Lab3 Multithreading

- 姓名：吴欣怡
- 学号：PB21051111
- 虚拟机用户名：OS-PB21051111

Uthread: switching between threads

实验分析

分析实验文档中的要求:
1.修改user/uthread_switch.s 中的thread_switch，使之能够实现： 保存被切换线程的寄存器，恢复切换到线程的寄存器，并返回到后一个线程指令中最后停止的点
2.修改struct thread 以保存寄存器（增加一个结构体）
3.修改user/uthread.c中的thread_create()，正确设置相关的寄存器。
3.修改user/uthread.c中的thread_schedule()，调用thread_switch，正确执行线程切换。

实验过程

第一步

理解代码内容和其它准备工作：
找到kernel/proc.h中定义了进程使用到的寄存器信息（context），直接把proc.h文件引用到uthread.c文件中。向struct thread中增加一个context结构体以保存寄存器

```
// Saved registers for kernel context switches.
struct context {
  uint64 ra;  //ra表示返回地址寄存器
  uint64 sp;  //sp表示栈指针寄存器

  // 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 threadContext;

};
```

据此修改uthread_switch.s：
sd 指令用于将寄存器的值存储到内存中。它将当前的寄存器状态保存到内存中的地址 a0 指向的位置。
ld 指令用于从内存中加载值到寄存器中。它从地址 a1 指向的位置加载之前保存的寄存器状态。
ret 指令，用于返回到 ra 寄存器中存储的地址。可以使控制流返回到之前暂停执行的线程或任务，以便恢复其执行状态。

```
thread_switch:
    sd ra, 0(a0)
    sd sp, 8(a0)
    sd s0, 16(a0)
    sd s1, 24(a0)
    sd s2, 32(a0)
    sd s3, 40(a0)
    sd s4, 48(a0)
    sd s5, 56(a0)
    sd s6, 64(a0)
    sd s7, 72(a0)
    sd s8, 80(a0)
    sd s9, 88(a0)
    sd s10, 96(a0)
    sd s11, 104(a0)

    ld ra, 0(a1)
    ld sp, 8(a1)
    ld s0, 16(a1)
    ld s1, 24(a1)
    ld s2, 32(a1)
    ld s3, 40(a1)
    ld s4, 48(a1)
    ld s5, 56(a1)
    ld s6, 64(a1)
    ld s7, 72(a1)
    ld s8, 80(a1)
    ld s9, 88(a1)
    ld s10, 96(a1)
    ld s11, 104(a1)

    ret    /* return to ra */
```

修改thread_switch在user/uthread.c中的定义

```
extern void thread_switch(struct context*,struct context*);
```

第二步

修改thread_create()函数，之前的部分已经把结构体、状态定义好了，只需要补充好ra和sp两个寄存器的值就行。
其中，ra寄存器定义为func的地址，即执行线程时从此处开始执行。
使线程的栈指针sp指向线程的堆栈顶部。

```
void
thread_create(void (*func)())
{
  struct thread *t;

  for (t = all_thread; t < all_thread + MAX_THREAD; t++) {
    if (t->state == FREE) break;
  }
  t->state = RUNNABLE;
  // YOUR CODE HERE
  t->threadContext.ra=(uint64)func;
  t->threadContext.sp=(uint64)(t->stack)+ STACK_SIZE;
}
```

修改thread_schedule()，只需要调用thread_switch函数即可。根据注释，需要从t转为next_thread，所以thread_switch的参数为t和next_thread的threadContext部分。

```
  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(&t->threadContext,&next_thread->threadContext);
  } else
    next_thread = 0;
```

Using threads

实验分析

回答问题：为什么两个线程都丢失了键，而不是一个线程?
丢失keys的原因是put中对entry进行写操作时，可能有另外一个线程同时对entry进行写操作，那么两个线程其中一个的修改就会丢失，在get操作时就会找不到key对应的entry。若两个线程的操作交叉运行，则后运行的一个会被前运行的一个覆盖，使得插入的节点少一个，又因为这种行为是不可预测的，所以两个线程都有可能丢失。
1.结合pthread调用指令，在notxv6/ph.c中的put和get中插入lock和unlock语句，使得在两个线程中丢失的键数始终为0。

实验过程

第一步

查看ph.c，原始的Put会造成键丢失的原因的分析如上。那么我们需要通过锁来限制两个线程同时做写操作。初始化一个锁数组，其长度和哈希表的bucket数等长。

```
struct entry *table[NBUCKET];
int keys[NKEYS];
int nthread = 1;
pthread_mutex_t lock[NBUCKET];
```

数组table的元素table[i]是一个链表，链表中存放元素entry，那么并行访问不同的table不会导致修改数据的丢失。在put函数中，针对每一个table[i]调用相应的锁，这样不同线程在访问不同table[i]时无需相互等待。

```
static
void put(int key, int value)
{
  int i = key % NBUCKET;
  pthread_mutex_lock(&lock[i]);  //获得锁
  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]);  //释放锁
}
```

第二步

验证可知，的确没有丢失键了。而且与单线程相比，双线程实现了并行加速。

```
ubuntu@VM7782-OS-PB21051111:/home/ubuntu/桌面/xv6-labs-2020$ ./ph 2
100000 puts, 6.486 seconds, 15418 puts/second
1: 0 keys missing
0: 0 keys missing
200000 gets, 11.075 seconds, 18059 gets/second
```

Barrier

实验分析

1.结合pthread调用指令，在notxv6/barrier.c中的barrier()函数。
需要处理的情况有：已到达屏障的线程在其它线程到达屏障之前要等待；做好轮数记录和bstate.nthread的更新。

实验过程

barrier本身需要nthread计数和加互斥锁。考虑到pthread_cond_wait 在调用时释放mutex ，并在返回前重新获取nutex的性质。只需要两类处理barrier:所有线程均已达到，进入下一轮，更新nthread，唤醒所有线程；还有线程未到达，是当前线程进入睡眠。

```
static void
barrier()
{
  // YOUR CODE HERE
  //
  // Block until all threads have called barrier() and
  // then increment bstate.round.
  //
  pthread_mutex_lock(&bstate.barrier_mutex);  //加锁
  bstate.nthread+=1;  //barrier被调用，计数器加一
  if(bstate.nthread<nthread)
  { //还有线程未调用barrier
     pthread_cond_wait(&bstate.barrier_cond,&bstate.barrier_mutex);   //在cond上进入睡眠，释放锁mutex，在醒来时重新获取
  }
  else
  {   //所以线程均已调用barrier
      bstate.round+=1;  //轮数加一
      bstate.nthread=0;  //对于下一轮的nthread清零
      pthread_cond_broadcast(&bstate.barrier_cond); //唤醒线程
  }
  pthread_mutex_unlock(&bstate.barrier_mutex);  //释放锁


}
```

测试barrier性能：

```
ubuntu@VM7782-OS-PB21051111:/home/ubuntu/桌面/xv6-labs-2020$ make barrier
make: “barrier”已是最新。
ubuntu@VM7782-OS-PB21051111:/home/ubuntu/桌面/xv6-labs-2020$ ./barrier 2
OK; passed
ubuntu@VM7782-OS-PB21051111:/home/ubuntu/桌面/xv6-labs-2020$ ./barrier 3
OK; passed
ubuntu@VM7782-OS-PB21051111:/home/ubuntu/桌面/xv6-labs-2020$ ./barrier 1
OK; passed
```

实验评分

```
ubuntu@VM7782-OS-PB21051111:/home/ubuntu/桌面/xv6-labs-2020$ python3 grade-lab-t
hread
make: “kernel/kernel”已是最新。
== Test uthread == uthread: OK (3.1s)
== Test answers-thread.txt == answers-thread.txt: OK
== Test ph_safe == make: “ph”已是最新。
ph_safe: OK (19.8s)
== Test ph_fast == make: “ph”已是最新。
ph_fast: OK (45.2s)
== Test barrier == make: “barrier”已是最新。
barrier: OK (2.5s)
== Test time ==
time: OK
Score: 60/60
```

实验总结

对于进程之间的协调和锁的正确使用有了更多理解。