**Parallel Programming with POSIX Threads**

Newton's Pi approximation formula can be written as:

**pi = 4 [ 1 - 1/3 + 1/5 - 1/7 + 1/9 ... +((-1)^n)/(2n+1) ]**

Write a C program using pthreads that calculates in parallel the nth (nth product) approximation of PI using Newton's formula, using m threads and j number of products to iterate; where each thread computes a different set of products.

Code stub:

#include <pthread.h>   
#include <stdio.h>  
  
/\* this data is shared by the thread(s) \*/  
int threads;

unsigned long long iterations;

double \* pi;

void \* runner(void \* param); /\* the thread \*/  
  
int main(int argc, char \* argv[]) {

        if (argc != 3) {  
            fprintf(stderr, "usage: a.out <iterations> <threads>\n");  
            /\*exit(1);\*/  
            return -1;  
        }  
        if (atoi(argv[1]) < 0 || atoi(argv[2]) < 0) {  
            fprintf(stderr, "Arguments must be non-negative\n ");  
                /\*exit(1);\*/  
                return -1;  
            }

...

        /\* create the thread identifiers \*/  
 ...  
        /\* create set of attributes for the thread \*/  
 ...

/\* populate variables... \*/  
 ...

            /\* get the default attributes \*/  
            ...  
            /\* create threads \*/  
            ...  
            /\* now wait for the threads to exit \*/  
            ...

/\* compute and print results \*/  
            ...

...printf("pi = %.15f\n",...

        }  
        /\*\*  
         \* The thread will begin control in this function  
         \*/  
        void \* runner(void \* param) {  
            int threadid=...  
            pi[threadid] = 1;

            //complete function

            pthread\_exit(0);  
        }

Compile with: gcc -o pi pi.c -lpthread

Useful references: <http://www.yolinux.com/TUTORIALS/LinuxTutorialPosixThreads.html>

Arguments: Iterations, threads.

Note: Main program thread + runner threads, you are not required to use synchronization.

# Exercise 1

## 20/20

Answer the following:

1. Describe how you are going to parallel the series formula.
2. Describe, briefly, how your implementation will compute the nth Pi approximation in parallel. (thread deployment, work per thread and computation of final results)
3. Describe, in detail, how each runner function will compute its part of the series, present an example pseudocode.
4. Describe how you are going to handle/avoid race conditions and how the use of shared memory might aid you on this issue.

# Exercise 2

## 20/20

Paste your parallel-computed implementation of the Newtonws' Pi approximation using pThreads below.

Paste the output of running your implementation with 10000 iterations and 4 threads.

# Exercise 3

## 20/20

**Process Synchronization**

Implement the producer-consumer problem using a bounded-buffer and pthreads that manages race conditions using **pthreads mutex locks.**

Your program will run **n** producers and **m** consumers at the same time.

Use the following non-synchronized implementation as a starting point. Beware of the indexes and conditional variables. You will need to implement two indexes, one for all consumers and one for all producers. Your insert/remove conditionals (or busy waits) must take into consideration both indexes. This will convert the buffer into a circular buffer thus the indexes will only be incremented (use modulus etc) This implementation will not run correctly as it does not handle race conditions. To test your implementation you can try using 2 producers and 1 consumer.

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

#define BUFFER\_SIZE 20

int count = 0;  
int buffer [BUFFER\_SIZE];

pthread\_t tid;

void insert(int item){

while (count == BUFFER\_SIZE);

buffer[count] = item;

count++;

sleep(1);

}

int remove\_item(){

int item;

while (count == 0);

item = buffer[count];

count--;

sleep(1);

return item;

}

void \* producer(void \* param){  
   int item;  
   while(1){  
      item = rand() % BUFFER\_SIZE ;  
      insert(item);  
      printf("inserted: %d\n", item);  
   }  
}

void \* consumer(void \* param){

int item;

while(1){

item = remove\_item();

printf("removed: %d\n", item);

}

}

int main(int argc, char \* argv[])  
{  
    int producers = atoi(argv[1]);  
    int consumers = atoi(argv[2]);  
    int i;

    for (i = 0; i < producers; i++)  
       pthread\_create(&tid, NULL, producer,NULL);

    for (i = 0; i < consumers; i++)  
       pthread\_create(&tid, NULL, consumer, NULL);

    pthread\_join(tid,NULL);  
}

A correct implementation will run like this:

./ProducerConsumer\_mutex 2 1

in: 1 inserted: 3

in: 2 inserted: 6

out: 1 removed: 3

out: 2 removed: 6

in: 3 inserted: 17

in: 4 inserted: 15

out: 3 removed: 17

in: 5 inserted: 13

in: 6 inserted: 15

in: 7 inserted: 6

out: 4 removed: 15

in: 8 inserted: 12

in: 9 inserted: 9

out: 5 removed: 13

in: 10 inserted: 1

Paste your implementation below.

# Exercise 4

## 20/20

Modify your producer-consumer implementation so that it uses **semaphores** to handle race conditions instead of mutexes.

Use the pthread library implementation.

Paste your code below.

# Exercise 5

## 20/20

Modify your producer-consumer implementation so that it uses **monitors** to handle race conditions instead of semaphores or mutexes.

Use the pthread library implementation

Useful resources:

<https://www.cs.cmu.edu/afs/cs/academic/class/15492-f07/www/pthreads.html#SYNCHRONIZATION>

<http://www.yolinux.com/TUTORIALS/LinuxTutorialPosixThreads.html>

Paste your code below.