

## 4-7

- 多线程程序 4-7/two\_threads.cpp

```
1 void* produce(void* rank) {
2     pthread_mutex_lock(&mtx);
3     message = "hello!";
4     available = 1;
5     pthread_mutex_unlock(&mtx);
6     printf("Producer: Sends hello to consumer\n");
7     return NULL;
8 }
9
10 void* consume(void* rank) {
11     while (1) {
12         pthread_mutex_lock(&mtx);
13         if (available) {
14             printf("Consumer receives: %s\n", message.c_str());
15             pthread_mutex_unlock(&mtx);
16             break;
17         }
18         pthread_mutex_unlock(&mtx);
19     }
20     return NULL;
21 }
22
23 int main() {
24     pthread_t producer, consumer;
25     pthread_mutex_init(&mtx, NULL);
26     message = "";
27     available = 0;
28     pthread_create(&producer, NULL, produce, NULL);
29     pthread_create(&consumer, NULL, consume, NULL);
30     pthread_join(producer, NULL);
31     pthread_join(consumer, NULL);
32     message = "";
33     available = 0;
34     pthread_create(&consumer, NULL, consume, NULL);
35     pthread_create(&producer, NULL, produce, NULL);
36     pthread_join(consumer, NULL);
37     pthread_join(producer, NULL);
38     return 0;
39 }
```

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

```
Producer: Sends hello to consumer
Consumer receives: hello!
Producer: Sends hello to consumer
Consumer receives: hello!
```

- 2k个线程 4-7/two\_thousands.cpp

```
1 void* produce(void* rank_) {
2     long rank = (long) rank_;
3     while (1) {
4         pthread_mutex_lock(&mtx);
5         if (!available) {
6             sprintf(message, "Hello from %ld", rank);
7             available = 1;
8             pthread_mutex_unlock(&mtx);
9             break;
10        }
11        pthread_mutex_unlock(&mtx);
12    }
13    printf("[%ld]: Sends hello\n", rank);
14    return NULL;
15 }
16
17 void* consume(void* rank_) {
18     long rank = (long) rank_;
19     char local_msg[110];
20     while (1) {
21         pthread_mutex_lock(&mtx);
22         if (available) {
23             strcpy(local_msg, message);
24             available = 0;
25             pthread_mutex_unlock(&mtx);
26             break;
27         }
28         pthread_mutex_unlock(&mtx);
29     }
30    printf("[%ld]: %s\n", rank, local_msg);
31    return NULL;
32 }
```

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

```
[1946]: Sends hello
[1947]: Hello from 1946
[1948]: Sends hello
[1949]: Hello from 1948
[1950]: Sends hello
[1951]: Hello from 1950
[1952]: Sends hello
[1954]: Sends hello
[1953]: Hello from 1954
[1956]: Sends hello
[1955]: Hello from 1956
[1957]: Hello from 1956
[1958]: Sends hello
[1959]: Hello from 1958
[1960]: Sends hello
[1961]: Hello from 1960
[1962]: Sends hello
[1963]: Hello from 1962
[1964]: Sends hello
[1965]: Hello from 1964
[1966]: Sends hello
[1967]: Hello from 1966
[1968]: Sends hello
[1969]: Hello from 1968
[1970]: Sends hello
[1971]: Hello from 1970
[1972]: Sends hello
[1973]: Hello from 1972
[1974]: Sends hello
[1975]: Hello from 1974
[1976]: Sends hello
[1977]: Hello from 1976
[1978]: Sends hello
[1979]: Hello from 1978
[1980]: Sends hello
[1981]: Hello from 1980
[1982]: Sends hello
```

- 一般化程序

```
1 void* send_recv(void* rank_) {
2     long rank = (long) rank_;
3     long dest = (rank+1) % max_thread;
4     bool sendSucc = 0, recvSucc = 0;
5     char local_msg[110];
6
7     while (1) {
8         if (!sendSucc && empty) {
9             pthread_mutex_lock(&mtx);
10            if (empty) {
11                sprintf(message, "Hello to %lld from %lld", dest, rank);
12                empty = 0;
13                available[dest] = 1;
14                sendSucc = 1;
15            }
16        }
17    }
```

```

16         pthread_mutex_unlock(&mtx);
17     }
18     if (!recvSucc && available[rank]) {
19         pthread_mutex_lock(&mtx);
20         strcpy(local_msg, message);
21         recvSucc = 1;
22         empty = 1;
23         pthread_mutex_unlock(&mtx);
24         printf("[%11d]: %s\n", rank, local_msg);
25     }
26     if (sendSucc && recvSucc)
27         break;
28     pthread_yield();
29 }
30 return NULL;
31 }
32

```

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

```

[1880]: Hello to 1880 from 1879
[1989]: Hello to 1989 from 1988
[1877]: Hello to 1877 from 1876
[1959]: Hello to 1959 from 1958
[1978]: Hello to 1978 from 1977
[1993]: Hello to 1993 from 1992
[1845]: Hello to 1845 from 1844
[1949]: Hello to 1949 from 1948
[1909]: Hello to 1909 from 1908
[1996]: Hello to 1996 from 1995
[1956]: Hello to 1956 from 1955
[1828]: Hello to 1828 from 1827
[1977]: Hello to 1977 from 1976
[1935]: Hello to 1935 from 1934
[1919]: Hello to 1919 from 1918
[1865]: Hello to 1865 from 1864
[1966]: Hello to 1966 from 1965
[1793]: Hello to 1793 from 1792

```

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`

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

```
0: 0    1: 1    2: 3    3: 6    4: 10   5: 15   6: 21   7: 28   8: 36   9: 45   10: 55  11: 66
12: 78  13: 91  14: 105 15: 120 16: 136 17: 153 18: 171 19: 190 20: 210 21: 231
22: 253 23: 276 24: 300 25: 325 26: 351 27: 378 28: 406 29: 435 30: 465 31:
: 496 32: 528 33: 561 34: 595 35: 630 36: 666 37: 703 38: 741 39: 780 40: 820
41: 861 42: 903 43: 946 44: 990 45: 1035 46: 1081 47: 1128 48: 1176 49: 1225
50: 1275 51: 1326 52: 1378 53: 1431 54: 1485 55: 1540 56: 1596 57: 1653 58: 171
1 59: 1770 60: 1830 61: 1891 62: 1953 63: 2016 64: 2080 65: 2145 66: 2211 67:
2278 68: 2346 69: 2415 70: 2485 71: 2556 72: 2628 73: 2701 74: 2775 75: 2850 7
6: 2926 77: 3003 78: 3081 79: 3160 80: 3240 81: 3321 82: 3403 83: 3486 84: 3570
85: 3655 86: 3741 87: 3828 88: 3916 89: 4005 90: 4095 91: 4186 92: 4278 93: 437
1 94: 4465 95: 4560 96: 4656 97: 4753 98: 4851 99: 4950 [2017011307@bootstraper 5-8]
```

## 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`

```

1 # pragma omp parallel for num_threads(thread_num)
2   for (int i = 0; i < bin_count; i++) {
3       bin_counts[i] = 0;
4       bin_maxes[i] = min_meas + (i+1)*bin_width;
5   }

```

#### Find\_bin

```

1     omp_lock_t *lock_bin_counts = new omp_lock_t[bin_count];
2 # pragma omp parallel num_threads(thread_num)\
3     default(none) shared(lock_bin_counts, data, bin_maxes, bin_count,
4 data_count, min_meas, bin_counts)
4     {
5 #         pragma omp for
6         for (int i=0; i<bin_count; i++)
7             omp_init_lock(&lock_bin_counts[i]);
8 #         pragma omp for
9         for (int i = 0; i<data_count; i++) {
10            int bin = which_bin(data[i], bin_maxes, bin_count, min_meas);
11            omp_set_lock(&lock_bin_counts[bin]);
12            bin_counts[bin]++;
13            omp_unset_lock(&lock_bin_counts[bin]);
14        }
15 #         pragma omp for
16         for (int i=0; i<bin_count; i++)
17             omp_destroy_lock(&lock_bin_counts[i]);
18     }
19     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

```

Enter the number of threads
10
Enter the number of bins
5
Enter the minimum measurement
0
Enter the maximum measurement
100
Enter the number of data
40
Generated data:
84.019 39.438 78.310 79.844 91.165 19.755 33.522 76.823 27.777 55.397 47.740 62.887 36.478 51.340 95.2
23 91.620 63.571 71.730 14.160 60.697 1.630 24.289 13.723 80.418 15.668 40.094 12.979 10.881 99.892 21.82
6 51.293 83.911 61.264 29.603 63.755 52.429 49.358 97.278 29.252 77.136
0.000-20.000:  XXXXXXXX
20.000-40.000:  XXXXXXXXX
40.000-60.000:  XXXXXXXX
60.000-80.000:  XXXXXXXXXXXX
80.000-100.000: XXXXXXXXX

```

## Pthread Programming

```

1  #include <stdio>
2  #include <stdlib>
3  #include <string>
4  #include <pthread.h>
5  #include <assert.h>
6
7
8  const int max_num_thread = 4096;
9  int tot_thread = 2; // the main thread and the thread forked by main
10 pthread_mutex_t mtx;
11
12
13 long fib_local(int x) {
14     long a = 0, b = 1, c;
15     for (int i=2; i<=x; i++) {
16         c = b; b = a + b; a = c;
17     }
18     return b;
19 }
20
21 void* fib_thread(void* x_) {
22     long x = (long)x_;
23     if (x == 0)
24         return (void*)(0);
25     if (x == 1)
26         return (void*)(1);
27     bool new_thread = 0;
28     pthread_mutex_lock(&mtx);
29     if (tot_thread+2 <= max_num_thread) {
30         new_thread = 1;
31         tot_thread += 2;
32     }
33     pthread_mutex_unlock(&mtx);
34     long ans;
35     if (new_thread) {
36         pthread_t thd1, thd2;
37         void *ans1, *ans2;
38         pthread_create(&thd1, NULL, fib_thread, (void*)(x-1));
39         pthread_create(&thd2, NULL, fib_thread, (void*)(x-2));
40         pthread_join(thd1, &ans1);
41         pthread_join(thd2, &ans2);
42         pthread_mutex_lock(&mtx);
43         tot_thread -= 2;
44         pthread_mutex_unlock(&mtx);
45         return (void*)(long(ans1)+long(ans2));
46     } else {
47         return (void*)(fib_local(x));
48     }
49
50 }
51
52 int main() {
53     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

```
[2017011307@bootstraper pthread]$ ./fib
10
55
```