CS 596: Introduction to Parallel Computing
Lecture 15: Shared Memory Programming with Pthreads

Mary Thomas

Department of Computer Science
Computational Science Research Center (CSRC)
San Diego State University (SDSU)

Posted: 10/16/14
Last Update: 10/16/14
Table of Contents

1 Shared Memory Programming with PThreads
2 Basic Pthreads Program
   • Hello World
   • Matrix-Vector Multiplication
3 Controlling Access to Critical Sections
   • Busy-Waiting
   • Pthreads - Mutexes
   • Pthreads - Producer/Consumer Model, Synchronization and Semaphores
   • PThreads - Barriers and Condition Variables
4 Next Time
**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 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.
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).
Shared Memory Programming with PThreads

UNIX PROCESS

THREADS WITHIN A UNIX PROCESS
POSIX Threads

- 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,
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.
**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.
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

Source: https://computing.llnl.gov/tutorials/pthreads/
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

/* Global variable: accessible to all threads */
int thread_count;

void *Hello(void* rank); /* Thread function */

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

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 */
    if (argc != 2) Usage(argv[0]);

    thread_count = strtol(argv[1], NULL, 10);
    if (thread_count <= 0 || thread_count > MAX_THREADS) Usage(argv[0]);
```c
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 */

/* Hello */

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

    return NULL;
} /* Hello */
```
Compiling and running a Pthreads program

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

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

```bash
[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
Matrix-Vector Multiplication with Pthreads

\[
\begin{array}{cccc}
  a_{00} & a_{01} & \cdots & a_{0,n-1} \\
  a_{10} & a_{11} & \cdots & a_{1,n-1} \\
  \vdots & \vdots & \ddots & \vdots \\
  a_{i0} & a_{i1} & \cdots & a_{i,n-1} \\
  \vdots & \vdots & \ddots & \vdots \\
  a_{m-1,0} & a_{m-1,1} & \cdots & a_{m-1,n-1} \\
\end{array}
\]

\[
\begin{array}{c}
  x_0 \\
  x_1 \\
  \vdots \\
  x_{n-1} \\
\end{array}
\]

\[
\begin{array}{c}
  y_0 \\
  y_1 \\
  \vdots \\
  y_{m-1} \\
\end{array}
\]

\[
y_i = a_{i0}x_0 + a_{i1}x_1 + \ldots + a_{i,n-1}x_{n-1}
\]
Matrix-Vector Multiplication with Pthreads

\[
\begin{bmatrix}
  a_{i1} & \ldots & a_{ij} & \ldots & a_{in}
  \\
  \vdots
  \\
  a_{i1} & \ldots & a_{ij} & \ldots & a_{in}
  \\
  a_{m1} & \ldots & a_{mj} & \ldots & a_{mn}
\end{bmatrix}
\begin{bmatrix}
  b_{1j}
  \\
  \vdots
  \\
  b_{ij}
  \\
  \vdots
  \\
  b_{nj}
\end{bmatrix}
= 
\begin{bmatrix}
  c_{11} & \ldots & c_{1j} & \ldots & c_{1p}
  \\
  \vdots
  \\
  c_{i1} & \ldots & c_{ij} & \ldots & c_{ip}
  \\
  \vdots
  \\
  c_{m1} & \ldots & c_{mj} & \ldots & c_{mp}
\end{bmatrix}
\]
Serial Pseudo-code

```c
/* 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

<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>

thread 0

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

general case

\[
y[i] = 0.0; \\
\text{for } (j = 0; j < n; j++) \\
y[i] += A[i][j] \times x[j];
\]
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;
}
Compiling and Running Pth_Mat_Vec on tuckoo

[mthomas@tuckoo pacheco/ch4] mthomas% gcc -g -Wall -o pth_mat_vect pth_mat_vect.c -lpthread
[mthomas@tuckoo pacheco/ch4] mthomas% ./pth_mat_vect 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
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;
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>$\pi$</th>
<th>$10^5$</th>
<th>$10^6$</th>
<th>$10^7$</th>
<th>$10^8$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>3.14159</td>
<td>3.141593</td>
<td>3.1415927</td>
<td>3.14159265</td>
<td></td>
</tr>
<tr>
<td>1 Thread</td>
<td>3.14158</td>
<td>3.141592</td>
<td>3.1415926</td>
<td>3.14159264</td>
<td></td>
</tr>
<tr>
<td>2 Threads</td>
<td>3.14158</td>
<td>3.141480</td>
<td>3.1413692</td>
<td>3.14164686</td>
<td></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
POSIX Threads: Pacheco *pthd_pi*.c (1)

```c
/* 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 = %ld 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 */
POSIX Threads: Pacheco \textit{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 to 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>Started by main thread</td>
</tr>
<tr>
<td>3</td>
<td>Assign y = 1</td>
<td>Call Compute()</td>
</tr>
<tr>
<td>4</td>
<td>Put x=0 and y=1 into registers</td>
<td>Assign y = 2</td>
</tr>
<tr>
<td>5</td>
<td>Add 0 and 1</td>
<td>Put x=0 and y=2 into registers</td>
</tr>
<tr>
<td>6</td>
<td>Store 1 in memory location x</td>
<td>Add 0 and 2</td>
</tr>
<tr>
<td>7</td>
<td>Store 2 in memory location x</td>
<td></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.
Controlling Access to Critical Sections

Busy-Waiting

**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
Pthreads: global sum with 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 */
```
Controlling Access to Critical Sections

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
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 */ );
```
Mutexes

- 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 */
Controlling Access to Critical Sections

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.
Controlling Access to Critical Sections

Pthreads - Mutexes

Thread runtimes

- **Busy-Wait**
- **Mutex**

<table>
<thead>
<tr>
<th>Busy Wait</th>
<th>0</th>
<th>10</th>
<th>20</th>
<th>30</th>
<th>40</th>
<th>50</th>
<th>60</th>
<th>70</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pthreads</td>
<td>0</td>
<td>0.5</td>
<td>1</td>
<td>1.5</td>
<td>2</td>
<td>2.5</td>
<td>3</td>
<td>4</td>
</tr>
</tbody>
</table>

Legend:
- Blue line: Busy-Wait
- Red line: Mutex
Controlling Access to Critical Sections

Pthreads - Mutexes

**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 $\frac{T_{parallel}}{T_{serial}} \approx \text{threadcount}$ then you have Speedup
### Controlling Access to Critical Sections

### Pthreads - Mutexes

<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>
</tbody>
</table>

Possible sequence of events with busy-waiting and more threads than cores.
Busy-waiting enforces the order 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.
Matrix-Matrix Challenge

```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.
Sending messages using pthreads

`void *Send_msg(void *rank)`

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

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 */
```

\[
[P_{source}] \rightarrow [P_{myrank}] \rightarrow [P_{destination}]
\]
Sending Messages Using Pthreads: mutex does not control when messages are sent so some get lost.

[gidget:intro-par/pgming-pacheco/ipp-source/ch4] mthomas% ./pth_msg 4
Thread 0 > No message from 3
Thread 1 > Hello to 1 from 0
Thread 3 > No message from 2
Thread 2 > Hello to 2 from 1

[gidget:intro-par/pgming-pacheco/ipp-source/ch4] mthomas% ./pth_msg 10
Thread 0 > No message from 9
Thread 3 > No message from 2
Thread 2 > No message from 1
Thread 1 > Hello to 1 from 0
Thread 5 > No message from 4
Thread 4 > Hello to 4 from 3
Thread 6 > Hello to 6 from 5
Thread 7 > Hello to 7 from 6
Thread 9 > No message from 8
Thread 8 > Hello to 8 from 7
Possible Solutions

- Try busy-wait, but we will waste cpu time.
  
  ```c
  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
  
  ```c
  messages [dest] = my_msg;
  Notify thread [P_dest] to enter block
  ```

  ```c
  Await notification from thread [P_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 in 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 */);
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 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
Barriers and Condition Variables

- **Barriers** are used for timing, debugging, and synchronization of the threads.
- Used to make sure that they are all at the same point in a program.
- Not part of the Pthreads standard, so have to build customized barrier.
Using barriers to time the slowest thread

```c
/* Shared */
double elapsed_time;
.

/* Private */
double my_start, my_finish, my_elapsed;
.
Synchronize threads;
Store current time in my_start;
/* Execute timed code */
.
Store current time in my_finish;
my_elapsed = my_finish - my_start;

elapsed = Maximum of my_elapsed values;
```
Using barriers for debugging

point in program we want to reach;
barrier;
if (my_rank == 0) {
    printf("All threads reached this point\n");
    fflush(stdout);
}
Busy-waiting and a Mutex

- Implementing a barrier using busy-waiting and a mutex is straightforward.
- We use a shared counter protected by the mutex.
- When the counter indicates that every thread has entered the critical section, threads can leave the critical section.
Busy-waiting and a Mutex

/* Shared and initialized by the main thread */
int counter; /* Initialize to 0 */
int thread_count;
pthread_mutex_t barrier_mutex;
...

void* Thread_work(...) {
  ...
  /* Barrier */
  pthread_mutex_lock(&barrier_mutex);
  counter++;
  pthread_mutex_unlock(&barrier_mutex);
  while (counter < thread_count);
  ...
}

PE’s could still end up spinning. Issue with global mutex counter: not all threads will see its value, could result in hung processes.

We need one counter variable for each instance of the barrier, otherwise problems are likely to occur.
Implementing a barrier with semaphores

```c
/* Shared variables */
int counter;    /* Initialize to 0 */
sem_t count_sem; /* Initialize to 1 */
sem_t barrier_sem; /* Initialize to 0 */

void* Thread_work(...) {
    . . .
    /* Barrier */
    sem_wait(&count_sem);
    if (counter == thread_count - 1) {
        counter = 0;
        sem_post(&count_sem);
        for (j = 0; j < thread_count - 1; j++)
            sem_post(&barrier_sem);
    } else {
        counter++;
        sem_post(&count_sem);
        sem_wait(&barrier_sem);
    }
    . . .
```
Condition Variables

- A condition variable is a data object that allows a thread to suspend execution until a certain event or condition occurs.
- When the event or condition occurs another thread can signal the thread to "wake up."
- A condition variable is always associated with a mutex.
Condition Variables

lock mutex;
if condition has occurred
  signal thread(s);
else {
  unlock the mutex and block;
  /* when thread is unblocked, mutex is relocked */
}
unlock mutex;
Send_msg output on OS Mountain Lion

API:

- `pthread_cond_init (condition, attr)`  -- dynamically initialize condition variables
- `pthread_cond_destroy (condition)`  -- destroy condition variables
- `pthread_condattr_init (attr)`
- `pthread_condattr_destroy (attr)`

- `pthread_mutex_lock (mutex)`  -- used by a thread to acquire a lock on the specified mutex variable
- `pthread_mutex_trylock (mutex)`
- `pthread_mutex_unlock (mutex)`

- `pthread_cond_wait (condition, mutex)`  -- blocks the calling thread until the specified condition is signalled
- `pthread_cond_signal (condition)`  -- signal (or wake up) another thread which is waiting on the condition variable.
- `pthread_cond_broadcast (condition)`  -- use instead of `pthread_cond_signal()` if more than one thread is waiting
Implementing a barrier with condition variables

```c
/* Shared */
int counter = 0;
pthread_mutex_t mutex;
pthread_cond_t cond_var;

void* Thread_work(...) {
    ...

    /* Barrier */
    pthread_mutex_lock(&mutex);
    counter++;
    if (counter == thread_count) {
        counter = 0;
        pthread_cond_broadcast(&cond_var);
    } else {
        while (pthread_cond_wait(&cond_var, &mutex) != 0);
    }
    pthread_mutex_unlock(&mutex);
}
```
## Comparing three barrier methods

<table>
<thead>
<tr>
<th>pthreads</th>
<th>pth_cond_bar</th>
<th>pth_sem_bar</th>
<th>pth_busy_bar</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>4.87E-04</td>
<td>2.36E-04</td>
<td>4.66E-04</td>
</tr>
<tr>
<td>4</td>
<td>2.24E-03</td>
<td>3.14E-04</td>
<td>2.15E-03</td>
</tr>
<tr>
<td>8</td>
<td>1.21E-02</td>
<td>4.95E-04</td>
<td>3.88E-02</td>
</tr>
<tr>
<td>32</td>
<td>2.65E-02</td>
<td>2.53E-03</td>
<td>8.22E+00</td>
</tr>
<tr>
<td>64</td>
<td>6.03E-02</td>
<td>5.12E-03</td>
<td>2.60E+01</td>
</tr>
<tr>
<td>128</td>
<td>1.10E-01</td>
<td>9.60E-03</td>
<td>4.12E+01</td>
</tr>
<tr>
<td>256</td>
<td>2.20E-01</td>
<td>1.79E-02</td>
<td>8.04E+01</td>
</tr>
<tr>
<td>512</td>
<td>4.67E-01</td>
<td>3.18E-02</td>
<td>1.49E+02</td>
</tr>
</tbody>
</table>
Comparing three barrier methods

Run-time vs Number of Pthreads

Time (seconds) vs Pthreads for different barrier methods:
- **Condition**
- **Semph**
- **Busy**
Next Time

- Next class: 10/21/14
- HW #3 Due: 10/23/14
- Quiz 2 (MPI) and Quiz 3 (Pthreads) will be held on 11/04/14.
- Quiz 2: two parts:
  - Part 1: take home assignment on 10/30/14, due 11/04/14
    [was 10/23/14, due 10/28]
    (75% of the grade)
  - Part 2: in class quiz on 11/04/14 [was 10/28]
    (25% of the grade)