Multithreading-Using threads

20307130350 信息安全 陈丹纯

实验内容

分析

1. There is something wrong in `notxv6/ph.c`, thus making the code do not function as expected when working with multi-threads.
2. Check out the `ph.c` to find out the reasons, especially in `put()` and `insert()`. Write them in the `answers-thread.txt`.
3. Modify the `ph.c` especially `put()` and `get()` to lock and unlock the **critical sections**. Make sure the modified can pass the `ph_safe` by running `make grade` in terminal.
4. Try more times to optimize the performances to pass the `ph_fast`.

原文

more details in [Lab: Multithreading (mit.edu)](#).

实验分析

实验代码和answers-thread.txt已上传git。

多线程出错的原因

因为table是global variable，所以当两个及以上多个线程正好在同一个table中put()时，以两个线程为例：

1. **error**: `insert()` 中的插入链表操作可能会导致覆盖。eg.当线程0运行完`e->next=n;`之后发生context switch，线程1也执行put`e->next=n`之后并继续执行`*p=e`；再切换为线程0执行`*p=e`,先执行`*p=e`的线程会丢失新添加的key。
2. **warning**: put()中执行`e->value=value`来更新存在的key时会导致覆盖，导致不同的线程丢失率不同。比如两个线程都要修改同一个 entry, 但由于调度先后执行`e->value=value`，则先执行该语句的会丢失更新的key，另一线程得以保留。因此两个线程对key的丢失不同。
3. **error**: 两个线程一个在执行`put()`时恰好执行到`insert(key,value,&table,table[i])`时，切换到另一个线程执行`insert()`中的`*p=e*`，如果针对同一个table，再切换回来会导致`put()`中的`insert(args)`中的`table`变化，导致key丢失。

```c
static void *
put_thread(void *xa)
{
  int n = (int) (long) xa; // thread number
  int b = NKEYS/nthread;

  for (int i = 0; i < b; i++) {
    put(keys[b*n + i], n);
  }

  return NULL;
}
```

proc.h ✕ swtch.S ✕ ph.c ✕

```c
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(lock+i);
  // is the key already present?
  struct entry *e = 0;
  for (e = table[i]; e != 0; e = e->next) {
    if (e->key == key)
      break;
  }
  if(e){
    // update the existing key.
    e->value = value;
  } else {
    // the new is new.
    insert(key, value, &table[i], table[i]);
  }
  pthread_mutex_unlock(lock+i);

}

```

Lock方案

根据上面多线程出错原因分析，是对全局变量`table`操作时需要互斥。要保证0 missing，只需解决`原因1`和`原因3`即`insert()`中对entry节点的覆盖对其他线程`insert()`或者`put()中insert()`。因此有多种加锁方案。

1. 对hash表的每个table的`insert`（smallest critical section）都加互斥锁

1. 因为要对于每个table都加锁，定义**全局变量**`lock[NBUCKET]`；

```
pthread_mutex_t lock[NBUCKET];
```

2. 在`main()`中初始化锁。

```
for(int i=0;i<NBUCKET;i++){
    pthread_mutex_init(lock+i,NULL);
}
```

3. `the smallest critical section`为：

```
insert(key, value, &table[i], table[i]);
```

```
} else {
  // the new is new.
  pthread_mutex_lock(lock+i);
  insert(key, value, &table[i], table[i]);
  pthread_mutex_unlock(lock+i);
}
```

对应的实验结果：

```
utegan@ubuntu:~/xv6-labs-2021$ make ph
gcc -o ph -g -O2 -DSOL_THREAD -DLAB_THREAD notxv6/ph.c -pthread
utegan@ubuntu:~/xv6-labs-2021$ ./ph 2
100000 puts, 3.247 seconds, 30796 puts/second
0: 0 keys missing
1: 0 keys missing
200000 gets, 6.476 seconds, 30885 gets/second
utegan@ubuntu:~/xv6-labs-2021$ ./ph 4
100000 puts, 3.985 seconds, 25095 puts/second
1: 0 keys missing
2: 0 keys missing
3: 0 keys missing
0: 0 keys missing
400000 gets, 20.798 seconds, 19233 gets/second
utegan@ubuntu:~/xv6-labs-2021$ make grade
make clean
```

```
make[1]: Leaving directory '/home/utegan/xv6-labs-2021'
ph_safe: OK (9.7s)
== Test ph_fast == make[1]: Entering directory '/home/utegan/xv6-labs-
make[1]: 'ph' is up to date.
make[1]: Leaving directory '/home/utegan/xv6-labs-2021'
ph_fast: OK (24.6s)
== Test barrier == make[1]: Entering directory '/home/utegan/xv6-labs-
gcc -o barrier -g -O2 -DSOL_THREAD -DLAB_THREAD notxv6/barrier.c -pthr
```

2. 直接对所有table的smallest critical section加锁。

因为无需对每个table分别加锁，所有的table都加同样的锁,只需定义并初始化一个lock。

```
  // the new is new.
  pthread_mutex_lock(lock);
  insert(key, value, &table[i], table[i]);
  pthread_mutex_unlock(lock);
}
```

结果：

```
make[1]: Leaving directory '/home/utegan/xv6-labs-2021'
ph_safe: OK (13.2s)
== Test ph_fast == make[1]: Entering directory '/home/utegan/xv6-labs-2021'
make[1]: 'ph' is up to date.
make[1]: Leaving directory '/home/utegan/xv6-labs-2021'
ph_fast: OK (26.7s)
== Test barrier == make[1]: Entering directory '/home/utegan/xv6-labs-2021'
gcc -o barrier -g -O2 -DSOL_THREAD -DLAB_THREAD notxv6/barrier.c -pthread
make[1]: Leaving directory '/home/utegan/xv6-labs-2021'
```

3. 直接对put()加锁。

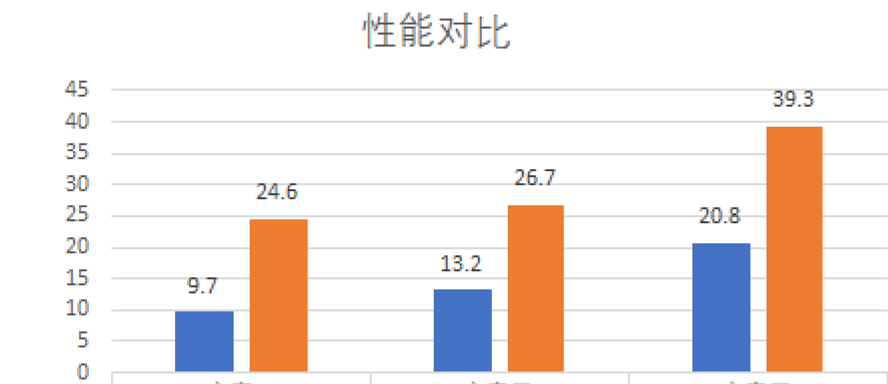
```
for (int i = 0; i < b; i++) {
  pthread_mutex_lock(lock);
  put(keys[b*n + i], n);
  pthread_mutex_unlock(lock);
}
```

结果：

```
ph_safe: OK (20.8s)
== Test ph_fast == make[1]: Entering directory '/home/utegan/xv6-labs-2021'
make[1]: 'ph' is up to date.
make[1]: Leaving directory '/home/utegan/xv6-labs-2021'
ph_fast: FAIL (39.3s)
    Parallel put() speedup is less than 1.25x
== Test barrier == make[1]: Entering directory '/home/utegan/xv6-labs-2021'
gcc -o barrier -g -O2 -DSOL_THREAD -DLAB_THREAD notxv6/barrier.c -pthread
```

实验结果分析

不同方案性能分析



	方案一	方案二	方案三
ph_safe	9.7	13.2	20.8
ph_fast	24.6	26.7	39.3

原因分析

1. 显然方案三不是最小critical section，其`ph_fast`也没有达标。
2. 下面重点分析方案一和方案二：
    1. 两者都是对smallest critical section加锁。
    2. 但是方案一的性能比方案二更高的原因在于方案一**分别对每个table设置了一个特定的锁**，因此当且仅当当不同线程对同一个`table`（即`i`相同时）执行`insert`时，才会互斥。
    3. 方案二是对所有的`table`**不加区分**地设置了**同一个锁**，因此当不同线程执行`insert`的时候都会发生互斥，和`i`无关，频率更高，对性能的影响自然更大。