CS4328 – Operating Systems
Program Assignment #1 (Total Points: 150)

Due 1 second before class begins on Oct. 23, 2013

Overview

The purpose of this project is to practice multithreaded programming using Pthreads, Win32 Thread and OpenMP. There are three independent parts of this project. Please read instructions carefully.

You must submit your project in one zip file (named as firstname_lastname_proj1) to the Project Turn In Site (https://hwupload.cs.txstate.edu/), which should include all your source code for part1, part2, part 3 and the according doc file for part 2 and part3 questions. You only need to submit a hard copy of the doc file in class on Oct. 23. Do not print out the code.

The submission system will automatically record your submission time, which will be used to decide whether or not you miss the deadline. There will be a penalty of 20% per day (including weekend and holiday) for late submission. The late submission after two days (including weekend and holiday) will not be accepted. Therefore, DO NOT DELAY. Start writing the program from Day 1. If you wait until the night before the due date, you will have a miserable night and it is less likely you could complete the project.

You are encouraged to discuss with other students but you have to **DO YOUR OWN WORK**. You may get ***ZERO*** for this project or ***FAIL*** the class if you copy code from others or allow others to copy your code.

Part 1: Implement Fibonacci Sequence using Pthreads and Win32 Thread (50 points).

The Fibonacci sequence is the series of numbers 0, 1, 1, 2, 3, 5, 8, …. Formally, it can be expressed as:

$$F_n = \begin{cases} 0 & \text{if } n = 0; \\ 1 & \text{if } n = 1; \\ F_{n-1} + F_{n-2} & \text{if } n > 1. \end{cases}$$

That is, after two starting values, each number is the sum of the two preceding numbers. The Fibonacci numbers for n = 0, 1, 2, … ,20 are:

F_0	F_1	F_2	F_3	F_4	F_5	F_6	F_7	F_8	F_9	F_{10}	F_{11}	F_{12}	F_{13}	F_{14}	F_{15}	F_{16}	F_{17}	F_{18}	F_{19}	F_{20}
0	1	1	2	3	5	8	13	21	34	55	89	144	233	377	610	987	1597	2584	4181	6765

Part 1a (25 points): Write a **multithreaded** program that generates the Fibonacci sequence using Pthreads. This program should work as follows: The user will enter on the command line the number of Fibonacci numbers that the program is to generate. The program will then create a separate thread that will generate the Fibonacci numbers, placing the sequence in data that can be shared by the threads (an array is probably the most convenient data structure). When the thread finishes execution, the parent thread will output the sequence generated by the child thread. Because the parent thread cannot begin

outputting the Fibonacci sequence until the child thread finishes, this will require having the parent thread wait for the child thread to finish, using the ***thread join*** techniques described in class.

Part 1b (25 points): Write a **multithreaded** program that does the same Fibonacci calculate as stated in Part 1a using Win32 thread.

Part 2 a: Implement Matrix Multiplication using Pthreads (30 points)

Background of Matrix Multiplication

Given two matrices, *A* and *B*, where matrix *A* contains *M* rows and *K* columns and matrix *B* contains *K* rows and *N* columns, the **matrix product** of *A* and *B* is matrix *C*, where *C* contains *M* rows and *N* columns. The entry in matrix *C* for row *i*, column *j*, denoted as $C_{i,j}$, is the sum of the products of the elements for row *i* in matrix *A* and column *j* in matrix *B*. That is,

$$C_{i,j} = \sum_{n=1}^{K} (A_{i,n} \times B_{n,j})$$

For example, if *A* is a 2-by-3 matrix and *B* is a 3-by-2 matrix, the result matrix will be a 2-by-2 matrix.

$$\begin{bmatrix} 1 & 0 & 2 \\ -1 & 3 & 1 \end{bmatrix} \begin{bmatrix} 3 & 1 \\ 2 & 1 \\ 1 & 0 \end{bmatrix} = \begin{bmatrix} 1 \times 3 + 0 \times 2 + 2 \times 1 & 1 \times 1 + 0 \times 1 + 2 \times 0 \\ -1 \times 3 + 3 \times 2 + 1 \times 1 & -1 \times 1 + 3 \times 1 + 1 \times 0 \end{bmatrix} = \begin{bmatrix} 5 & 1 \\ 4 & 2 \end{bmatrix}$$

Passing Parameters to Each Thread

The parent thread will create *M* × *N* worker threads, passing each worker the values of row *i* and column *j* that it is to use in calculating the matrix product. This requires passing two parameters to each thread. The easiest approach with Pthreads is to create a data structure using a struct. The members of this structure are *i* and *j*, and the structure appears as follows:

```
/* structure for passing data to threads */

Struct v
{
    int i ; /* row */
    int j ; /* column */
};
```

The Pthreads program will create the worker threads using a strategy similar to that shown below:

```
/* We have to create M × N worker threads */

for ( i = 0;  i < M, i++)
  for ( j = 0;  j < N; j++ )  {
      struct v *data = (struct v *) malloc (sizeof (struct v)) ;
      data->i =  i
      data->j =  j
```

```
        /* Now create the thread passing its data as a parameter */
    }
}
```

The data pointer will be passed to the pthread_create (), which in turn will pass it as a parameter to the function that is to run as a separate thread.

Waiting for Threads to Complete

Once all worker threads have completed, the main thread will output the product contained in matrix C. This requires the main thread to wait for all worker threads to finish before it can output the value of the matrix product. Several different strategies can be used to enable a thread to wait for other threads to finish. In this example, the main thread has to wait for multiple threads to finish before it can continue its execution.

The following code shows how you could join on ten threads using the Pthreads:

```
#define NUM_THREADS 10

pthread_t workers[NUM_THREADS];

for (int i = 0; i < NUM_THREADS; i++)

    pthread_join(workers[i], NULL);
```

For this project, you need to calculate each element $C_{i,j}$ in a separate worker thread. This will involve creating M × N worker threads. The main (or parent) thread will initialize the matrices A and B and allocate sufficient memory for matrix C, which will hold the product of matrices A and B. These matrices will be declared as global data so that each worker thread has access to A, B, and C.
Matrices A and B can be initialized statically, as shown below:

```
#define M 3
#define K 2
#define N 3
int A [M] [K] = {{1,4}, {2,5}, {3,6}};
int B [K] [N] =  {{8,7,6}, {5,4,3}};
int C [M]  [N] ;
```

Your code should be able to calculate the C = A × B and print out the correct matrix C. I may change the elements value of A and B to test your code but you can assume I will not change the value of M, K, and N.

Part 2b: Answer the following questions (20 points)

Now you have generated a multithreaded program which can multiply a 3-by-2 matrix and a 2-by-3 matrix. Please answer the following questions:

1. We can also write a program to do matrix multiplication without using multithreading. What are the advantages and disadvantages of using multithreading? (5 points)

2. Is your code still applicable for multiplying two 100-by-100 matrices? If yes, why? If no, how will you change your current code to make it applicable? You only need to write down the key changes and you can use pseudo code. (5 points)
3. Is your code efficient in the scenario of multiplying giant matrices, say one million by one million matrices? If yes, why? If not, what are the potential performance problems? (Hint: think about the overhead of creating thread and data locality issue). How to solve these problems? (10 points)

Part 3: How many 99s (Pthreads and OpenMP)? (50 points).

The following sequential code will count how many 99s in an array.

```
#include <omp.h>
#include <iostream>
#include <ctime>
using namespace std;

int main()
{
        int count = 0, i;
        int const size = 100000;
        int myArray [size];
        double start_time, end_time;

        //initialize random number generator
        srand((unsigned)time(NULL));

        // Initialize the array using random numbers
        for (i = 0; i < size; i++)
                myArray[i] = rand() % 100;

        //Serial Code
        start_time = omp_get_wtime();
        for (i = 0; i < size; i++)
                if (myArray[i] == 99)
                        count ++;

        end_time = omp_get_wtime();

        printf ("The serial code indicates that there are %d 99s in the array.\n\n", count);
        printf ("The serial code used %f seconds to complete the execution.\n\n", end_time - start_time);

        return 0;
}
```

Part 3a (20 points): Write the correct and efficient multithreading code using OpenMP. Record the execution time of your parallel code and calculate the speedup when using 2 threads, 4 threads and 8 threads. Your score will be based on the correctness and speedup of your parallel code.

Part 3b (30 points): Write the correct and efficient multithreading code using Pthreads. Record the execution time of your parallel code and calculate the speedup when using 2 threads, 4 threads and 8 threads. Your score will be based on the correctness and speedup of your parallel code.