Xi'an Jiaotong-Liverpool University
西交利物浦大学

DTS202TC Foundation of Parallel Computing

Lecture 3

Instructor: Md Maruf Hasan

Module Leader: Hong-Bin Liu

AIAC, Xi'an Jiaotong-Liverpool University

November 2023

Administration

- Assessment groups have been released on the Learning Mall
- Check and Confirm before the deadline.

Goals for this week

- Profiling tools (Review)
- Parallel Programming with Pthreads.

Outline

Xi'an Jiaotong-Liverpool University
西交利物浦大学

- Profiling Tools
- Pthreads
  - Background
  - Pthreads Basics
    - Our First Pthreads Program
    - Fundamental Concepts and Common Functions
  - Matrix-Vector Multiplication
  - Critical Sections and Synchronization
    - Critical Sections
    - Busy-Waiting
    - Mutex
    - Semaphore
    - Barrier and Condition Variables
  - Thread Safety

Profiling tools (profilers)

Wikipedia: Profiling is a form of dynamic program analysis that measures, for example, the space (memory) or time complexity of a program, the usage of particular instructions, or the frequency and duration of function calls. Most commonly, profiling information serves to aid program optimization, and more specifically, performance engineering.

- Gprof, Xcode Instruments
- mpiP
- HPCToolkit, caliper
- Even, MATLAB got a Profiler too!

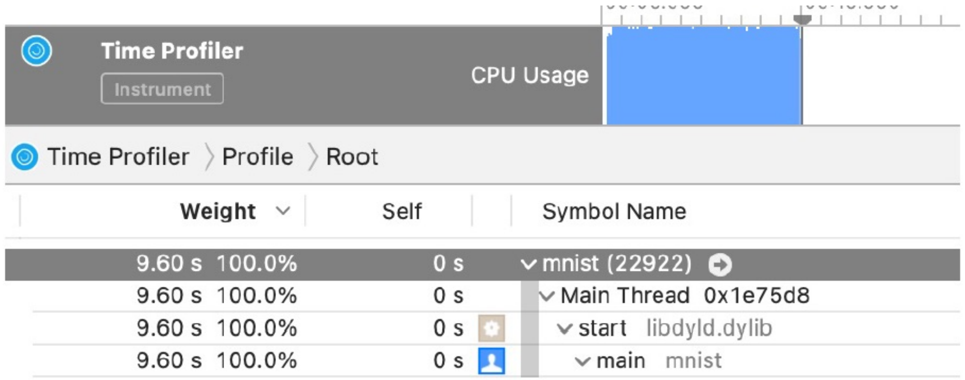


Figure 1: Instruments on Mac

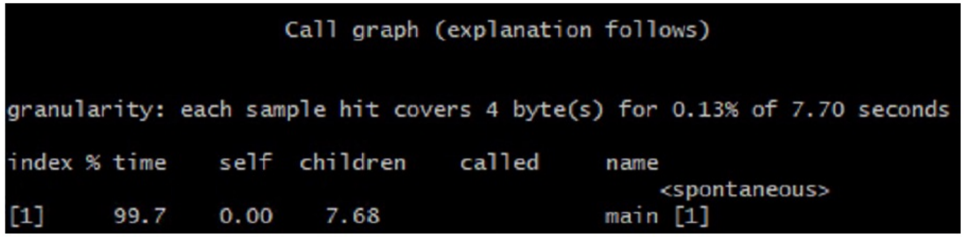


Figure 2: gprof on Cygwin Windows

Gprof Tutorial

GNU gprof Tutorial:
https://www.thegeekstuff.com/2012/08/gprof-tutorial/

How to Profile a C program in Linux using GNU gprof
https://www.maketecheasier.com/profile-c-program-linux-using-gprof/

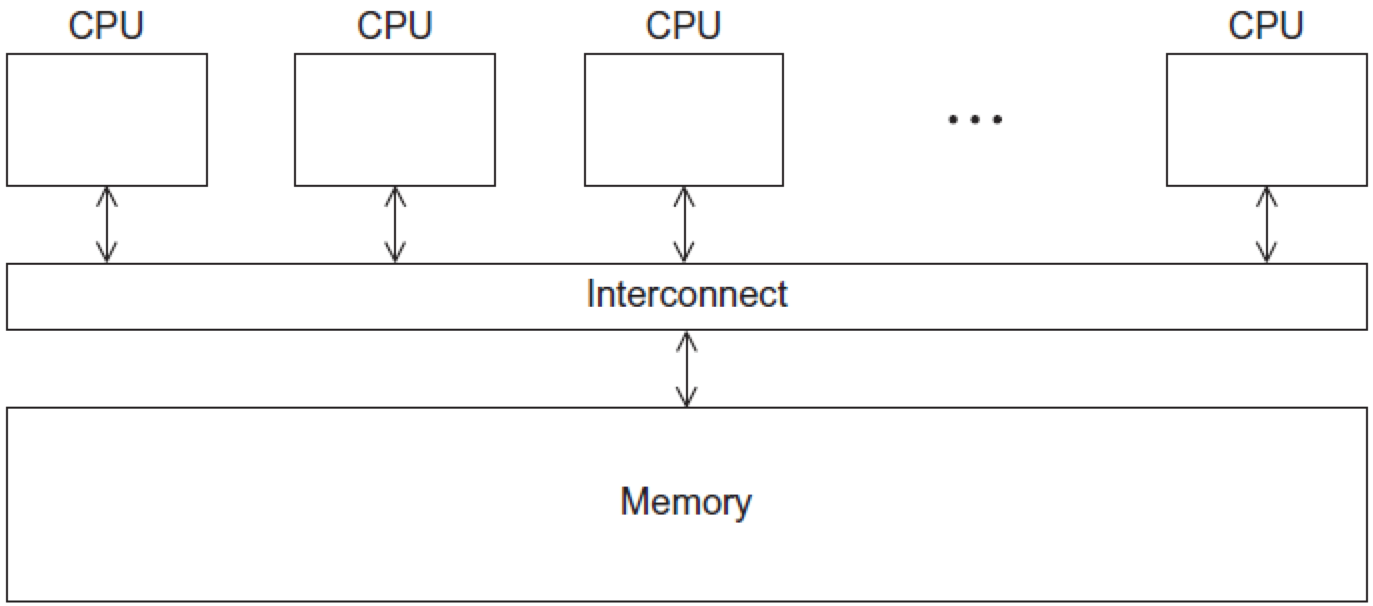
Covered in the Tutorial Session. Important Stuff. Why?
Performance Evaluation, Justification

A profiler is a Microscope for a Programmer!

Outline - Pthreads

- Background
- Pthreads Basics
  - Our First Pthreads Program
  - Fundamental Concepts and Common Functions
- Matrix-Vector Multiplication
- Critical Sections and Synchronization
  - Critical Sections
  - Busy-Waiting
  - Mutex
  - Semaphore
  - Barrier and Condition Variables
- Thread Safety

A Shared Memory System



Threads and Processes

- In shared-memory programming, a process is an instance of a running (or suspended) program.
- Threads are analogous to a "light-weight" process.
- In a shared-memory program, a single process may have multiple threads of control.

POSIX® Threads(Pthreads)

- IEEE had a POSIX 1003 group that defined an interface to multithreaded programming
  - This is called Pthreads (POSIX Thread). Portable
  - A standard for Unix-like operating systems. The Pthreads API is only available on POSIX systems, however, we are using Cygwin on Windows or Unix Virtual Machine.
  - A library that can be linked with C programs.
  - Provides **primitives for thread management and synchronization**

- Threads are peers, no parent/child relationship

Outline

- Background
- Pthreads Basics
  - Our First Pthreads Program
  - Fundamental Concepts and Common Functions
- Matrix-Vector Multiplication
- Critical Sections and Synchronization
  - Critical Sections
  - Busy-Waiting
  - Mutex
  - Semaphore
  - Barrier and Condition Variables
- Thread Safety

Hello World!

Xi'an Jiaotong-Liverpool University
西交利物浦大学

declares the various Pthreads functions, constants, types, etc.

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

global variable

```
/* 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));
```

Hello World! Cont.

Create and start new threads

Join threads

```
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

Xi'an Jiaotong-Liverpool University 西交利物浦大学

```
. /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

- thread_count
- Can introduce subtle and confusing bugs!
- Limit use of global variables to situations in which they' re really needed.
    - Shared variables.

Outline

- Pthreads Basics
  - Our First Pthreads Program
  - Fundamental Concepts and Common Functions
- Matrix-Vector Multiplication
- Critical Sections and Synchronization
  - Critical Sections
  - Busy-Waiting
  - Mutex
  - Semaphore
  - Barrier and Condition Variables
  - Read-Write Locks
- Thread Safety

Starting the Threads

- In Pthreads, the threads are started by the program executable.
  - Include code to explicitly start the threads
  - Need data structures to store threads information

- Processes in MPI are usually started by a script.

Starting the Threads

pthread.h

pthread_t

One object for each 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 */ ) ;
```

The function creates and starts a new 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 <u>before</u> calling.

A Closer Look (2)


```
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*.

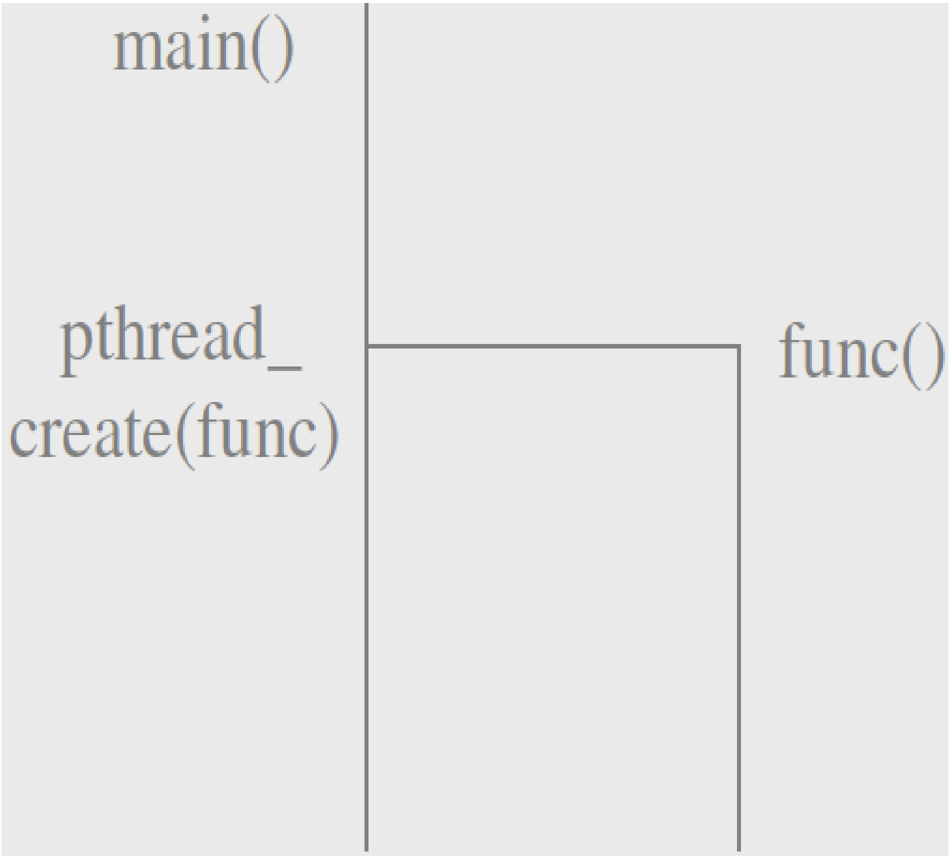
The function that the thread is to run.

Function started by pthread_create

- Prototype:
  `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.

Example of Thread Creation

```
void *func(void *arg) {
int *i=arg;
…..
}
void main()
{
int X; pthread_t id;
….
pthread_create(&id, NULL,
                        func, &X);
…
}
```



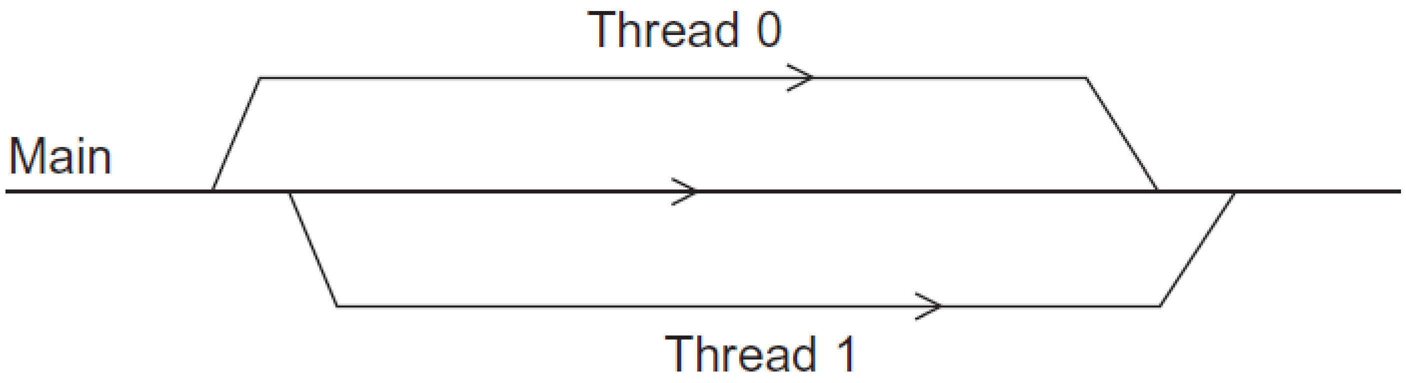
Stopping the Threads

- We call the function pthread_join once for each thread.
- int pthread_join(pthread_t new_id,   /* in */
  void ** ret_val_p   /* out */)
  - Waits for the thread with identifier new_id to terminate, either by returning or by calling pthread_exit()
- A single call to pthread_join will wait for the thread associated with the pthread_t object to complete.

Thread Joining Example

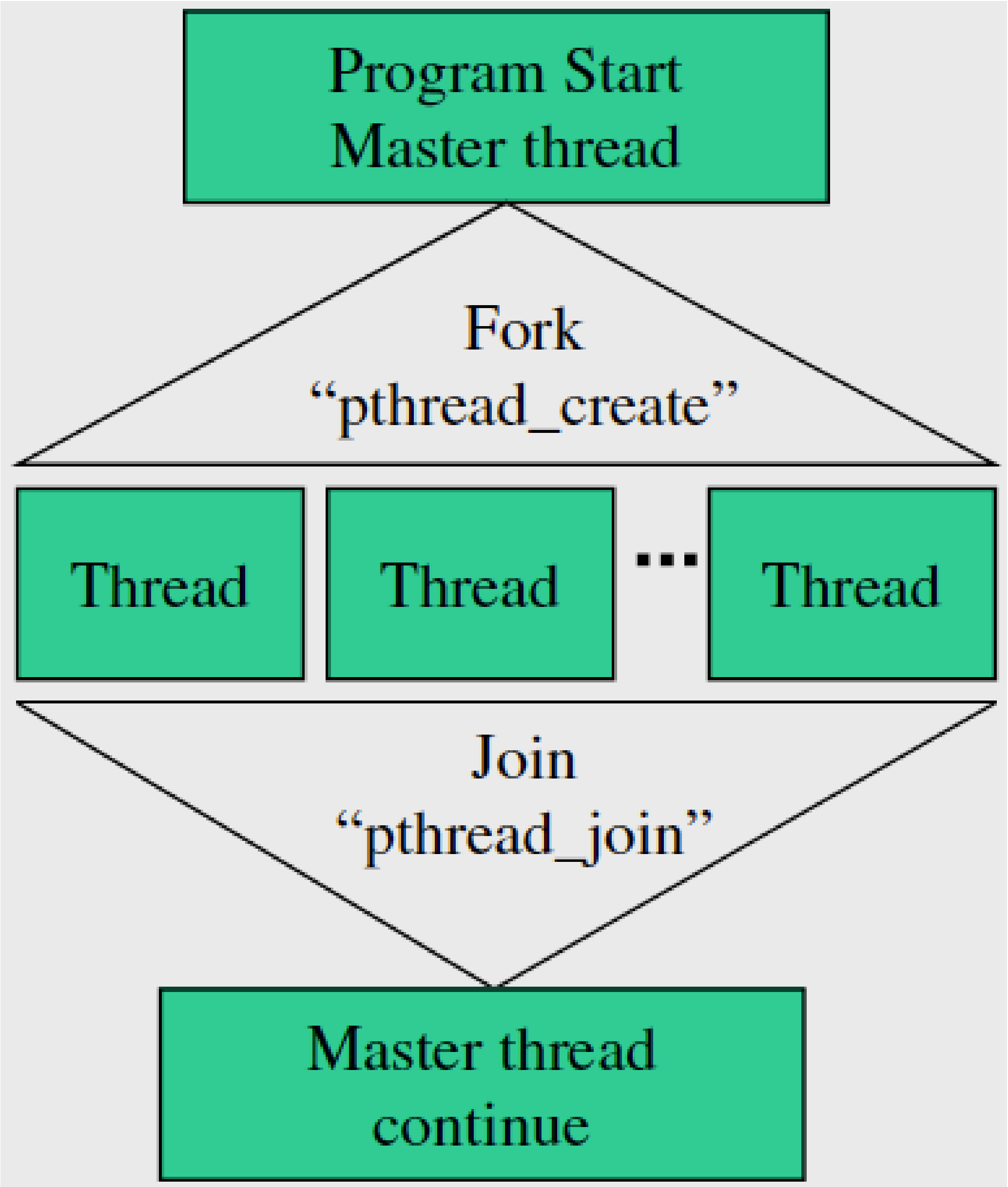

```
void *func(void *) { ….. }
pthread_t id; int X;
pthread_create(&id, NULL, func, &X);
…..
pthread_join(id, NULL);
…..
```

Running the Threads



Main thread forks and joins two threads.

Pthreads Programming Model



Pthread ID


```
pthread_t pthread_self(void);
```

- To determine the thread ID of the calling thread

Outline

- Background
- Pthreads Basics
  - Our First Pthreads Program
  - Fundamental Concepts and Common Functions
- Matrix-Vector Multiplication
- Critical Sections and Synchronization
  - Critical Sections
  - Busy-Waiting
  - Mutex
  - Semaphore
  - Barrier and Condition Variables
- Thread Safety

Matrix-vector Multiplication

$$
\begin{array}{|c|c|c|c|}
\hline
a_{00} & a_{01} & \cdots & a_{0,n-1} \\
\hline
a_{10} & a_{11} & \cdots & a_{1,n-1} \\
\hline
\vdots & \vdots & & \vdots \\
\hline
a_{i0} & a_{i1} & \cdots & a_{i,n-1} \\
\hline
\vdots & \vdots & & \vdots \\
\hline
a_{m-1,0} & a_{m-1,1} & \cdots & a_{m-1,n-1} \\
\hline
\end{array}
\quad
\begin{array}{|c|}
\hline
x_0 \\
x_1 \\
\vdots \\
x_{n-1} \\
\hline
\end{array}
=
\begin{array}{|c|}
\hline
y_0 \\
\hline
y_1 \\
\hline
\vdots \\
\hline
y_i = a_{i0}x_0 + a_{i1}x_1 + \cdots a_{i,n-1}x_{n-1} \\
\hline
\vdots \\
\hline
y_{m-1} \\
\hline
\end{array}
$$

FIGURE 3.11

Matrix-vector multiplication

$$y_i = \sum_{j=0}^{n-1} a_{ij}x_j$$

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];
}
```

Using Three Pthreads

Thread	Components of y
0	y[0], y[1]
1	y[2], y[3]
2	y[4], y[5]

thread 0

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

general case

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

Pthreads Matrix-vector Multiplication


```
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 */
```

Outline

- Background
- Pthreads Basics
  - Our First Pthreads Program
  - Fundamental Concepts and Common Functions
- Matrix-Vector Multiplication
- Critical Sections and Synchronization
  - Critical Sections
  - Busy-Waiting
  - Mutex
  - Semaphore
  - Barrier and Condition Variables
- Thread Safety

Data Race in Pthreads Program


```
static int s = 0;
```

```
Thread 1

  for i = 0, n/2-1
      s = s + f(A[i])
```

```
Thread 2

  for i = n/2, n-1
      s = s + f(A[i])
```

- Problem is a race condition on variable s in the program
- When multiple threads attempt to access a shared resource, at least one of the accesses is an update, and the accesses can result in an error, we have a race condition .

Estimating π

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

```
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 π


```
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

	n			
	10^5	10^6	10^7	10^8
π	3.14159	3.141593	3.1415927	3.14159265
1 Thread	3.14158	3.141592	3.1415926	3.14159264
2 Threads	3.14158	3.141480	3.1413692	3.14164686

- Note that as we increase n, the estimate with one thread gets better and better.
- However, for larger values of n , the result computed by two threads actually gets worse.
- It matters if multiple threads try to update a single shared variable.

Possible race condition

y = Compute(my rank);
x = x + y;
y: private, x: shared, initialized to 0

Time	Thread 0	Thread 1
1	Started by main thread	
2	Call `Compute()`	Started by main thread
3	Assign `y = 1`	Call `Compute()`
4	Put `x=0` and `y=1` into registers	Assign `y = 2`
5	Add 0 and 1	Put `x=0` and `y=2` into registers
6	Store 1 in memory location x	Add 0 and 2
7		Store 2 in memory location x

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.

Critical Sections

- Critical Section is a block of code that updates a shared resources that can only be updated by one thread a time.

Outline

Xi'an Jiaotong-Liverpool University
西交利物浦大学

- Background
- Pthreads Basics
  - Our First Pthreads Program
  - Fundamental Concepts and Common Functions
- Matrix-Vector Multiplication
- Critical Sections and Synchronization
  - Critical Sections
  - Busy-Waiting
  - Mutex
  - Semaphore
  - Barrier and Condition Variables
  - Read-Write Locks
- Thread Safety

Pthreads Synchronization

- Create/exit/join
  - provide some form of synchronization,
  - at a very coarse level,
  - requires thread creation/destruction.
- Need for finer-grain synchronization
  - mutex locks
  - condition variables
  - ….
- PTHREADS provides a variety of synchronization facilities for threads to cooperate in accessing shared resources.

Busy-Waiting


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

flag initialized to 0 by main thread

- Busy-waiting is not an ideal solution to the problem of controlling access to a critical section.
- Since thread 1 will execute the test over and over until thread 0 executes flag++ , if thread 0 is delayed, thread 1 will simply "spin" on the test, eating up CPU cycles. This can be positively disastrous for performance.

Busy-Waiting

- A thread repeatedly tests a condition, but, effectively, does no useful work until the condition has the appropriate value.
- The busy-wait solution would work "provided the statements are executed exactly as they're written."
- Beware of optimizing compilers, though!

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

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

- Turning off compiler optimizations can seriously degrade performance.

Pthreads Global Sum with Busy-waiting

Xi'an Jiaotong-Liverpool University 西交利物浦大学

```
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 */
```

When n=10^8, the serial sum is consistently faster than the parallel sum.

Global Sum Function with Critical Section after Loop


```
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 */
```

Outline

- Background
- Pthreads Basics
  - Our First Pthreads Program
  - Fundamental Concepts and Common Functions
- Matrix-Vector Multiplication
- Critical Sections and Synchronization
  - Critical Sections
  - Busy-Waiting
  - Mutex
  - Semaphore
  - Barrier and Condition Variables
  - Read-Write Locks
- Thread Safety

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.
- Used to guarantee that one thread "excludes" all other threads while it executes the critical section.

Mutexes

- The Pthreads standard includes a special type for mutexes: pthread_mutex_t. Need to be initialized by the system before it's used.

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

- Creates a new mutex lock.
- Attribute: normal, recursive, errorcheck

(Igonre the second argument in this book, just pass NULL.)

Mutexes

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

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

- In order to gain access to a critical section a thread calls pthread_mutex_lock

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

  – Tries to acquire the lock specified by mutex.
  – If mutex is already locked, then calling thread blocks until mutex is unlocked.

Mutexes

- When a thread is finished executing the code in a critical section, it should call pthread_mutex_unlock

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

  - If calling thread has mutex currently locked, this will unlock the mutex.
  - If other threads are blocked waiting on this mutex, one will unblock and acquire mutex, which one is determined by the scheduler.

Global Sum Function that Uses a Mutex


```
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 */
```

Outline

- Background
- Pthreads Basics
  - Our First Pthreads Program
  - Fundamental Concepts and Common Functions
- Matrix-Vector Multiplication
- Critical Sections and Synchronization
  - Critical Sections
  - Busy-Waiting
  - Mutex
  - Semaphore
  - Barrier and Condition Variables
  - Read-Write Locks
- Thread Safety

Some Issues

- 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.

Problems with a Mutex Solution


```
/* 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 */
```

A First Attempt at Sending Messages Using Pthreads


```
/* 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 */
```

No receiving message for multithreads

Possible Solutions with Problems

S1: Change if statement to while statement:

```
–if (messages[my_rank] != NULL)
        printf("Thread %ld  > %s\n", my_rank, messages[my_rank]);
–while (messages[my_rank] = NULL);
        printf("Thread %ld  > %s\n", my_rank, messages[my_rank]);
```

- However, there is a busy-waiting problem.

- S2: Mutex Lock:

```
…
Pthread_mutex_lock(mutex[dest]);
…
Messages[dest]=msg;
Pthread_mutex_unlock(mutex[dest]);
…
Pthread_mutex_lock(mutex[myrank]);
printf("Thread %ld  > %s\n", my_rank, messages[my_rank]);
…
```

- However, it still will crash.

Semaphore

- Semaphores can be thought of as a special type of unsigned int , so they can take on the values 0, 1, 2, ... When they take on the values 0 and 1, called a binary semaphore.
- For binary semaphore, 0 corresponds to a locked mutex, and 1 corresponds to an unlocked mutex.
  - Initialized to 1(unlocked).
  - Before the critical, call sem_wait . Executing sem_wait will block if the semaphore is 0. If the semaphore is nonzero, it will decrement the semaphore and proceed.
  - After executing the critical, call sem_post , which increments the semaphore, and a thread waiting in sem_wait can proceed.

Syntax of the Various Semaphore Functions

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

```
#include <semaphore.h>
```

Initialize the semaphore descriptor.

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

Unlock a semaphore.

Destroy a semaphore.

```
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 */);
```

Lock a semaphore.

Send Messages Using Semaphores


```
/* messages is allocated and initialized to NULL in main   */
/* semaphores is allocated and initialized to 0 (locked) in
      main */
void* Send_msg(void* rank) {
   long my_rank = (long) rank;
   long dest = (my_rank + 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;
   sem_post(&semaphores[dest])
         /* ''Unlock'' the semaphore of dest */

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

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

- **The message-sending problem didn't involve a critical section.**
- **Thread my_rank couldn't proceed until thread source had finished creating the message.**
- **When a thread can't proceed until another thread has taken some action, is sometimes called producer-consumer synchronization.**

Outline

- Background
- Pthreads Basics
  - Our First Pthreads Program
  - Fundamental Concepts and Common Functions
- Matrix-Vector Multiplication
- Critical Sections and Synchronization
  - Critical Sections
  - Busy-Waiting
  - Mutex
  - Semaphore
  - Barrier and Condition Variables
- Thread Safety

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.

Using Barriers to Time the Slowest Thread


```
/* 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 busy-wait loop.

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);
   . . .
}
```

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

Implementing a Barrier with Semaphores

protects the counter

block threads that have entered the barrier

```
/* 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);
   }
   . . .
```

Reusing barrier_sem results in a race condition.

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;
```

Condition Variables

- Condition variables in Pthreads have type pthread_cond_t.

```
int pthread_cond_init(
      pthread_cond_t*             cond_p      /* out */,
      const pthread_condattr_t*   cond_attr_p /* in  */);
```

- Creates a new condition variable cond.
- Attribute: ignore for now.

```
int pthread_cond_destroy(pthread_cond_t*  cond_p  /* in/out */);
```

- Destroys the condition variable cond.

Condition Variables

Xi'an Jiaotong-Liverpool University 西交利物浦大学

```
int pthread_cond_signal(pthread_cond_t*  cond_var_p /* in/out */);
```

- Unblock one of the blocked threads waiting on cond.
- Which one is determined by scheduler.
- If no thread waiting, then signal is a no-op.

```
int pthread_cond_broadcast(pthread_cond_t* cond_var_p /* in/out */);
```

- Unblock all of the blocked threads waiting on cond.
- If no thread waiting, then signal is a no-op.

Condition Variables


```
int pthread_cond_wait(
      pthread_cond_t*     cond_var_p    /* in/out */,
      pthread_mutex_t*    mutex_p       /* in/out */);
```

- Blocks the calling thread, waiting on cond.
- Unlocks the mutex.

```
pthread_mutex_unlock(&mutex_p);
wait_on_signal(&cond_var_p);
pthread_mutex_lock(&mutex_p);
```

Implementing a Barrier with Condition Variables


```
/* 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);
    . . .
}
```

Outline

- Background
- Pthreads Basics
  - Our First Pthreads Program
  - Fundamental Concepts and Common Functions
- Matrix-Vector Multiplication
- Critical Sections and Synchronization
  - Critical Sections
  - Busy-Waiting
  - Mutex
  - Semaphore
  - Barrier and Condition Variables
- Thread Safety

Thread-Safety

- A block of code is thread-safe if it can be simultaneously executed by multiple threads without causing problems.

Example

- Suppose we want to use multiple threads to "tokenize" a file that consists of ordinary English text.
- The tokens are just contiguous sequences of characters separated from the rest of the text by white-space — a space, a tab, or a newline.

Simple Approach

- Divide the input file into lines of text and assign the lines to the threads in a round-robin fashion.
- The first line goes to thread 0, the second goes to thread 1, . . . , the tth goes to thread t, the t +1st goes to thread 0, etc.
- We can serialize access to the lines of input using semaphores.
- After a thread has read a single line of input, it can tokenize the line using the strtok function.

The Strtok Function


```
char* strtok(
      char*          string      /* in/out */,
      const char*    separators  /* in     */);
```

- The idea is that in the first call, strtok caches a pointer to string, and for subsequent calls it returns successive tokens taken from the cached copy.
- The first time it' s called the string argument should be the text to be tokenized.(One line of input.)
- For subsequent calls, the first argument should be NULL.

Multi-threaded Tokenizer


```
void* Tokenize(void* rank) {
   long my_rank = (long) rank;
   int count;
   int next = (my_rank + 1) % thread_count;
   char *fg_rv;
   char my_line[MAX];
   char *my_string;

   sem_wait(&sems[my_rank]);
   fg_rv = fgets(my_line, MAX, stdin);
   sem_post(&sems[next]);
   while (fg_rv != NULL) {
      printf("Thread %ld > my line = %s", my_rank, my_line);

      count = 0;
      my_string = strtok(my_line, " \t\n");
      while ( my_string != NULL ) {
         count++;
         printf("Thread %ld > string %d = %s\n", my_rank, count,
               my_string);
         my_string = strtok(NULL, " \t\n");
      }

      sem_wait(&sems[my_rank]);
      fg_rv = fgets(my_line, MAX, stdin);
      sem_post(&sems[next]);
   }

   return NULL;
}  /* Tokenize */
```

Running with One Thread

- It correctly tokenizes the input stream.

Pease porridge hot.
Pease porridge cold.
Pease porridge in the pot
Nine days old.

Running with Two Threads


```
Thread 0 > my line = Pease porridge hot.
Thread 0 > string 1 = Pease
Thread 0 > string 2 = porridge
Thread 0 > string 3 = hot.
Thread 1 > my line = Pease porridge cold.
Thread 0 > my line = Pease porridge in the pot
Thread 0 > string 1 = Pease
Thread 0 > string 2 = porridge
Thread 0 > string 3 = in
Thread 0 > string 4 = the
Thread 0 > string 5 = pot
Thread 1 > string 1 = Pease
Thread 1 > my line = Nine days old.
Thread 1 > string 1 = Nine
Thread 1 > string 2 = days
Thread 1 > string 3 = old.
```

Oops!

What happened?

Xi'an Jiaotong-Liverpool University 西交利物浦大学

- strtok caches the input line by declaring a variable to have static storage class.
- This causes the value stored in this variable to persist from one call to the next.
- Unfortunately for us, this cached string is shared, not private.
- Thus, thread 0' s call to strtok with the third line of the input has apparently overwritten the contents of thread 1' s call with the second line.
- So the strtok function is not thread-safe. If multiple threads call it simultaneously, the
output may not be correct.

Other Unsafe C Library Functions

- Regrettably, it’ s not uncommon for C library functions to fail to be thread-safe.
- The random number generator random in stdlib.h.
- The time conversion function localtime in time.h.

“re-entrant”(Thread Safe) Functions

- In some cases, the C standard specifies an alternate, thread-safe, version of a function.

```
char* strtok_r(
      char*         string       /* in/out */,
      const char*   separators,  /* in     */
      char**        saveptr_p    /* in/out */);
```

Caveat

Incorrect programs can produce correct output!

Summary

- Background
- Pthreads Basics
- Matrix-Vector Multiplication
- Critical Sections and Synchronization
  - Critical Sections
  - Busy-Waiting
  - Mutex
  - Semaphore
  - Barrier and Condition Variables
- Thread Safety