CSCI-GA.3033-009
Multicore Processors:
Architecture & Programming

Lecture 5: Threads ... Pthreads
(Part 1)

Mohamed Zahran (aka Z)
mzahran@cs.nyu.edu
http://www.mzahran.com

Some slides of this lecture are from:
• Peter S. Pacheco book
Multithreading

Processes

Threads

Processors
Processes Vs Threads

- Process is the unit for resource allocation and a unit of protection.
- Process has its own address space.
- A thread has:
  - an execution state (Running, Ready, etc.)
  - saved thread context when not running
  - an execution stack
  - some per-thread static storage for local variables
  - access to the memory and resources of its process (all threads of a process share this)
Benefits of Threads

- Takes less time to create a new thread than a process
- Less time to terminate a thread than a process
- Switching between two threads takes less time than switching between processes
- Threads enhance efficiency in communication between programs
A Thread

- **Definition**: sequence of related instructions executed independently of other instruction sequences
- A thread can create another thread
- Each thread maintains its current machine state

<table>
<thead>
<tr>
<th>User-Level Threads</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kernel-Level Threads</td>
</tr>
<tr>
<td>Hardware Threads</td>
</tr>
</tbody>
</table>
User-Lever Threads (ULT)

- All thread management is done by the application
- The kernel is not aware of the existence of threads
User-Level Threads (ULTs)

Advantages

• Thread switch does not require kernel-mode.
• Scheduling (of threads) can be application specific.
• Can run on any OS.

Disadvantages

• A system-call by one thread can block all threads of that process.
• In pure ULT, multithreading cannot take advantage of multiprocessing.
Kernel-Level Threads (KLTs)

- Thread management is done by the kernel
- no thread management is done by the application
- Windows OS is an example of this approach
Kernel-Level Threads (KLTs)

**Advantages**
- The kernel can simultaneously schedule multiple threads from the same process on multiple processors
- If one thread in a process is blocked, the kernel can schedule another thread of the same process
- Kernel routines can be multithreaded

**Disadvantages**
- The transfer of control from one thread to another within the same process requires a switch to the kernel
Combined (Hybrid) Approach

- Thread creation is done completely in user space.
- Bulk of scheduling and synchronization of threads is by the application (i.e. user space).
- Multiple ULTs from a single application are mapped onto (smaller or equal) number of KLTs.
- Solaris is an example
Relationship Between ULTs & KLTs

• 1:1
  – user-level thread maps to kernel-level thread
  – e.g. win32, Linux (original C-library), windows 7, FreeBSD

• N:1 (user-level threads)
  – Kernel is not aware of the existence of threads
  – e.g. Early version of Java, Solaris Green Thread

• M:N
Before We Proceed ...

• How do you know your parallel program is:
  – **Effective**: correct
  – **Efficient**: makes the best use of the available parallelism
  – **Scalable**: What happens when the number of threads increases? number of cores? problem size?
Standard Definition of Performance

• For a program running on machine $X$, 

\[
\text{Performance}_X = \frac{1}{\text{Execution time}_X}
\]

• "$X$ is $n$ times faster than $Y$"

\[
\frac{\text{Performance}_X}{\text{Performance}_Y} = n
\]

- Example: time taken to run a program
  - 10s on $A$, 15s on $B$
  - Execution Time$_B$ / Execution Time$_A$
    - $= 15s / 10s = 1.5$
  - So $A$ is 1.5 times faster than $B$
Speedup

- Number of cores = $p$
- Serial run-time = $T_{\text{serial}}$
- Parallel run-time = $T_{\text{parallel}}$

$$S = \frac{T_{\text{serial}}}{T_{\text{parallel}}}$$
Efficiency of a parallel program

\[ E = \frac{S}{p} = \frac{T_{\text{serial}}}{T_{\text{parallel}}} \]
Scalability

• In general, a problem is *scalable* if it can handle ever increasing problem sizes.

• If we increase the number of threads and keep the efficiency fixed without increasing problem size, the problem is *strongly scalable*.

• If we keep the efficiency fixed by increasing the problem size at the same rate as we increase the number of threads, the problem is *weakly scalable*. 
POSIX Threads

- **Portable Operating System Interface**
- Is an IEEE standard
- API
- Maintain compatibilities among OSes
- Pthreads → a POSIX standard for threads
POSIX Threads (Pthreads)

- Low-level threading libraries
- Native threading interface for Linux now
- Use kernel-level thread (1:1 model)
  - starting from kernel 2.6
- developed by the IEEE committees in charge of specifying a Portable Operating System Interface (POSIX)
- Shared memory
POSIX Threads (Pthreads)

• Because threads within the same process share resources:
  – Changes made by one thread to shared system resources (such as closing a file) will be seen by all other threads
  – Two pointers having the same value point to the same data
  – Reading and writing to the same memory locations is possible
  – Therefore requires explicit synchronization by the programmer
POSIX Threads (Pthreads)

- C language programming types and procedure calls
- implemented with a `pthread.h` header
- To compile with GNU compiler, 2 methods:
  - `gcc/g++ programe -lpthread`
  - `gcc/g++ -pthread programe`
- Programmers are responsible for synchronizing access (protecting) globally shared data.
- Capabilities like thread priority are not part of the core Pthreads library.
POSIX Threads (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 */

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 */
Running a Pthreads program

./helloworld <number of threads>

./helloworld 1

Hello from the main thread
Hello from thread 0 of 1

./helloworld 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
Starting the Threads

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

- Opaque
  - The actual data that they store is system-specific.
  - Their data members aren’t directly accessible to user code.

- Pthreads standard guarantees that a pthread_t object does store enough information to uniquely identify the thread with which it’s associated.
int pthread_create (  
    pthread_t* thread_p /* out */ ,  
    const pthread_attr_t* attr_p /* in */ ,  
    void* (*start_routine) ( void * ) /* in */ ,  
    void* arg_p /* in */ ) ;  

If you want to keep the default, just use NULL.  
Allocate before calling.  

Pointer to the argument that should  
be passed to the function start_routine.  

The function that the thread is to run.
Function started by pthread_create

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

• Void* can be cast to any pointer type in C.

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

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

• We call the function `pthread_join` once for each thread.
• Blocking function.

```c
int pthread_join(
    pthread_t thread; /* in */
    void **ret_val; /*out*/
);
```
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 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>

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

\[
y[i] = 0.0;
\]
\[
for \ (j = 0; j < n; j++)
\]
\[
y[i] += A[i][j]*x[j];
\]
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 */
That previous example was easy!

- $A$, $X$, and $Y$ can all be global.
- Threads update *different* elements of $Y$ simultaneously $\rightarrow$ no problem

Let’s see a not so easy one!
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;
A thread function for computing $\pi$

```c
void* Thread_sum(void* rank) {
    long my_rank = (long) rank;
    double factor;
    long long i;
    long long my_n = n/thread_count;
    long long my_first_i = my_n*my_rank;
    long long my_last_i = my_first_i + my_n;

    if (my_first_i % 2 == 0) /* my_first_i is even */
        factor = 1.0;
    else /* my_first_i is odd */
        factor = -1.0;

    for (i = my_first_i; i < my_last_i; i++, factor = -factor) {
        sum += factor/(2*i+1);
    }

    return NULL;
} /* Thread_sum */
```
Using a dual core processor

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(10^5)</td>
<td>(10^6)</td>
<td>(10^7)</td>
<td>(10^8)</td>
</tr>
<tr>
<td>(\pi)</td>
<td>3.14159</td>
<td>3.141593</td>
<td>3.1415927</td>
<td>3.14159265</td>
</tr>
<tr>
<td>1 Thread</td>
<td>3.14158</td>
<td>3.141592</td>
<td>3.1415926</td>
<td>3.14159264</td>
</tr>
<tr>
<td>2 Threads</td>
<td>3.14158</td>
<td>3.141480</td>
<td>3.1413692</td>
<td>3.14164686</td>
</tr>
</tbody>
</table>

Note that as we increase \(n\), the estimate with one thread gets better and better!!

Reason: Race Condition in updating sum by more than one thread.
Possible race condition

Assume two threads are executing the following code, with x initially 0:
\[ y = \text{computer}(\text{my\_rank}) \]
\[ x = x + y; \quad \text{Critical Section} \]

<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></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>
Solution 1: Busy Waiting

flag initialized to 0 by main thread

\[
y = \text{Compute}(\text{my\_rank}) ; \\
\text{while } (\text{flag} \neq \text{my\_rank}) ; \\
x = x + y ; \\
\text{flag}++;
\]

This will ensure no race condition ... But:

- Busy doing nothing
- serialization
- optimizing compilers can mess with it!
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);
        sum += factor/(2*i+1);
        flag = (flag+1) % thread_count;
    }

    return NULL;
} /* Thread_sum */
```

In dual core
with n = $10^8$
serial $\rightarrow$ 2.8s
2 threads $\rightarrow$ 19.5s!
Optimization: Global sum function with critical section after loop

```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;

    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 */
```

In dual core
with n = $10^8$
serial $\rightarrow$ 2.8s
2 threads $\rightarrow$ 1.5s!

But still the thread
is using the CPU doing
nothing $\rightarrow$ busy waiting
is not the ideal solution
for mutual exclusion
Solution 2: Mutexes

- 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.
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 */);
```
Mutexes

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

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

• 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

```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;

    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 $n = 10^8$ 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>
Conclusions

• Processes $\rightarrow$ threads $\rightarrow$ processors
• User-level threads and kernel-level threads are not the same but they have direct relationship
• Pthreads assume shared memory