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.

```c
/**
 * incrementb.c
 */

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

#define NUM_THREADS 10

typedef struct _thread_data_t {
    int tid;
} thread_data_t;

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 = 4
x = 4
x = 5
x = 3
x = 6
x = 7
x = 7
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.

```
$ 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
```
```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;
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.

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

    double t1, t2;
    ThreadInfo ti[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

```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);
    }
}