lab Multithreading

lab 实现

exercise 0

`user/uthread.c`和`user/uthread_switch.S`实现了用户进程的线程创建和调度。

每一个线程有一个独立的线程栈，线程状态和线程context（线程切换时需要保存和恢复）。

`thread_init`初始化0线程，即`main()`所在线程，0线程需要context用于`thread_switch`时保存context，但0线程并不会再次被调度。

`thread_create`创建一个新的线程，将第一个状态为FREE的线程状态设置为RUNNABLE，并且重新设置ra和sp便于第一次被调度能正确执行传入的函数。

`thread_schedule`进行线程调度，找到一个状态RUNNABLE的非0线程，并和现有的进程完成context switch，使调度到新的线程。

具体的完善代码见exercise 1。

exercise 1 uthread

完成用户程序层面的线程切换，在了解了xv6进程切换和调度的过程后实现起来非常简单。首先是为`struct thread`增加一个`struct context`用来记录线程切换时线程的context（寄存器）。`struct context`和进程context一样保存callee-saved寄存器和ra, sp寄存器，因为caller-saved寄存器在函数调用时已经保存在了栈中。

```
// Saved registers for user thread context switches.
struct context {
  uint64 ra;
  uint64 sp;

  // callee-saved
  uint64 s0;
  uint64 s1;
  uint64 s2;
  uint64 s3;
  uint64 s4;
  uint64 s5;
  uint64 s6;
  uint64 s7;
  uint64 s8;
  uint64 s9;
  uint64 s10;
  uint64 s11;
};

struct thread {
  char       stack[STACK_SIZE]; /* the thread's stack */
  int        state;             /* FREE, RUNNING, RUNNABLE */
  struct context context;
};
```

接下来在`thread_schedule()`中增加切换context的语句，需要保存上一个线程的context，恢复下一个线程的context。

```
  if (current_thread != next_thread) {         /* switch threads?  */
    next_thread->state = RUNNING;
    t = current_thread;
    current_thread = next_thread;
    /* YOUR CODE HERE
     * Invoke thread_switch to switch from t to next_thread:
     * thread_switch(??, ??);
     */
    thread_switch((uint64)&t->context, (uint64)&next_thread->context);
  }
```

在`thread_create(void (*func)())`中创建线程，由于要保证线程在`thread_schedule()`第一次被schedule时能执行`thread_create`时传入的函数时，在`thread_create`中需要设置好线程context的ra和sp，保证在该线程第一次被调度时ra设置成函数地址，sp设置成线程的栈，从而返回在线程的栈上执行函数。

```
  t->state = RUNNABLE;
  // when scheduler choose the thread, it's ra is the func and sp is the stack
top.
  t->context.ra = (uint64)func;
  t->context.sp = (uint64)&t->stack[STACK_SIZE-1];
```

第一个exercise完成，uthread创建了三个线程打印1-100的数，当一个线程打印完一行就切换到另一个线程。

```
thread_c: exit after 100
thread_a: exit after 100
thread_b: exit after 100
thread_schedule: no runnable threads
```

exercise Using threads

利用UNIX的线程提高hash table的效率。对于问题

```
Why are there missing keys with 2 threads, but not with 1 thread? Identify a
sequence of events with 2 threads that can lead to a key being missing.

回答：当两个线程同时写入hash table时有可能发生这样一种情况，两个进程同时获得了相同的
&table[i]，即同一个bucket的首元素，而后将两个元素在首元素之前插入新的entry时，只有后插入的
entry在bucket里（只有后一个*p = e才修改了bucket的首元素），这就造成了一些key的遗失。
```

由此我们需要对hash table增加lock保证在`put`操作时只有一个线程在对一个bucket进行操作。

首先为每一个bucket加上一个lock。

```
struct bucket {
  struct entry* head;
  pthread_mutex_t lock;
}table[NBUCKET];
```

相应的需要在`main`函数中增加对lock的初始化，在first puts之前初始化即可。

```
  for (int i = 0; i < NBUCKET; ++i){
    pthread_mutex_init(&table[i].lock, NULL);
  }
```

修改`put`，加入`pthread_mutex_lock`和`phread_mutex_unlock`保证线程在bucket中寻找相应entry，修改entry和插入entry时只有当前线程在进行操作。

```
 pthread_mutex_lock(&table[i].lock);
  for (e = table[i].head; 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]);
  }
  pthread_mutex_unlock(&table[i].lock);
```

`insert`函数也跟着改一下。

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

可以看见两个线程的puts操作比一个线程的puts操作时间减少了大约33%，同时也通过了ph_fast测试。

```
yik@ubuntu:~/xv6-labs-2020$ make ph
gcc -o ph -g -O2 notxv6/ph.c -pthread
yik@ubuntu:~/xv6-labs-2020$ ./ph 1
100000 puts, 6.288 seconds, 15904 puts/second
0: 0 keys missing
100000 gets, 6.273 seconds, 15942 gets/second
yik@ubuntu:~/xv6-labs-2020$ ./ph 2
100000 puts, 4.220 seconds, 23699 puts/second
0: 0 keys missing
1: 0 keys missing
200000 gets, 6.300 seconds, 31745 gets/second
```

exercise Barrier

目的是让所有的线程在达到某一点时要等待所有的进程都达到这一点时才能继续。

`struct barrier`内设置了一个互斥锁和一个条件锁，互斥锁保证只有一个线程对`bstate`进行修改，条件锁保证线程在该条件变量的控制下block，直到被`pthread_cond_broadcast`唤起。`nthread`记录当前到达这一点的线程数，`round`记录所有线程到达这一点的回数。

```
struct barrier {
  pthread_mutex_t barrier_mutex;
  pthread_cond_t barrier_cond;

  int nthread;      // Number of threads that have reached this round of the
barrier
  int round;     // Barrier round
} bstate;
```

在一个线程对`bstate`进行修改前先获取互斥锁，`bstate.nthread`加一，如果`bstate.nthread == nthread`，则`bstate.rountd`加一，调用`pthread_cond_broadcast`unblock所有在该条件变量下block的线程如果还有线程没达到这一点，则调用`pthread_cond_wait`，该线程在条件变量下block，同时释放`bstate.barrier_mutex`，保证在该线程被block时其他线程能获取`bstate.barrier_mutex`并对`bstate`的变量进行修改，在该线程被unblock时再获取`bstate.barrier_mutex`。

最后不要忘了释放`bstate.barrier_mutex`，让下一个进入`barrier`的线程能获取互斥锁。

```
static void
barrier()
{
  // YOUR CODE HERE
  //
  // Block until all threads have called barrier() and
  // then increment bstate.round.
  //

  pthread_mutex_lock(&bstate.barrier_mutex);
  // ++bstate.nthread;
  if(++bstate.nthread == nthread){
    ++bstate.round;
    bstate.nthread = 0;
    pthread_cond_broadcast(&bstate.barrier_cond);
  }
  else{
    pthread_cond_wait(&bstate.barrier_cond, &bstate.barrier_mutex);
  }
  pthread_mutex_unlock(&bstate.barrier_mutex);
}
```

至此所有的exercise就完成了

```
== Test uthread ==
$ make qemu-gdb
uthread: OK (3.8s)
== Test answers-thread.txt == answers-thread.txt: OK
== Test ph_safe == make[1]: Entering directory '/home/yik/xv6-labs-2020'
make[1]: 'ph' is up to date.
make[1]: Leaving directory '/home/yik/xv6-labs-2020'
ph_safe: OK (11.9s)
== Test ph_fast == make[1]: Entering directory '/home/yik/xv6-labs-2020'
make[1]: 'ph' is up to date.
make[1]: Leaving directory '/home/yik/xv6-labs-2020'
ph_fast: OK (24.9s)
== Test barrier == make[1]: Entering directory '/home/yik/xv6-labs-2020'
make[1]: 'barrier' is up to date.
make[1]: Leaving directory '/home/yik/xv6-labs-2020'
barrier: OK (12.7s)
== Test time ==
time: OK
Score: 60/60
```

lab 总结

这个lab虽然代码量很小，难度不大，但却学到了很多多线程编程的知识。首先是线程如何切换，切换时如何保存和恢复context。其次是利用UNIX提供的库函数进行多线程编程，多线程可以有效提高程序的效率，但是多线程要求保证一些变量和状态的不变性，这就需要一种数据结构——锁。互斥锁只允许持有锁的线程对被锁保护的数据结构的修改，条件锁让线程的block和unblock受到条件变量的控制，便于对thread group进行控制。