Chapter 4

Shared Memory Programming with Pthreads
Processes and Threads

- Recall: a process is an instance of a running (or suspended) program.
- In a shared memory program a single process may have multiple threads of control.
- Each thread can be executing a different line of code in the program, but all threads share the address space of the process.
POSIX®Threads aka Pthreads

POSIX is a set of standards for Unix-like operating systems. Pthreads is part of the standard.

- Includes a library that can be linked with C and C++ programs.
- Specifies an application programming interface (API) for multi-threaded programming.
Caveat

- The Pthreads API is only available on POSIXR systems — Linux, MacOS X, Solaris, HPUX, …
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

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

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

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 */
    thread_count = strtol(argv[1], NULL, 10);

    thread_handles = malloc(thread_count*sizeof(pthread_t));

    declares the various Pthreads functions, constants, types, etc.
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 */
```c
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 a Pthread program

gcc -g -Wall -o pth_hello pth_hello.c -lpthread

link in the Pthreads library
Global variables

- Using global variables to share data among threads can introduce subtle and confusing bugs!

- So limit use of global variables to situations where really needed:
  - Shared variables.
Starting the Threads

- Processes in MPI are usually started by a *script* provided by the system.
- With Pthreads the threads are explicitly started by the programmer as part of the program executable.
Starting the Threads

```c
pthread_t

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

One object for each thread.
pThread_t objects

- Opaque – we can’t access them
- The data stored is system-specific.
- Internal fields not directly accessible to user code.
- Pthreads standard guarantees that a pThread_t object stores enough information to uniquely identify the associated thread.
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 this, so just pass NULL.

Address of the pthread handle - allocate before calling.
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 start_routine.

Address of 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.
MATRIX-VECTOR MULTIPLICATION IN PTHREADS
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 threads

<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

```c
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][j]*x[j];
    }

    return NULL;
} /* Pth_mat_vect */
```
CRITICAL SECTIONS
Estimating $\pi$ 

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

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;
Using a dual core processor

<table>
<thead>
<tr>
<th></th>
<th>$n$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$10^5$</td>
</tr>
<tr>
<td>$\pi$</td>
<td>3.14159</td>
</tr>
<tr>
<td>1 Thread</td>
<td>3.14158</td>
</tr>
<tr>
<td>2 Threads</td>
<td>3.14158</td>
</tr>
</tbody>
</table>

Note that as we increase $n$, the estimate with one thread gets better and better.
A thread function for computing π

```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 */
```
### 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 <code>Compute()</code></td>
<td>Call <code>Compute()</code></td>
</tr>
<tr>
<td>3</td>
<td>Assign $y = 1$</td>
<td>Call <code>Compute()</code></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></td>
<td>Store 2 in memory location $x$</td>
</tr>
</tbody>
</table>
Possible solution: busy-waiting

Here, a thread sits in a loop testing a condition to see if it can continue, but does no useful work until the condition has the appropriate value.

Beware of optimizing compilers, though!

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

flag initialized to 0 by main thread
Pthreads global sum with busy-waiting

```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) {
        while (flag != my_rank); // busy waiting here
        sum += factor/(2*i+1);
        flag = (flag+1) % thread_count;
    }

    return NULL;
} /* Thread_sum */
```
void* Thread_sum(void* rank) {
    long my_rank = (long) rank;
    double factor, my_sum = 0.0;
    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 */
Mutexes

- A thread that is busy-waiting uses the CPU continually yet may be accomplishing nothing.

- A *mutex* (*mutual exclusion*) is a special variable used to restrict access to a critical section to a single thread at a time.
Mutexes

Use a mutex to *guarantee* that program *excludes* all other threads while it executes the critical section.

Pthreads library includes a special type for mutexes: `pthread_mutex_t`. To create a mutex:

```c
int pthread_mutex_init(
    pthread_mutex_t* mutex_p, /* out */
    const pthread_mutexattr_t* attr_p /* in */);
```
Mutexes

In order to gain access to a critical section a thread calls:

```c
int pthread_mutex_lock(pthread_mutex_t* mutex_p /* in/out */);
```

When a thread is finished executing the code in a critical section, call:

```c
int pthread_mutex_unlock(pthread_mutex_t* mutex_p /* in/out */);
```

When the program finishes using a mutex, it calls:

```c
int pthread_mutex_destroy(pthread_mutex_t* mutex_p /* in/out */);
```
Global sum function using a mutex (1)

```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;
    double my_sum = 0.0;

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

    // rest of the function
}
```
Global sum function that uses a mutex (2)

```c
for (i = my_first_i; i < my_last_i; i++, factor = -factor) {
    my_sum += factor/(2*i+1);
}
pthread_mutex_lock(&mutex);    // Initiate mutual exclusion
sum += my_sum;
pthread_mutex_unlock(&mutex);  // Release mutual exclusion

return NULL;
} /* Thread_sum */
```
<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 \(\pi\) programs using \(n = 108\) terms on a system with two four-core processors.
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>2</td>
<td>—</td>
<td>—</td>
<td>crit sect</td>
<td>susp</td>
<td>busy wait</td>
</tr>
</tbody>
</table>


Summary

A thread in shared-memory programming is analogous to a process in distributed memory programming. However, a thread is often “lighter-weight” than a full-fledged process.

When using Pthreads, all threads have access to any global variables, while local variables usually are private to the thread running the function (when are they not?)
A race condition occurs when multiple threads attempting to access a shared resource (such as a global variable or a shared file), at least one of which is performing an update, results in an indeterminate result.

Put another way: the result depends on which thread acquires access to the resource first.
Summary

A critical section is a block of code that updates a shared resource that can only be executed by one thread at a time.

Note that execution of a critical section is, effectively, serial code.

Therefore we want as to minimize the size and number of critical sections.