 25 from 24
Thread 26 > Hello to 26 from 25
Thread 27 > Hello to 27 from 26
Thread 28 > Hello to 28 from 27
Thread 29 > Hello to 29 from 28
Thread 0 > Hello to 0 from 29
Send_msg output on OS Mountain Lion

[gidget] mthomas\% ./pth_msg_sem 30
Thread 0 > (null)
Thread 2 > (null)
Thread 1 > Hello to 1 from 0
Thread 3 > Hello to 3 from 2
Thread 4 > Hello to 4 from 3
Thread 5 > Hello to 5 from 4
Thread 6 > Hello to 6 from 5
Thread 7 > Hello to 7 from 6
Thread 8 > Hello to 8 from 7
Thread 11 > Hello to 11 from 10
Thread 10 > (null)
Thread 9 > Hello to 9 from 8
Thread 12 > Hello to 12 from 11
Thread 13 > Hello to 13 from 12
Thread 14 > Hello to 14 from 13
Thread 15 > Hello to 15 from 14
Thread 16 > Hello to 16 from 15
Thread 17 > Hello to 17 from 16
Thread 19 > (null)
Thread 18 > Hello to 18 from 17
Thread 20 > Hello to 20 from 19
Thread 21 > Hello to 21 from 20
Thread 22 > Hello to 22 from 21
Thread 23 > Hello to 23 from 22
Thread 24 > Hello to 24 from 23
Thread 25 > Hello to 25 from 24
Thread 26 > Hello to 26 from 25
Thread 27 > Hello to 27 from 26
Thread 28 > Hello to 28 from 27
Thread 29 > Hello to 29 from 28