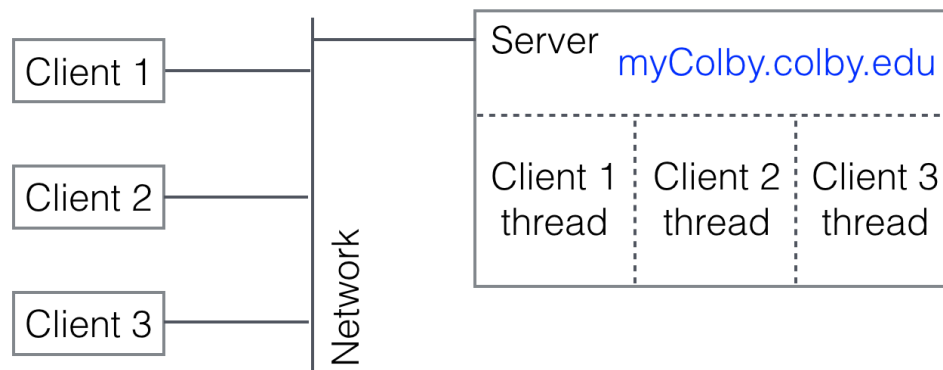


Concurrent Programming

Overview

- A **concurrent program** is a program designed to have *two or more* execution context. Such a program is said to be **multithreaded**, since more than once execution context can be *active simultaneously*.
- Although the programs used in this course so far are all single-threaded, multithread programming is widely used in daily life.
- Do you know any examples of concurrent programming?
 - One typical example is the networks. Take myColby for instance, what we use on a web browser is the front end of myColby system. We call it the client side. There is a database system on the other side of the network managing all kinds of information, such as students' information, faculty's information, courses' information, and etc. We all the database system the server side, which is the back end of the myColby system.
 - Every time a student logs into myColby, a new thread is created for the student. So that the server side must be able to handle multithreads.
 - Please note that all these threads share the same database systems. A database consists of a number of tables. A table is actually a piece of memory. It is very possible that several users want to access the same piece of memory at the same time: read the information or modify the information stored at that piece of memory (the remaining seats of a course).
 - How to make those uses can access to the same piece of memory successfully and get the correct information. These make concurrent programming challenging but interesting.



Race Condition

- **Race condition** is one of the **fundamental problems** that can occurs while **executing** different threads **asynchronously**.
- A race condition *occurs when the meaning of a program depends upon the order* in which each thread accesses a shared variable.
- Take post-increment for example. **Post-increment** is **not atomic**. It actually takes three machine instructions:
 1. fetch the value from memory

2. calculate the new value
 3. write the new value back to the memory
- Suppose there are two threads, A and B, executing the post-increment statement simultaneously, and suppose the initial value of i is 0.
 - **What are the possible results we can get?**
 - If A and B execute the three steps in the second order, the value of i is 2 after finishing executing.
 - All other possible orders lead to the value of i is 2.

$i++$

1. fetch i value, v , from memory
2. calculate new value v' based on v
3. store v' back to the memory allocated to i

thread A, B

$i = 0$

A(1)	A(1)	A(1)
B(1)	A(2)	A(2)
A(2)	A(3)	B(1)
A(3)	B(1)	B(2)
B(2)	B(2)	A(3)
B(3)	B(3)	B(3)

$i = 1$

$i = 2$

$i = 1$

Dinning Philosophers

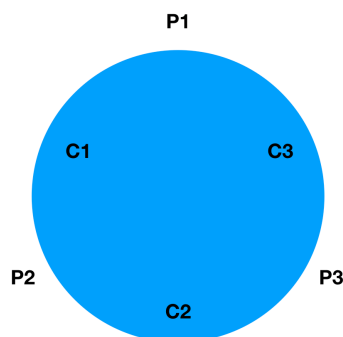
- It is **unrealistic and fanciful**. But the **synchronization behavior** that it models can **happen in real system**.
- The **description** of this problem is that:
 - A collection of **N philosophers** sits at a round table, where $N > 1$.
 - **N chopsticks** are placed on the table, one between each pair of adjacent philosophers.
 - No philosopher can eat unless he has **two chopsticks**.
 - If a philosopher doesn't eat, the he/she is back to sleep.
- Obviously, adjacent philosophers cannot eat at the same time. Each philosopher alternately eats and sleeps, waiting when necessary for the requisite chopsticks before eating.
- Our task is to **write code simulating** the dining philosophers so that no philosopher starves.
- An obvious protocol that each philosopher might follow is:

```
while (true) {
    grab left chopstick;
    grab right chopstick;
    eat;
    release left chopstick;
    release right chopstick;
    sleep;
}
```

- **What is the problem of this program? How could the problem happen?**
 - This code leads to a **deadlock**.
 - Now **assume** that action of the philosophers are **perfectly interleaved**: the first philosopher grabs his left chopstick, then the second philosopher grabs his left chopstick, and so in until the Nth philosopher grabs his left chopstick. Then the first philosopher tries to grab his right chopstick, the second philosopher tries to grab his right chopstick, and so on. they all have to wait because no right chopstick is available and they all starve.

Deadlock

- **Deadlock** occurs when a **ring of threads** comes to a point in the program where **each needs a resource from the next thread in the ring** in order to continue. No thread can provide the resource because each thread is waiting for another thread.
- **What are the possible solutions to this problem?**
 - Theoretical computer scientists have proven that there is **no deterministic uniform solution** to this problem. (By **uniform**, we mean that every philosopher **executes exactly the same code with no access to identifying state information** such as the name of the “current” philosopher.) But many non-uniform solutions exists.
 - For example, we could **number the philosophers**. **Even numbered** philosophers **ask for the left chopstick first, odd numbered ones ask for the right chopstick first**.
 - Another common solution to this sort of deadlock is to **order the resources** (in this case chopsticks) and **force the processes** (philosophers) **to grab chopsticks in ascending order**. This solution is very general and is widely used in practice.
 - Consider the case where we have 3 philosophers: P1, P2, P3. Then we order the chopsticks C1, C2, C3. (Draw circle with P1-P3 outside of the circle and C1-C3 inside the circle.)



- Now, no matter what, all the philosophers will be able to eat. For instance, if P2 gets C1, and P3 gets C2, P1 must wait until C1 is free (grabbing in ascending order). So P3 will get C3 (since there will be no contention), and finish eating. This will release C2 and C3, allowing P2 to get C2 and finish eating. Finally, this will release C1, allowing P1 to get C1 and C3 (since there will be no further contention) and finish eating.
- REF: <http://www.cs.rice.edu/~cork/book/node96.htmls>

Threads

- In a multithreaded program, **all threads execute the same piece of code**. They **share the heap**, but each thread **has its own stack frame**.
- **main() function** comprise a **single, default thread**. Threads other than the default one can be **created by programmers**.

Threads in C

- To create a multithreaded C program, you need to know these basic:

- ♦ **pthread** (POSIX thread): threads that use the POSIX standard programming interface
 - `#include <pthread.h>`
 - Define a worker function: a C routine that the thread will execute once it is created
 - `void *foo (void *args) {}`
 - Initialize `pthread_attr_t`: you can use NULL for the default values
 - `pthread_attr_t attr;`
 - `pthread_attr_init (&attr);`
 - Create a thread
 - `pthread_t thread;`
 - `pthread_create (&thread, &attr, foo, arg);`
 - Thread management
 - `pthread_join (thread, status);` //suspend execution of the calling thread until the target thread terminates
 - `pthread_exit (status);` // terminates the calling thread
 - Compiling: using `-pthread`

- First, include the header file `pthread.h`
 - Second, a worker function should be defined. A worker function is a C routine that the thread will execute once it is created.
 - third, the thread attribute should be initialized. You can use NULL for the default values. (attr: scope, joinable, size and etc.)
 - Now you can create a thread by using the `pthread_create` function.
 - After create threads, you can use `pthread_join` to suspend execution of the calling thread until the target thread terminates.
 - Or you can use `pthread_exit` to terminates the calling thread.
 - When compiling, remember to include the pthread library. You may not need this on Mac. If you use other machines, double check if the library path has been set already. (I use the `-lpthread` flag)
- Show the `helloThreads.c`, and run the code. Comment out the join for loop, run the program again, and ask students why the results are different. [`pthread_join` will suspends the calling thread (main thread) until the target thread terminates.]

```

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

#define NUM_THREADS 5

typedef struct {
    int id;
} threadInfo;

void *hello_thread(void *threadinfo) {
    threadInfo *ti = (threadInfo *) threadinfo;
    printf("Thread %d saying Hello!\n", ti->id);
    pthread_exit(NULL);
}

int main () {
    int i;
    threadInfo ti[NUM_THREADS];
    pthread_t thread[NUM_THREADS];

    // Set up the parameters for each thread
    for (i = 0; i < NUM_THREADS; i++)
        ti[i].id = i;

    // Get the threads going
    for (i = 0; i < NUM_THREADS; i++)
        pthread_create(&(thread[i]), NULL, hello_thread, &(ti[i]));

    // Join up with them. This will wait until they are done.
    for (i = 0; i < NUM_THREADS; i++)
        pthread_join(thread[i], NULL);

    return 0;
}

```

```

[rsrtaylor@n22 cs333]$ gcc -o test helloThreads.c -lpthread
[rsrtaylor@n22 cs333]$ ./test
Thread 1 saying Hello!
Thread 2 saying Hello!
Thread 0 saying Hello!
Thread 3 saying Hello!
Thread 4 saying Hello!

```

- with join for loop, the output is several “saying Hello!”
- without join for loop, the output is empty.

Synchronization

- Show students the incrementb.c, go through the code, and ask them the result? [The result may not equal to the number of threads]
 - Code REF: <http://randu.org/tutorials/threads/>
 - The code expects each thread increments the counter by one, and the final result should be the number of threads.
 - counter is a global variable, all threads share the same piece of memory.
- Why do we get incorrect results?
 - **Race condition** happens. **Multi-threads** try to **read and write to the shared memory in an unsynchronized way**.
- How do we address the races?

```

/**
 * incrementb.c
 */

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

#define NUM_THREADS 10

/* create thread argument struct for thr_func() */
typedef struct _thread_data_t {
    int tid;
} thread_data_t;

/* shared data between threads */
int shared_x;

void *thr_func(void *arg) {
    thread_data_t *data = (thread_data_t *)arg;

    printf("hello from thr_func, thread id: %d\n", data->tid);
    sleep(1);
    shared_x++;
    printf("x = %d\n", shared_x);

    pthread_exit(NULL);
}

int main(int argc, char **argv) {
    pthread_t thr[NUM_THREADS];
    int i, rc;
    /* create a thread_data_t argument array */
    thread_data_t thr_data[NUM_THREADS];

    /* initialize shared data */
    shared_x = 0;

    /* create threads */
    for (i = 0; i < NUM_THREADS; ++i) {
        thr_data[i].tid = i;
        if ((rc = pthread_create(&thr[i], NULL, thr_func, &thr_data[i]))) {
            fprintf(stderr, "error: pthread_create, rc: %d\n", rc);
            return EXIT_FAILURE;
        }
    }

    /* block until all threads complete */
    for (i = 0; i < NUM_THREADS; ++i) {
        pthread_join(thr[i], NULL);
    }

    return EXIT_SUCCESS;
}

```

```

$ ./a.out
hello from thr_func, thread id: 0
hello from thr_func, thread id: 3
hello from thr_func, thread id: 1
hello from thr_func, thread id: 2
hello from thr_func, thread id: 4
hello from thr_func, thread id: 5
hello from thr_func, thread id: 6
hello from thr_func, thread id: 7
hello from thr_func, thread id: 8
hello from thr_func, thread id: 9
x = 1
x = 2
x = 2
x = 4
x = 4
x = 5
x = 3
x = 6
x = 7
x = 7

```

- If a thread has to execute multiple atomic instructions on a shared variable, it must **lock out other thread until it is done with its critical section**.
- The **section of program where a thread read or write a shared variable** are called **critical sections**.
- What is the critical section in the sample code? [counter++, printf]
- We call the solution, **synchronization**.

Semaphore

- One way to synchronize is Semaphore.
- A semaphore (s) uses two **atomic functions**, P(s) and V(s).
 - P(s) - if $s > 0$ then assign $s = s - 1$; otherwise block the thread that called P.
 - V(s) - if a thread T is blocked on the semaphore s, wake up T; otherwise assign $s = s + 1$.
- Control of a single resource requires two semaphores, just as access to a one-lane construction area requires two separate signals.

```
/* initial state */
Semaphore empty = 1;
Semaphore full = 0;
Thing    commonBuffer;

/* Thread A */
while (true) {
    Thing value = producer();
    P(empty);
    commonBuffer = value;
    V(full);
}

/* Thread B */
while (true) {
    P(full);
    Thing value = commonBuffer;
    V(empty);
    consumer(value);
}
```

- What is the key to semaphore here?
 - The key to semaphore here is that **each process increments only one** semaphore and **decrements only one** semaphore. Therefore, each process depends upon the other process to execute the opposite operation.
- We call this type of **semaphore** a **mutex**.
 - Represents **single access to a resource**.
 - Guarantees **mutual exclusion to a critical section**.
 - Another type is called Counting Semaphore. It allows certain kinds of unsynchronized concurrent access a resource (e.g., multiple reading threads).
 - REF: <https://cseweb.ucsd.edu/classes/fa05/cse120/lectures/120-l6.pdf>

Synchronization in C

- Synchronization in C uses Mutex.
- The Mutex is provided in pthread.
- What is pthread?
 - Historically, hardware vendors have implemented their own proprietary versions of threads. These implementations differed substantially from each other making it difficult for programmers to develop portable threaded applications.

- In order to take full advantage of the capabilities provided by threads, a standardized programming interface was required.
 - For UNIX systems, this interface has been specified by the IEEE POSIX 1003.1c standard (1995).
 - Implementations adhering to this standard are referred to as POSIX threads, or Pthreads.
 - Most hardware vendors now offer Pthreads in addition to their proprietary API's.
- Pthreads are defined as a set of C language programming types and procedure calls, implemented with a pthread.h header/include file and a thread library - though this library may be part of another library, such as libc, in some implementations.
- REF: <https://computing.llnl.gov/tutorials/pthreads/#Thread>
- A mutex lock is a variable that can be “locked” by only one thread at a time. The **thread with the lock is allowed to modify/read protected data. When it is done, it releases the lock.**
 - **Initialize** a mutex lock variable:
 - pthread_mutex_t mutex;
 - pthread_mutex_init (&mutex, NULL);
 - **Lock** a mutex lock variable: *Only one thread will be allowed to do this. The rest of the threads will be forced to wait until the lock is released. Threads will be chosen non-deterministically.*
 - pthread_mutex_lock(&mutex);
 - **Unlock** a mutex lock variable: The thread that has the lock should be the one to unlock it.
 - pthread_mutex_unlock(&mutex);
 - **Cleanup** a mutex lock variable:
 - pthread_mutex_destroy(&mutex);
- Show increment.c, highlight the mutex, and run the code.

```
$ gcc increment.c -lpthread
$ ./a.out
hello from thr_func, thread id: 1
hello from thr_func, thread id: 0
hello from thr_func, thread id: 4
hello from thr_func, thread id: 3
hello from thr_func, thread id: 2
hello from thr_func, thread id: 5
hello from thr_func, thread id: 6
hello from thr_func, thread id: 7
hello from thr_func, thread id: 8
hello from thr_func, thread id: 9
x = 1
x = 2
x = 3
x = 4
x = 5
x = 6
x = 7
x = 8
x = 9
x = 10
```

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

#define NUM_THREADS 10

/* create thread argument struct for thr_func() */
typedef struct _thread_data_t {
    int tid;
} thread_data_t;

/* shared data between threads */
int shared_x;
pthread_mutex_t lock_x;

void *thr_func(void *arg) {
    thread_data_t *data = (thread_data_t *)arg;

    printf("hello from thr_func, thread id: %d\n", data->tid);
    sleep(1);
    /* get mutex before modifying and printing shared_x */
    pthread_mutex_lock(&lock_x);
    shared_x++;
    printf("x = %d\n", shared_x);
    pthread_mutex_unlock(&lock_x);

    pthread_exit(NULL);
}

int main(int argc, char **argv) {
    pthread_t thr[NUM_THREADS];
    int i, rc;
    /* create a thread_data_t argument array */
    thread_data_t thr_data[NUM_THREADS];

    /* initialize shared data */
    shared_x = 0;

    /* initialize pthread mutex protecting "shared_x" */
    pthread_mutex_init(&lock_x, NULL);

    /* create threads */
    for (i = 0; i < NUM_THREADS; ++i) {
        thr_data[i].tid = i;
        if ((rc = pthread_create(&thr[i], NULL, thr_func, &thr_data[i]))) {
            fprintf(stderr, "error: pthread_create, rc: %d\n", rc);
            return EXIT_FAILURE;
        }
    }

    /* block until all threads complete */
    for (i = 0; i < NUM_THREADS; ++i) {
        pthread_join(thr[i], NULL);
    }

    // destroy the mutex lock (we are done with it)
    pthread_mutex_destroy(&lock_x);

    return EXIT_SUCCESS;
}
```

Example

- Write a program that reads an int value, N, from the command line, and counts the number of prime numbers that are no larger than N.
 - Prime numbers: an integer greater than 1 that cannot be formed by multiplying two smaller integers.
 - Regular way: a loop from 0 to N, each iteration checks whether the loop variable is prime or not. If so, increment the counter.
 - The regular way works well when N is small. However, if N is large (e.g., 100,000 or even larger), is there any better way to implement the program?
 - We can use multithreading program, and let each thread count a part of the range between 0 and N. Then, these threads can count concurrently, and will shorten the computing time.
- We will implement a program that first uses two threads to count the primes, and then uses one thread to conduct the task. The program can also time the two methods so that we can study which way is faster in what scenario.

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

typedef struct {
    int start;
    int extent;
    int count;
} ThreadInfo;

// check whether a number is a prime
// return 1 if it is a prime, else 0
int is_prime(int n) {
    if (n < 2) return 0;
    if (n == 2) return 1;
    if (n%2 == 0) return 0;
    for (int i = 3; i < n; i += 2) {
        if (n%i == 0) return 0;
    }
    return 1;
}

// worker func: check the primes in
// the given range
void *thread_count_primes(void *thread_info) {
    ThreadInfo *ti = (ThreadInfo *)thread_info;
    for (int i = ti->start; i < ti->start + ti->extent; i++) {
        if (is_prime(i))
            ti->count++;
    }

    pthread_exit(NULL);
}
```

```
int main (int argc, char** argv) {
    int N = 100000;
    if (argc > 1) N = atoi(argv[1]);

    double t1, t2;
    ThreadInfo ti[3];
    pthread_t thread[3]; // use 2 of the threads to count N numbers

    ti[0].start = 0;
    ti[0].extent = N/2;
    ti[0].count = 0;
    ti[1].start = N/2;
    ti[1].extent = N/2;
    ti[1].count = 0;

    ti[2].start = 0;
    ti[2].extent = N;
    ti[2].count = 0;

    t1 = get_time_sec();

    // create threads
    for (int i = 0; i < 2; i++) {
        pthread_create(&thread[i], NULL, thread_count_primes, &ti[i]);
    }

    // join threads
    for (int i = 0; i < 2; i++) {
        pthread_join(thread[i], NULL);
    }

    // sum counts
    int count = 0;
    for (int i = 0; i < 2; i++) {
        count += ti[i].count;
    }

    t2 = get_time_sec();
    printf("There are %d primes not larger than 100000\n", count);
    printf("It took %f seconds to count the number with 2 threads\n", t2 - t1);

    t1 = get_time_sec();

    // create thread
    pthread_create(&thread[2], NULL, thread_count_primes, &ti[2]);

    // join thread
    pthread_join(thread[2], NULL);

    // sum count
    count = ti[2].count;

    t2 = get_time_sec();
    printf("There are %d primes not larger than 100000\n", count);
    printf("It took %f seconds to count the number with 1 thread\n", t2 - t1);

    return EXIT_SUCCESS;
}
```

Threads in Java

- There are two ways to create threads in Java: extends the Thread class or implement the Runnable interface.
- In both ways, we need to implement run method, which is equivalent to the worker function in C.
- Then, we need to create a thread object and call the start method.
- Show HelloRunnable.java and HelloThread.java

```
/**
 * HelloRunnable.java
 */
public class HelloRunnable implements Runnable {
    public void run () {
        System.out.println("Hello Runnable!");
    }

    public static void main (String[] args) {
        Thread t = new Thread(new HelloRunnable());
        t.start();
    }
}
```

```
/**
 * HelloThread.java
 */
public class HelloThread extends Thread {
    public void run () {
        System.out.println("Hello Thread!");
    }

    public static void main (String[] args) {
        Thread t = new HelloThread();
        t.start();
    }
}
```

- The difference in creating a thread object is due to that the Thread class implements runnable interface.
- Show MyJoin.java
 - Run the code without the join part (for (JoinThread s : list)), and ask why we get the output.
 - Output: []
 - Reason: thread for main method ends before other threads finish.
 - Solution: using join()
 - t.join() causes the current thread to pause execution until t's thread terminates.
 - Run the code with join part, but without synchronization part (MyJoin.addName(getName())), ask the reason.
 - Output: [null, null, null, null, Thread-0]
 - Reason: race conditions
 - Solution: synchronization
 - Run the code with join part and the synchronization part.
 - Output: [Thread-4, Thread-0, Thread-3, Thread-1, Thread-2]
 - synchronized: synchronized methods provide a simply strategy to avoid race conditions.

```
/**
 * MyJoin.java
 */
import java.util.ArrayList;

class JoinThread extends Thread {
    public void run () {

        for (int i = 0; i < 10; i++) {
            try {
                Thread.sleep(10);
            } catch (Exception e) {
                System.err.println(e);
            }
        }

        //MyJoin.names.add(getName());
        MyJoin.addName(getName()); //synchronization part
    }
}

public class MyJoin {
    public static ArrayList<String> names = new ArrayList<String>();

    public static synchronized void addName (String s) {
        names.add(s);
    }

    public static void main (String args[]) {
        ArrayList<JoinThread> list = new ArrayList<JoinThread>();

        for (int i = 0; i < 5; i++) {
            JoinThread s = new JoinThread();
            list.add(s);
            s.start();
        }

        // join part
        for (JoinThread s : list) {
            try {
                s.join();
            } catch (Exception e) {
                System.err.println(e);
            }
        }

        System.out.println(names);
    }
}
```

```

public class PrimeCounter implements Runnable {
    private int start;
    private int extent;
    private int count;

    public PrimeCounter (int s, int e) {
        start = s;
        extent = e;
        count = 0;
    }

    public boolean is_prime (int n) {
        if (n < 2) return false;
        if (n == 2) return true;
        if (n%2 == 0) return false;
        for (int i = 3; i < n; i += 2) {
            if (n % i == 0) return false;
        }
        return true;
    }

    public void run () {
        for (int i = start; i < start + extent; i++) {
            if (is_prime(i)) {
                count += 1;
            }
        }
    }

    public int count () {
        return count;
    }

    public static void main (String args[]) {
        int N = 100000;

        PrimeCounter pc1 = new PrimeCounter(0, N/2);
        PrimeCounter pc2 = new PrimeCounter(N/2, N/2);
        PrimeCounter pc3 = new PrimeCounter(0, N);

        Thread t1 = new Thread(pc1);
        Thread t2 = new Thread(pc2);
        Thread t3 = new Thread(pc3);

        long start = System.currentTimeMillis();

        t1.start();
        t2.start();
        try {
            t1.join();
            t2.join();
        } catch (Exception e) {
            System.err.println(e);
        }

        int count = pc1.count() + pc2.count();

        long end = System.currentTimeMillis();
        System.out.printf("Two threads count %d primes in %d ms\n", count, end-
start);
    }
}

```

```

        start = System.currentTimeMillis();
        t3.start();
        try {
            t3.join();
        } catch (Exception e) {
            System.err.println(e);
        }

        count = pc3.count();

        end = System.currentTimeMillis();
        System.out.printf("One threads count %d primes in %d ms\n", count, end-
start);
    }
}

```

Threads in Python

- Python has a package called “threading” that allows multiple threads to be executed, but it does not allow for multiple cores to be run simultaneously. This means it is helpful only when it makes logical sense to create a separate thread for a separate task - it doesn’t help with speed!
- There is another package called multiprocessing, and it will take advantage of multiple cores, but it does so by creating new processes, which are “heavier weight” than threads because the operating system manages each process separately. Shared memory is more tricky with multiple processes.
- Below is code for counting primes in Python using multithreading. It isn’t any faster than doing it sequentially.

```
# PrimeCounter.py

import threading
import time

class PrimeCounter (threading.Thread):

    def __init__ (self, s, e):
        threading.Thread.__init__(self)
        self.begin = s
        self.extent = e
        self.count = 0

    def is_prime (self, n):
        if (n < 2):
            return False
        if (n == 2):
            return True
        if (n % 2 == 0):
            return False
        for i in range (3, n, 2):
            if (n % i == 0):
                return False;
        return True

    def run (self):
        for i in range (self.begin, self.begin + self.extent):
            if self.is_prime(i):
                self.count += 1

    def getCount (self):
        return self.count

def main ():
    N = 100000

    pc1 = PrimeCounter(0, N//2)
    pc2 = PrimeCounter(N//2, N//2)
    pc3 = PrimeCounter(0, N)

    t1 = time.clock()

    pc1.start()
    pc2.start()

    pc1.join()
    pc2.join()

    t2 = time.clock()

    print("2 thread count ", pc1.getCount() + pc2.getCount(), " primes")
    print("time is ", t2 - t1)

    t1 = time.clock()

    pc3.start()
    pc3.join()

    t2 = time.clock()

    print("1 thread count ", pc3.getCount(), " primes")
    print("time is ", t2 - t1)

if __name__ == "__main__":
    main()
```

