Shared Memory Programming with Pthreads

An Introduction to Parallel Programming.
Goal

- Problems programming shared memory systems.
- Controlling access to a critical section.
- Thread synchronization.
- Programming with POSIX threads.
- Mutexes.
- Producer-consumer synchronization and semaphores.
- Barriers and condition variables.
- Read-write locks.
- Thread safety.
A Shared Memory System
What is a process?

- A process is an instance of a running (or suspended) program.
- Can be "multi-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).
What is a Thread?

- Threads are analogous to a light-weight process
- Shared memory program: single process may have multiple threads
- Runs independently from main program (e.g. multiple functions running concurrently)
- Thread models:
  - Manager/worker: A single thread, manager assigns work to other threads (workers)
  - Pipeline: Task is broken into series of sub-operations; each handled in series, but concurrently by another thread
  - Peer: After the main thread (manager) creates other threads, it participates in the work
POSIX® Threads

- Also known as Pthreads.

- 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 POSIX systems such as: Linux, MacOS X, Solaris
Process vs Thread

Process:
- Typically independent
- Has more state information than thread
- Separate address spaces
- Interact only through system IPC

Thread:
- Subsets of a process
- Multiple threads within a process share process state, memory, etc.
- Threads share their address space
Hello World! (1)

```c
#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));
```
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) {
    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
Running a Pthreads program

. / pth_hello  <number of threads>

. / pth_hello  1

Hello from the main thread
Hello from thread 0 of 1

. / pth_hello  4

Hello from the main thread
Hello from thread 0 of 4
Hello from thread 1 of 4
Hello from thread 2 of 4
Hello from thread 3 of 4
Global variables

- All threads have access to the same global, shared memory
- Can introduce subtle and confusing bugs!
- Threads also have their own private data.
- Limit use of global variables to situations in which they’re really needed.
  - Shared variables.
Starting the Threads

- Processes in MPI are usually started by a script.
- In Pthreads the threads are started by the program executable.
Starting the Threads

```
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
- 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)

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.
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.
Function Started by pthread_create
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.
Threads Can Create Other Threads
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

#define NUM_THREADS 4

char *messages[NUM_THREADS];

void *PrintHello(void *arg)
{
    long tid;
    tid = (long)arg;

    printf("Thread %d: %s\n", tid, messages[tid]);

    pthread_exit(NULL);
}

int main(int argc, char *argv[])
{
    pthread_t threads[NUM_THREADS];
    int i, rc;

    messages[0] = "English: Hello World!";  
    messages[1] = "French: Bonjour, le monde!"; 
    messages[3] = "Klingon: Nuq neH!";

    for(i = 0; i < NUM_THREADS; i++) {
        rc = pthread_create(&threads[i], NULL, 
                            PrintHello, (void*)i);
        if(rc) {
            printf("ERROR; return code from 
                    pthread_create() is %d\n", rc);
            exit(-1);
        }
    }

    pthread_exit(NULL);
}
Passing Arguments to Threads - #2

```c
char *messages[NUM_THREADS];

struct thread_data
{
    int tid, sum;
    char *message;
};

struct thread_data thread_data_array[NUM_THREADS];

void *PrintHello(void *arg)
{
    int tid, sum;
    char *message;
    struct thread_data *my_data;

    my_data = (struct thread_data*)arg;
    tid = my_data->tid;
    sum = my_data->sum;
    message = my_data->message;

    for(sum = 0, i = 0; i < NUM_THREADS; i++) {
        sum = sum + i;
        thread_data_array[i].tid = i;
        thread_data_array[i].sum = sum;
        thread_data_array[i].message = messages[i];

        rc = pthread_create(&threads[i], NULL, PrintHello,
                            (void*)&thread_data_array[i]);

        ...
    }
}
```
MATRIX-VECTOR MULTIPLICATION IN PTHREADS

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

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

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

\[
y_i = a_{i0}x_0 + a_{i1}x_1 + \cdots + a_{i,n-1}x_{n-1}
\]
Serial pseudo-code

```plaintext
/* 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

```cpp
y[0] = 0.0;
for (j = 0; j < n; j++)
    y[0] += A[0][j]* x[j];
```

general case

```cpp
y[i] = 0.0;
for (j = 0; j < n; j++)
    y[i] += A[i][j]* 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 */
```
Run-times and efficiencies of matrix-vector multiplication

<table>
<thead>
<tr>
<th>Threads</th>
<th>Matrix Dimension</th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>8,000,000 × 8</td>
<td>8000 × 8000</td>
<td>8 × 8,000,000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0.393</td>
<td>1.000</td>
<td>0.345</td>
<td>1.000</td>
<td>0.441</td>
</tr>
<tr>
<td>2</td>
<td>0.217</td>
<td>0.906</td>
<td>0.188</td>
<td>0.918</td>
<td>0.300</td>
</tr>
<tr>
<td>4</td>
<td>0.139</td>
<td>0.707</td>
<td>0.115</td>
<td>0.750</td>
<td>0.388</td>
</tr>
</tbody>
</table>

(times are in seconds)
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?
Race Condition

• Also called critical section problem
• A race condition or data race occurs when:
  – two processors (or two threads) access the same variable, and at least one does a write
  – The accesses are concurrent (not synchronized) so they could happen simultaneously
Synchronization Solutions

- Busy-Waiting
- Mutexes (locks)
- Semaphores
- Barriers Conditional Variables
Estimating $\pi$

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

default factor = 1.0;
default 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

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

<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>
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>

\[y = \text{compute(my\_rank)};\]
\[x = x + y;\]
Busy-Waiting

- A thread repeatedly tests a condition, but, effectively, does no useful work until the condition has the appropriate value.

- Beware of optimizing compilers, though!

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

flag initialized to 0 by main thread
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;
    while (flag != my_rank);
        sum += factor/(2*i+1);
        flag = (flag+1) % thread_count;
    }

    return NULL;
} /* Thread_sum */

sum is a shared global variable. Can we transform code and minimize thread interaction on this variable?
Global sum function with critical section after loop (1)

```c
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;
}
```

See “pth_pi_busy2.c”
Global sum function with critical section after loop (2)

```c
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

- Mutex (mutual exclusion) is a special type of variable used to restrict access to a critical section to a single thread at a time
- Guarantee that one thread “excludes” all other threads while it executes the critical section
- When a thread waits on a mutex/lock, CPU resource can be used by others
Mutexes

- The Pthreads standard includes a special type for mutexes: `pthread_mutex_t`.

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

- When a Pthreads program finishes using a mutex, it should call

```c
int pthread_mutex_destroy(pthread_mutex_t* mutex_p /* in/out */);
```
 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, it should call

```c
int pthread_mutex_unlock(pthread_mutex_t* mutex_p /* in/out */);
```
Global sum function that uses 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;
```
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);
sum += my_sum;
pthread_mutex_unlock(&mutex);

return NULL;
/* Thread_sum */
```
Run-times (in seconds) of $\pi$ programs using $n = 108$ terms on a system with two four-core processors.

<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>

\[
\frac{T_{\text{serial}}}{T_{\text{parallel}}} \approx \text{thread\_count}
\]
PRODUCER-CONSUMER SYNCHRONIZATION AND SEMAPHORES
• 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.
A 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 */
```
Syntax of the various semaphore functions

```
#include <semaphore.h>

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

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

```
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 */);
```
• Can do increments and decrements of semaphore value
• Semaphore can be initialized to any value
• Thread blocks if semaphore value is less than or equal to zero when a decrement is attempted
• As soon as semaphore value is greater than zero, one of the blocked threads wakes up and continues
• Semaphores have no ownership: any thread can modify them
• Semaphores are not part of Pthreads, so need to include semaphore.h
Barriers

- Synchronizing the threads to make sure that they all are at the same point in a program is called a barrier.
- No thread can cross the barrier until all the threads have reached it.
- 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.
/* 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);
    
}
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

```c
lock mutex;
if condition has occurred
    signal thread(s);
else {
    unlock the mutex and block;
    /* when thread is unblocked, mutex is relocked */
}
unlock mutex;
```

Pseudocode:

lock mutex
if(condition has occurred)
    signal thread(s)
else {
    unlock the mutex and block
    // when thread is unblocked, mutex is relocked
}
unlock mutex
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);
}
```
int main(int argc, char **argv) {
    pthread_t pro, con;

    // Initialize the mutex and condition variables
    pthread_mutex_init(&the_mutex, NULL);
    pthread_cond_init(&condc, NULL); // Initialize consumer condition variable
    pthread_cond_init(&condp, NULL); // Initialize producer condition variable

    // Create the threads
    pthread_create(&con, NULL, consumer, NULL);
    pthread_create(&pro, NULL, producer, NULL);

    // Wait for the threads to finish
    pthread_join(&con, NULL);
    pthread_join(&pro, NULL);

    // Cleanup
    pthread_mutex_destroy(&the_mutex); // Free up the_mutex
    pthread_cond_destroy(&condc); // Free up consumer condition variable
    pthread_cond_destroy(&condp); // Free up producer condition variable
}

• 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.
• In Pthreads programs, all the threads have access to global variables, while local variables usually are private to the thread running the function.
Concluding Remarks (2)

• When indeterminacy results from multiple threads attempting to access a shared resource such as a shared variable or a shared file, at least one of the accesses is an update, and the accesses can result in an error, we have a race condition.
Concluding Remarks (3)

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

• So the execution of code in a critical section should, effectively, be executed as serial code.
Concluding Remarks (4)

- **Busy-waiting** can be used to avoid conflicting access to critical sections with a flag variable and a while-loop with an empty body.

- It can be very wasteful of CPU cycles.

- It can also be unreliable if compiler optimization is turned on.
• A **mutex** can be used to avoid conflicting access to critical sections as well.

• Think of it as a lock on a critical section, since mutexes arrange for mutually exclusive access to a critical section.
Concluding Remarks (6)

- A **semaphore** is the third way to avoid conflicting access to critical sections.

- It is an unsigned int together with two operations: `sem_wait` and `sem_post`.

- Semaphores are more powerful than mutexes since they can be initialized to any nonnegative value.