Summary of PThreads Labs

```
void* threadTask(void* arg) {
  //implement what you want a thread to do
    return NULL; //if you don't want a thread to return something
}
```

How to create a Pthread

1. pthread_t threadName; //declare a thread
2. int pthread_create(pthread_t* *threadAdress*, const pthread_attr_t* *threadAttribute*, void* *(*threadTask)*, void* *arg*);

Where:

threadAddress: the location where the address of the newly created thread should be stored, or NULL if the thread ID is not required.

threadAttribute: the thread attribute object specifying the attributes for the thread that is being created.(security attributes)
(If threadAttribute is NULL, the thread is created with default attribute).

threadTask: the main function for the thread; the thread begins executing user code at this address (task of the thread)

arg: the argument passed to threadTask. (arg of type void pointer)
(A void pointer: is a pointer that has no associated data type with it. A void pointer can hold address of any type)

On success: pthread_create() returns 0.
On error: one of the following integer values is returned: EAGAIN, EFAULT, EINVAL. (search about these values!)

How to create an attribute for a Pthread:

1. pthread_attr_t attributeName; //declare an attribute
2. int pthread_attr_init(pthread_attr_t* *attributeAddress*); //initiate the attribute
   Where:
   attributeAddress: the thread attribute object to be initialized.
3. int pthread_attr_setdetachstate(pthread_attr_t* *attributeAddress*, int *detachstate*);
   Where:
   detachstate: the thread detached state attribute **value**
4. int pthread_attr_destroy(pthread_attr_t* *attributeAddress*)
   //free the attribute variable from memory

The thread's detached state determines whether another thread may wait for the termination of the

PTHREAD_CREATE_DETACHED
creates a new detached thread.
pthread_join() can't wait for a detached thread.

PTHREAD_CREATE_JOINABLE
creates a new non-detached thread.
pthread_join() must be called to release any resources associated with the terminated thread.

And then you pass the *attributeAddress* which is (&*attributeName*) as the *threadAttribute* parameter in the pthread_create() function
Note that: the default attribute of a thread is JOINABLE

How to make a thread join the calling thread (the thread that is currently working and called the function):

int pthread_join(pthread_t *targetThread*, void** *status*);

suspends execution of the calling thread until the targetThread terminates, unless the targetThread has already terminated.

Where:
targetThread: the thread to wait for.
status: the location where the exit status of the joined thread is stored. This can be set to **NULL** if the exit status is not required.

On success: pthread_join() returns 0.
On error: one of the following integer values is returned: EDEADLK, EINVAL, ESRCH, EFAULT. (search about these values!)

How to create and use mutex:

1. pthread_mutex_t mutexName; //declare a mutex variable (GLOBALLY)
2. int pthread_mutex_init(pthread_mutex_t* *mutexAddress*, const pthread_mutexattr_t* *attr*); //initialize the mutex

   Where:
   mutexAddress: the location of the mutex to be initialized
   attr: specifies the attributes to use to initialize the mutex, or NULL if default attributes should be used.

3. int pthread_mutex_lock(pthread_mutex_t* *mutexAddress*);

   locks the variables inside the **lock/unlock block** (and then the calling thread is called **THE OWNING THREAD**). (if it's already locked by another thread then it will wait until the owning thread Unlocks it)

   The **lock/unlock block** is the block between pthread_mutex_lock() and pthread_mutex_unlock()

4. int pthread_mutex_unlock(pthread_mutex_t* *mutexAddress*);

   unlocks the variables inside the **lock/unlock block IF CALLED BY THE OWNING THREAD**.
   (if it's not owned by another thread OR it's already unlocked then an error occurs)

5. int pthread_mutex_destroy(pthread_mutex_t* *mutexAddress*)
   //free the mutex from memory

How to terminate a thread:

Two cases (of many cases) when the thread we created to do some task is terminated are:

1. the thread has finished its task, or
2. the Master Thread (The Master Thread: is the one that created the other threads (in our case is the Main Fucntion)) is has finished its work, so it will terminate all the threads it created.

So, if we want our thread to *only* terminate when it finishes its work *even if* the Master thread has finished its work, we make our thread call the function pthread_exit().

void pthread_exit(void* *status*);

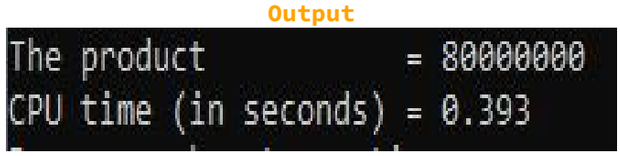
Where:
status: the exit status for the thread. This can be set to **NULL** if the exit status is not required.

Vector-Vector Dot Product Program

Serial (Sequential) Code

```
#include<iostream>
#include<iomanip>
#include<ctime>
using namespace std;

struct dotProductData {
  double* a; //first vector
  double* b; //second vector
  double sum; //sum of the multiplied corresponding numbers of the two vectors
  long vecLength; //vector length
}dotStr;

const long VecLen = 80000000;

void dotProduct() {
  double *x, *y;
  x = dotStr.a;
  y = dotStr.b;

  for (long i = 0; i < dotStr.vecLength; i++) {
    dotStr.sum += x[i] * y[i];
  }
}


void main() {

  dotStr.vecLength = VecLen;

  dotStr.a = new double[VecLen];
  dotStr.b = new double[VecLen];

  for (long i = 0; i < VecLen; i++) {
    dotStr.a[i] = 1;
    dotStr.b[i] = 1;
  }

  clock_t t1,t2;

  t1 = clock();

  dotProduct();

  t2 = clock();

  cout<<setprecision(15);
  cout << "The product           = " << dotStr.sum << endl;
  cout << "CPU time (in seconds) = " << (double)(t2-t1) / CLOCKS_PER_SEC << endl;
  system("pause");
}
```

Output



Multi-threading (Using Pthreads) Code

```
#include<iostream>
#include<iomanip>
#include<pthread.h>
#include<string>
#include<ctime>
using namespace std;

const long VecLen = 80000000;

struct dotProductData{
  double* a; //first vector
  double* b; //second vector
  double sum; //sum of the multiplied corresponding numbers of the two vectors
  long vecLength; //vector length
  dotProductData(){
    sum = 0;
    vecLength = VecLen;
  }
}dotStr;

#define ThreadsNo 4
pthread_t myThreads[ThreadsNo];
```

```
pthread_mutex_t mutexSum; //declares a mutex variable

void* dotProduct(void* arg){
  long offset = (long) arg;
  long len, start, end;
  double mySum = 0;
  len = dotStr.vecLength/ThreadsNo;
  start = offset*len;
  end = start + len;

  for(long i = start; i < end; i++){
    mySum += dotStr.a[i]*dotStr.b[i];
  }

  pthread_mutex_lock(&mutexSum);


  dotStr.sum += mySum;
  cout<<"Thread "<<offset<<" started from "<<setw(8)<<start<<" to "<<setw(8)<<end<<": mySum = "<<mySum<<" sum =
    "<<dotStr.sum<<endl;

  pthread_mutex_unlock(&mutexSum);


  pthread_exit(NULL);

  return NULL;
}


void main() {

  dotStr.a = new double[VecLen];
  dotStr.b = new double[VecLen];

  for(long i = 0; i < dotStr.vecLength; i++){
    dotStr.a[i] = 1;
    dotStr.b[i] = 1;
  }

  pthread_mutex_init(&mutexSum, NULL); //initially the specific mutexSum is UNLOCKED


  pthread_attr_t attr; //declares an attribute
  pthread_attr_init(&attr); //initializes a thread attribute object with the default settings for each attribute
  pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);

  clock_t t1,t2;

  t1 = clock();

  for(long i = 0; i < ThreadsNo; i++){
    pthread_create(&myThreads[i], &attr, dotProduct, (void*) i);
    /*if(i == 1)
      pthread_detach(myThreads[i]); //detach a thread even though it was created as joinable!!!
      (opposite to pthread_join())
    */
  }

  pthread_attr_destroy(&attr); //free attribute variable

  for(long i = 0; i < ThreadsNo; i++){
    pthread_join(myThreads[i],NULL); //the calling thread (Master thread which is main function) waits for other
     threads
  }
  /*  Master thread continues its instructions   */
  t2 = clock();

  cout<<setprecision(15);
  cout << "\nThe product           = " << dotStr.sum << endl;
  cout << "CPU time (in seconds) = " << (double)(t2-t1) / CLOCKS_PER_SEC << endl;

  pthread_mutex_destroy(&mutexSum); //free the mutex variable

  system("pause");

  pthread_exit(NULL); //exit Master Thread
}
```

Output

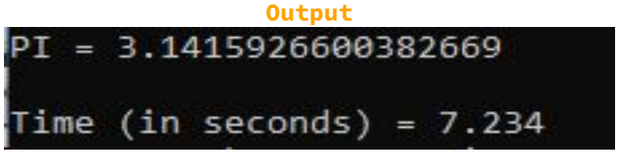
```
Thread 0 started from        0 to 20000000: mySum = 2e+007 sum = 2e+007
Thread 2 started from 40000000 to 60000000: mySum = 2e+007 sum = 4e+007
Thread 1 started from 20000000 to 40000000: mySum = 2e+007 sum = 6e+007
Thread 3 started from 60000000 to 80000000: mySum = 2e+007 sum = 8e+007

The product           = 80000000
CPU time (in seconds) = 0.164
```

PI Program

Serial (Sequential) Code

```cpp
#include<iostream>
#include<iomanip>
#include<ctime>
using namespace std;


void main() {

  clock_t t1 = clock();

  int N = 100000000000; //11 zeros
  double x = 0; //the first value of the x's
  double step = 1.0 / N; //delta x
  double sum = 0;
  for (int i = 0; i < N; i++) {
    x += step; //changing the value of the x's
    sum += 4.0 / (1 + x * x);
  }
  cout << setprecision(20);
  cout << "PI = " << sum * step << endl;

  clock_t t2 = clock();

  cout << "Time (in seconds) = " << (double)(t2 - t1) / CLOCKS_PER_SEC << endl;
  system("pause");
}
```

Output



Multi-threading (Using Pthreads) Code

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


const int ThreadsNo = 5; //the number of threads in the program
const long N = 100000000000; //11 zeros

pthread_mutex_t mutexSUM;
double sum = 0; //The (mutex) global sum

double step = 1.0/N; //delta x

void* f(void* arg){
  long id = (long) arg; //the thread id
  long len = N/(double)ThreadsNo; //the number of subintervals this thread is reponsible of
  double offset = id*len*step; //the start point of this thread (from which it will start performing its task)
  double x = offset; //the first value of x's
  double mySum = 0; //the (output) sum of this thread

  for(long i = 0; i < len; i++){
    x += step; //changing the value of the x's
    mySum += 4.0/(1+x*x);
  }

  pthread_mutex_lock(&mutexSUM);
  sum += mySum;
  cout<<"Thread "<<id<<" started from: "<<setw(3)<<offset<<" & ended at: "<<setw(3)<<offset+(len*step)<<" mySum
    = "<<mySum<<", sum = "<<sum<<endl;
  pthread_mutex_unlock(&mutexSUM);

  pthread_exit(NULL);

  return NULL;
}


void main(){

  clock_t t1 = clock();

  pthread_t myThreads[ThreadsNo]; //array of the threads in the program

  pthread_mutex_init(&mutexSUM,NULL); //initilize the mutex variable (the one that will control the access to
    the global sum "sum")

  for(long i = 0; i < ThreadsNo; i++){
    pthread_create(&myThreads[i],NULL,f,(void*)i);
  }

  for(long i = 0; i < ThreadsNo; i++){
    pthread_join(myThreads[i],NULL);
  }


  pthread_mutex_destroy(&mutexSUM); //free the mutex variable

  clock_t t2 = clock();

  cout<<setprecision(30);
  cout<<"\nPI                = "<<sum*step<<endl;
  cout<<"Time (in seconds) = "<<(double)(t2-t1)/CLOCKS_PER_SEC<<endl;

  system("pause");
  pthread_exit(NULL);
}
```

Output

```
Thread 1 started from: 0.2 & ended at: 0.4 mySum = 8.9047e+008, sum = 8.9047e+008
Thread 0 started from:   0 & ended at: 0.2 mySum = 9.59936e+008, sum = 1.85041e+009
Thread 2 started from: 0.4 & ended at: 0.6 mySum = 7.77659e+008, sum = 2.62806e+009
Thread 3 started from: 0.6 & ended at: 0.8 mySum = 6.53206e+008, sum = 3.28127e+009
Thread 4 started from: 0.8 & ended at:   1 mySum = 5.38127e+008, sum = 3.8194e+009


PI                = 3.1415926509001926
Time (in seconds) = 3.694
```