THREADS AND LOCKING作业报告

软工2班 3017218092 侯雨茜

一、准备工作

下载ph.c并将其编译：

```
$ gcc -g -O2 ph.c -pthread
$ ./a.out 2
```

2指定在哈希表上执行放置和获取操作的线程数。

编译产生如下输出如下：

```
(base) houyuqiandeMacBook-Pro:threads yuqianhou$ gcc -g -O2 ph.c -pthread
(base) houyuqiandeMacBook-Pro:threads yuqianhou$ ./a.out 2
1: put time = 0.004478
0: put time = 0.004490
1: get time = 4.755899
1: 16808 keys missing
0: get time = 4.757971
0: 16808 keys missing
completion time = 4.762747
```

每个线程分两个阶段运行。在第一阶段，每个线程将NKEYS / nthread键放入哈希表。在第二阶段，每个线程从哈希表获取NKEYS。从输出中得知每个线程每个阶段花费了多长时间以及应用程序的总运行时间。在上面的输出中，应用程序的完成时间约为4.8秒。每个线程的计算时间约为4.8秒（put约为0.0，get约为4.8）。

要查看是否使用两个线程可以提高性能，与一个线程进行比较：

```
$ ./a.out 1
```

编译产生如下输出如下：

```
(base) houyuqiandeMacBook-Pro:threads yuqianhou$ ./a.out 1
0: put time = 0.006168
0: get time = 4.457747
0: 0 keys missing
completion time = 4.464060
(base) houyuqiandeMacBook-Pro:threads yuqianhou$
```

1个线程的情况（～4.5s）的完成时间比2个线程的情况（～4.8s）略短，但是两线程情况在get阶段的总工作量是原来的两倍。因此，在两个内核的get阶段，双线程情况实现了近2倍的并行加速。

注意：1）完成时间与2个线程大致相同，但是此运行获得的结果是2个线程的两倍。我们正在实现良好的并行性。2）2个线程的输出表明缺少许多键。在运行过程中，可能缺少更多或更少的钥匙。如果使用1个线程运行，将永远不会丢失任何键。

二、代码修改

2.1 为什么会有2个或更多线程而不是1个线程丢失键？识别可能导致两个线程丢失键的事件序列。

可能线程1在线程0执行完 `insert` 函数中的 `e->next = n;` 后执行了 `insert` 操作，此时新增的两个Entry都指向n，并且线程1的Entry作为链表头放入了 `bucket` ，然后线程0的Entry也会放入，这样就覆盖了线程1的那个，就是丢失了一个key。（尽管线程1的key1与线程0的key2不相等，但当 `key1%NBUCKET == key2%NBUCKET == i` 时，就有可能同时想插入 `bucket[i]` 中）

为了避免这种事件序列，在 `put` 和 `get` 中插入lock和unlock语句，以使丢失的键数始终为0。相关的pthread调用为：

```
pthread_mutex_t lock;      // declare a lock
pthread_mutex_init(&lock, NULL);   // initialize the lock
pthread_mutex_lock(&lock);  // acquire lock
pthread_mutex_unlock(&lock);  // release lock
```

修改如下：

1. 在开始处定义5个互斥锁

```
// 定义5个互斥锁
pthread_mutex_t bucket_locks[NBUCKET];
```

2. 在put()中加入互斥锁

```
static
void put(int key, int value)
{
  int i = key % NBUCKET;
    pthread_mutex_lock(&bucket_locks[i]); // Acquire lock
  insert(key, value, &table[i], table[i]);
    pthread_mutex_unlock(&bucket_locks[i]); // Release lock
}
```

3. 在main()中加入对锁的初始化

```
    // initialize the lock
    for (i=0; i < NBUCKET; i++) {
        pthread_mutex_init(&bucket_locks[i], NULL);
    }
```

测试结果如下：

```
(base) houyuqiandeMacBook-Pro:threads yuqianhou$ gcc -g -O2 ph.c -pthread
(base) houyuqiandeMacBook-Pro:threads yuqianhou$ ./a.out 2
1: put time = 0.016578
0: put time = 0.017014
0: get time = 4.296980
0: 0 keys missing
1: get time = 4.298733
1: 0 keys missing
completion time = 4.315940
```

丢失的key已为0。

2.2 首先使用1个线程测试代码，然后使用2个线程测试它。是否正确（即是否消除了丢失的key？）？两线程版本是否比单线程版本更快?

测试结果如下：

```
(base) houyuqiandeMacBook-Pro:threads yuqianhou$ ./a.out 1
0: put time = 0.007681
0: get time = 4.372720
0: 0 keys missing
completion time = 4.380600
(base) houyuqiandeMacBook-Pro:threads yuqianhou$ ./a.out 2
1: put time = 0.015055
0: put time = 0.015361
0: get time = 4.405803
0: 0 keys missing
1: get time = 4.411926
1: 0 keys missing
completion time = 4.427441
```

丢失的key已被消除，两线程的版本比单线程更快。

2.3 修改您的代码，以使 `get` 操作在保持正确性的同时并行运行。（提示：在此应用程序中，为了 `get` 正确性，是否需要进行锁定？）

`get()` 只是遍历哈希表，并不会改变哈希表，所以加不加锁都不会导致key丢失。所以单纯只讨论key会不会丢失时， `get()` 里不用加锁

2.4 修改您的代码，以便某些 `put` 操作可以并行运行，同时保持正确性。（提示：每个bucket都有锁吗？）

为防止两个线程同时往同一个bucket里insert，因此需要给每个bucket都加上lock。

完整代码如下：

```
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>
#include <assert.h>
#include <pthread.h>
#include <sys/time.h>

#define SOL
#define NBUCKET 5
#define NKEYS 100000

struct entry {
  int key;
  int value;
  struct entry *next;
};
struct entry *table[NBUCKET];
int keys[NKEYS];
int nthread = 1;
volatile int done;

//pthread_mutex_t lock;     // declare a lock
//pthread_mutex_init(&lock, NULL);   // initialize the lock
//pthread_mutex_lock(&lock);  // acquire lock
//pthread_mutex_unlock(&lock);  // release lock

```

```
// 定义5个互斥锁
pthread_mutex_t bucket_locks[NBUCKET];

double
now()
{
 struct timeval tv;
 gettimeofday(&tv, 0);
 return tv.tv_sec + tv.tv_usec / 1000000.0;
}

static void
print(void)
{
  int i;
  struct entry *e;
  for (i = 0; i < NBUCKET; i++) {
    printf("%d: ", i);
    for (e = table[i]; e != 0; e = e->next) {
      printf("%d ", e->key);
    }
    printf("\n");
  }
}

static void
insert(int key, int value, struct entry **p, struct entry *n)
{
  struct entry *e = malloc(sizeof(struct entry));
  e->key = key;
  e->value = value;
  e->next = n;
  *p = e;
}

static
void put(int key, int value)
{
  int i = key % NBUCKET;
    pthread_mutex_lock(&bucket_locks[i]); // Acquire lock
  insert(key, value, &table[i], table[i]);
    pthread_mutex_unlock(&bucket_locks[i]); // Release lock
}

static struct entry*
get(int key)
{
  struct entry *e = 0;
```

```
75  //    pthread_mutex_lock(&bucket_locks[key % NBUCKET]); // Acquire
    lock
76    for (e = table[key % NBUCKET]; e != 0; e = e->next) {
77      if (e->key == key) break;
78    }
79  //    pthread_mutex_unlock(&bucket_locks[key % NBUCKET]); // Release
    lock
80    return e;
81  }
82
83  static void *
84  thread(void *xa)
85  {
86    long n = (long) xa;
87    int i;
88    int b = NKEYS/nthread;
89    int k = 0;
90    double t1, t0;
91
92    //  printf("b = %d\n", b);
93    t0 = now();
94    for (i = 0; i < b; i++) {
95      // printf("%d: put %d\n", n, b*n+i);
96      put(keys[b*n + i], n);
97    }
98    t1 = now();
99    printf("%ld: put time = %f\n", n, t1-t0);
100
101   // Should use pthread_barrier, but MacOS doesn't support it ...
102   __sync_fetch_and_add(&done, 1);
103   while (done < nthread) ;
104
105   t0 = now();
106   for (i = 0; i < NKEYS; i++) {
107     struct entry *e = get(keys[i]);
108     if (e == 0) k++;
109   }
110   t1 = now();
111   printf("%ld: get time = %f\n", n, t1-t0);
112   printf("%ld: %d keys missing\n", n, k);
113   return NULL;
114 }
115
116 int
117 main(int argc, char *argv[])
118 {
119   pthread_t *tha;
120   void *value;
121   long i;
```

```
  double t1, t0;

  if (argc < 2) {
    fprintf(stderr, "%s: %s nthread\n", argv[0], argv[0]);
    exit(-1);
  }
  nthread = atoi(argv[1]);
  tha = malloc(sizeof(pthread_t) * nthread);
  srandom(0);
  assert(NKEYS % nthread == 0);
  for (i = 0; i < NKEYS; i++) {
    keys[i] = random();
  }
    // initialize the lock
    for (i=0; i < NBUCKET; i++) {
        pthread_mutex_init(&bucket_locks[i], NULL);
    }
  t0 = now();
  for(i = 0; i < nthread; i++) {
    assert(pthread_create(&tha[i], NULL, thread, (void *) i) == 0);
  }
  for(i = 0; i < nthread; i++) {
    assert(pthread_join(tha[i], &value) == 0);
  }
  t1 = now();
  printf("completion time = %f\n", t1-t0);
}
```

至此，题目中的基本功能已成功实现。