

Concurrent Programming (III)

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]

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

#define NUM_THREADS 100
#define NUM_ITERATIONS 1000

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

/* shared data between threads */
int shared_x;

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

    printf("hello from thr_func, thread id: %d\n", data->tid);
    for (int i = 0; i < NUM_ITERATIONS; i++)
        shared_x++;
    printf("thread id: %d: shared_x = %d\n", data->tid, 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_info 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);
    }

    printf("Expected shared_x: %d, actual shared_x: %d\n", NUM_THREADS * NUM_ITERATIONS,
        shared_x);

    return EXIT_SUCCESS;
}
```

```
yingli@mbp-230460 code % ./a.out
hello from thr_func, thread id: 0
hello from thr_func, thread id: 4
hello from thr_func, thread id: 5
thread id: 4: shared_x = 2000
thread id: 5: shared_x = 3000
hello from thr_func, thread id: 3
thread id: 3: shared_x = 4000
...
hello from thr_func, thread id: 99
thread id: 99: shared_x = 95087
hello from thr_func, thread id: 93
thread id: 93: shared_x = 96087
hello from thr_func, thread id: 98
thread id: 98: shared_x = 97087
hello from thr_func, thread id: 94
thread id: 94: shared_x = 98087
Expected shared_x: 100000, actual
shared_x: 98087
```

- 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 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?
- 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**.
-

Synchronization in C

- Synchronization in C uses Mutex.
- The Mutex is provided in pthread.
- 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.

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

#define NUM_THREADS 100
#define NUM_ITERATIONS 1000

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

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

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

    printf("hello from thr_func, thread id: %d\n", data->tid);

    /* get mutex before modifying and printing shared_x */
    pthread_mutex_lock(&lock_x);
    for (int i = 0; i < NUM_ITERATIONS; i++)
        shared_x++;
    printf("thread id: %d: shared_x = %d\n", data->tid, 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_info 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);

    printf("Expected shared_x: %d, actual shared_x: %d\n", NUM_THREADS *
        NUM_ITERATIONS, shared_x);

    return EXIT_SUCCESS;
}
```

```

$ gcc increment.c -lpthread
yingli@mbp-230460 code % ./a.out
hello from thr_func, thread id: 0
thread id: 0: shared_x = 1000
hello from thr_func, thread id: 1
thread id: 1: shared_x = 2000
hello from thr_func, thread id: 4
thread id: 4: shared_x = 3000
hello from thr_func, thread id: 2
thread id: 2: shared_x = 4000
...
thread id: 29: shared_x = 97000
thread id: 97: shared_x = 98000
thread id: 98: shared_x = 99000
thread id: 99: shared_x = 100000
Expected shared_x: 100000, actual shared_x: 100000

```

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;
}

```