4-7

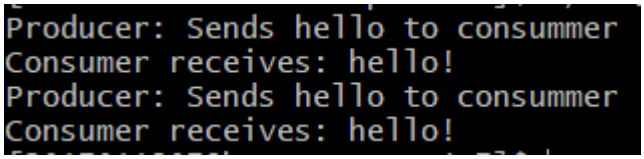
- 双线程程序 `4-7/two_threads.cpp`

```
void* produce(void* rank) {
    pthread_mutex_lock(&mtx);
    message = "hello!";
    available = 1;
    pthread_mutex_unlock(&mtx);
    printf("Producer: Sends hello to consummer\n");
    return NULL;
}

void* consume(void* rank) {
    while (1) {
        pthread_mutex_lock(&mtx);
        if (available) {
            printf("Consumer receives: %s\n", message.c_str());
            pthread_mutex_unlock(&mtx);
            break;
        }
        pthread_mutex_unlock(&mtx);
    }
    return NULL;
}

int main() {
    pthread_t producer, consumer;
    pthread_mutex_init(&mtx, NULL);
    message = "";
    available = 0;
    pthread_create(&producer, NULL, produce, NULL);
    pthread_create(&consumer, NULL, consume, NULL);
    pthread_join(producer, NULL);
    pthread_join(consumer, NULL);
    message = "";
    available = 0;
    pthread_create(&consumer, NULL, consume, NULL);
    pthread_create(&producer, NULL, produce, NULL);
    pthread_join(consumer, NULL);
    pthread_join(producer, NULL);
    return 0;
}
```

两个线程分别调用produce和consume，经测试，无论先启动生产者线程还是先启动消费者线程，程序都可正常运行，程序输出如下



- 2k个线程 `4-7/two_thousands.cpp`

```
void* produce(void* rank_) {
    long rank = (long) rank_;
    while (1) {
        pthread_mutex_lock(&mtx);
        if (!available) {
            sprintf(message, "Hello from %lld", rank);
            available = 1;
            pthread_mutex_unlock(&mtx);
            break;
        }
        pthread_mutex_unlock(&mtx);
    }
    printf("[%lld]: Sends hello\n", rank);
    return NULL;
}

void* consume(void* rank_) {
    long rank = (long) rank_;
    char local_msg[110];
    while (1) {
        pthread_mutex_lock(&mtx);
        if (available) {
            strcpy(local_msg, message);
            available = 0;
            pthread_mutex_unlock(&mtx);
            break;
        }
        pthread_mutex_unlock(&mtx);
    }
    printf("[%lld]: %s\n", rank, local_msg);
    return NULL;
}
```

与上一问相比，主要有以下更改：

1. produce也可能不能一次性成功，故需要忙等待
2. IO操作是程序瓶颈，应从critical section中移出来
3. 取出消息后available要置零

程序输出如下

- 一般化程序

```
void* send_recv(void* rank_) {
    long rank = (long) rank_;
    long dest = (rank+1) % max_thread;
    bool sendSucc = 0, recvSucc = 0;
    char local_msg[110];

    while (1) {
        if (!sendSucc && empty) {
            pthread_mutex_lock(&mtx);
            if (empty) {
                sprintf(message, "Hello to %lld from %lld", dest, rank);
                empty = 0;
                available[dest] = 1;
                sendSucc = 1;
            }
```

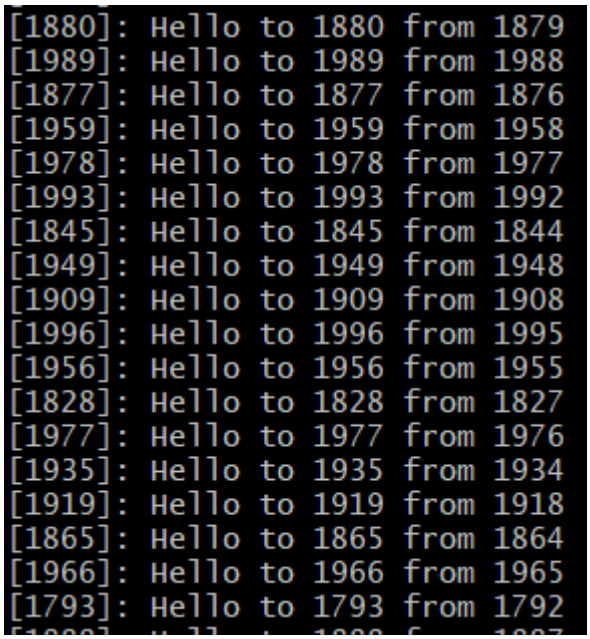
```
            pthread_mutex_unlock(&mtx);
        }
        if (!recvSucc && available[rank]) {
            pthread_mutex_lock(&mtx);
            strcpy(local_msg, message);
            recvSucc = 1;
            empty = 1;
            pthread_mutex_unlock(&mtx);
            printf("[%lld]: %s\n", rank, local_msg);
        }
        if (sendSucc && recvSucc)
            break;
        pthread_yield();
    }
    return NULL;
}

```

与上一问相比，主要有以下更改

1. 如果所有线程都先发消息再收消息，会出现死锁，因此每个线程都在交替尝试发消息和收消息。
2. 这次发送消息是有目标的，因此需要在发消息的时候指出消息是发给谁的
3. 消息有目标带来的另一问题是，接受消息的成功率很低，因此选择在critical section外确认消息是否是发给自己的。
4. 在尝试一次后，应使用`pthread_yield()`让出执行权，以让其他线程尝试接受。这样程序效率会有数倍提升。

部分运行结果



此题需要忙等待机制。

4-11

- 两个Delete操作同时执行

现在两个线程的`curr_p`都指向存储5的节点。这可能出现以下两个问题：

1. 两个线程都在执行Delete(5)，则都会返回删除成功
2. 线程0在执行Delete(8)，线程1在执行Delete(5)，则线程1可能在线程0查找到存储8的节点前释放存储5的节点。如果内存在线程0到达之前进行了重新分配用于其他用途，而不是作为链表节点，那么线程0在执行`curr_p = curr_p->next`后，对`curr_p`的引用可能会导致段违规。

- 一个Insert和一个Delete操作同时执行

  现在线程0的`pred_p`指向存储5的节点，线程1的`curr_p`指向存储5的节点。这可能出现以下两个问题:

  1. 线程0在执行Insert(5)，线程1在执行Delete(5)。线程0会报告插入失败，但5可能在线程0返回前就被删除了，因此插入操作应当是成功的。同理，如果线程0在执行Insert(6)，线程1在执行Delete(6)，则线程1会报告删除失败，但在线程1返回前，6的插入操作可能已经成功，因此删除操作应当是成功的。
  2. 线程0在执行Insert(8)，线程1在执行Delete(5)，则线程1可能在线程0查找到存储8的节点前释放存储5的节点。如果内存在线程0到达之前进行了重新分配用于其他用途，而不是作为链表节点，那么线程0在执行`curr_p = curr_p->next`后，对`curr_p`的引用可能会导致段违规。

- 一个Member和一个Delete操作同时执行

  这即是课本第123页描述的情况，现在两个线程的`curr_p`都指向存储5的节点，这可能出现以下两个问题：

  1. 线程0在执行Member(5)，线程1在执行Delete(5)，则线程0会报告5在链表中，但5可能在线程0返回前就被删除了。
  2. 线程0在执行Member(8)，线程1在执行Delete(5)，则线程1可能在线程0查找到存储8的节点前释放存储5的节点。如果内存在线程0到达之前进行了重新分配用于其他用途，而不是作为链表节点，那么线程0在执行`curr_p = curr_p->next`后，对`curr_p`的引用可能会导致段违规。

- 两个Insert同时执行

  现在两个线程的`pred_p`都指向存储5的节点，这可能出现以下两个问题:

  1. 两个线程都在执行Insert(6)，则它们都会报告插入成功
  2. 线程0在执行Insert(6)，线程1在执行Insert(7)，它们现在都执行到了程序4-10的第10行。之后，线程0先执行了11-19行，线程1后执行11-19行。则线程1会将存储5的节点的next设为存储7的节点，而覆盖掉线程0对6的插入。

- 一个Insert和一个Member同时执行

  现在线程0的`curr_p`指向存储5的节点，线程1的`pred_p`指向存储5的节点。

  1. 线程0在执行Member(6)，线程1在执行Insert(6)，则线程0会报告6不在链表中，但6可能在线程0返回前被成功插入。

4-12

只使用读锁或只使用写锁都是不安全的。

同时使用读锁和写锁也是不安全的，因为在查找和修改之间，其他线程可能获得写锁并对链表进行修改。

5-4

&&：1

||：0

&： -1

|: 0

^: 0

5-5

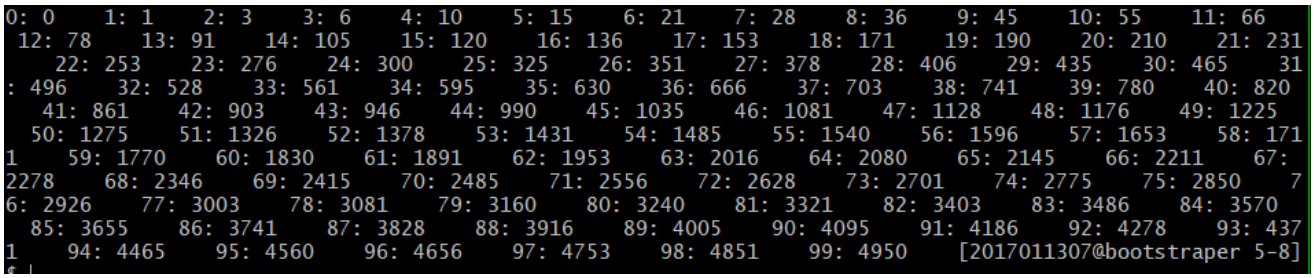
a) 1010.0

b)线程1算出4，线程2算出1004，四舍五入得1000；两个线程合并后得到1000.0

5-8

使用通项公式计算即可，代码见`5-8/main.cpp`

```
#   pragma omp parallel for num_threads(thread_count)
    for (int i=0; i<n; i++)
        a[i] = i * (i+1) / 2;
```



5-14

a) y[0]是所在缓存行中的第一个元素，则需要1个缓存行

b) y[0]不是所在缓存行中的第一个元素，则需要2个缓存行

c) 8种，第i种方法是把y[0]存在所在缓存行的第i个位置

d) 3种，`[12]-[34]`、`[13]-[24]`、`[14]-[23]`

e) 处理第1234行的两个线程属于处理器1，处理5678行的两个线程属于处理器2；y[0]是所在缓存行的第5个元素。这样，y[0]-y[3]位于一个缓存行，都由处理器1中的线程处理，y[4]-y[7]位于同一个缓存行，都由处理器2中的线程处理。

f) 8种元素分配给缓存行的方式，3种线程分配给处理器的方式，共$3 * 8 = 24$种

g) e)中的分配方案是唯一一种不会引起伪共享的分配方法。

Histogram

代码见`histogram/main.cpp`

并行化原题中的`Set_bin`和`Find_bin`

`Set_bin`

```
#   pragma omp parallel for num_threads(thread_num)
    for (int i = 0; i < bin_count; i++) {
        bin_counts[i] = 0;
        bin_maxes[i] = min_meas + (i+1)*bin_width;
    }
```

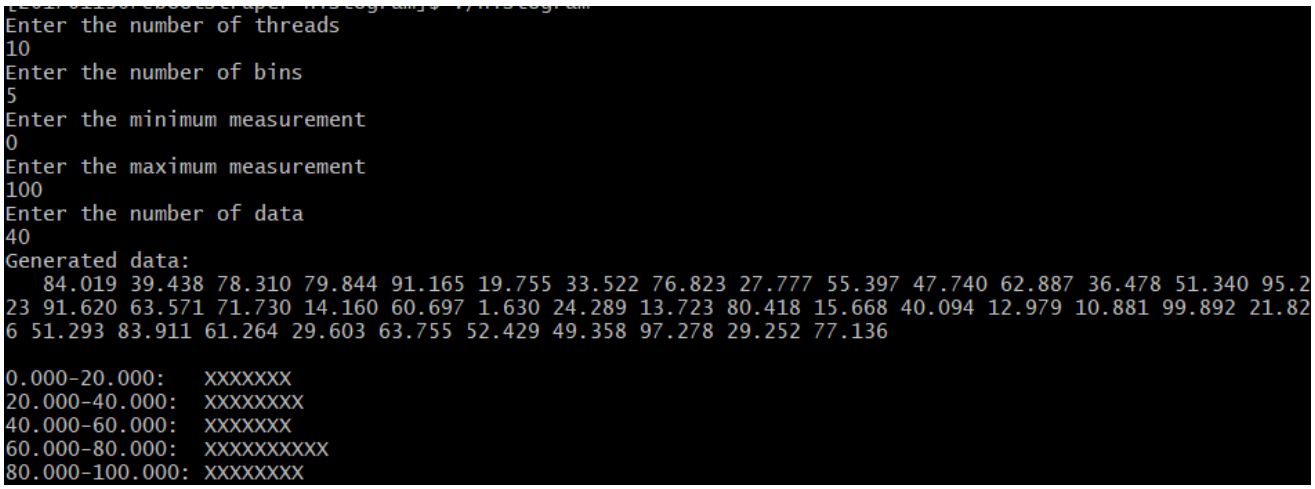
`Find_bin`

```
    omp_lock_t *lock_bin_counts = new omp_lock_t[bin_count];
#   pragma omp parallel num_threads(thread_num)\
        default(none) shared(lock_bin_counts, data, bin_maxes, bin_count,
data_count, min_meas, bin_counts)
    {
#       pragma omp for
        for (int i=0; i<bin_count; i++)
            omp_init_lock(&lock_bin_counts[i]);
#       pragma omp for
        for (int i = 0; i<data_count; i++) {
            int bin = Which_bin(data[i], bin_maxes, bin_count, min_meas);
            omp_set_lock(&lock_bin_counts[bin]);
            bin_counts[bin]++;
            omp_unset_lock(&lock_bin_counts[bin]);
        }
#       pragma omp for
        for (int i=0; i<bin_count; i++)
            omp_destroy_lock(&lock_bin_counts[i]);
    }
    delete[] lock_bin_counts;
```

要点如下

1. 使用锁来保证`bin_counts[bin]`自增操作的原子性
2. `find_bin`中需并行的地方较多，故最好预先开好线程

运行结果如下



Pthread Programming

```cpp
#include <cstdio>
#include <cstdlib>
#include <cstring>
#include <string>
#include <pthread.h>
#include <assert.h>

const int max_num_thread = 4096;
int tot_thread = 2; // the main thread and the thread forked by main
pthread_mutex_t mtx;


long fib_local(int x) {
    long a = 0, b = 1, c;
    for (int i=2; i<=x; i++) {
        c = b; b = a + b; a = c;
    }
    return b;
}

void* fib_thread(void* x_) {
    long x = (long)x_;
    if (x == 0)
        return (void*)(0);
    if (x == 1)
        return (void*)(1);
    bool new_thread = 0;
    pthread_mutex_lock(&mtx);
    if (tot_thread+2 <= max_num_thread) {
        new_thread = 1;
        tot_thread += 2;
    }
    pthread_mutex_unlock(&mtx);
    long ans;
    if (new_thread) {
        pthread_t thd1, thd2;
        void *ans1, *ans2;
        pthread_create(&thd1, NULL, fib_thread, (void*)(x-1));
        pthread_create(&thd2, NULL, fib_thread, (void*)(x-2));
        pthread_join(thd1, &ans1);
        pthread_join(thd2, &ans2);
        pthread_mutex_lock(&mtx);
        tot_thread -= 2;
        pthread_mutex_unlock(&mtx);
        return (void*)(long(ans1)+long(ans2));
    } else {
        return (void*)(fib_local(x));
    }

}

int main() {
    long n;
```

```
54      scanf("%lld", &n);
55      pthread_t thd;
56      pthread_mutex_init(&mtx, NULL);
57      pthread_create(&thd, NULL, fib_thread, (void*)(n));
58      void* ans;
59      pthread_join(thd, &ans);
60      printf("%lld\n", long(ans));
61      assert(tot_thread == 2);
62      return 0;
63  }
```

代码见`pthread/fib.cpp`，**需要输入n的值**

使用计数器`tot_thread`记录已有的线程个数，并在该数的锁的内部判断是否应该新建线程。在计算结束后，也应重置`tot_thread`的值。

在允许并行嵌套后，可以使用openmp进行创建两个线程的操作，所以可以在pthread版本上略作修改变成openmp版。但这样写出的程序并不能展现粗openmp较为便利的优势。

程序运行结果