实验三 Multithreading

PB20050987 梁兆懿

在实验3中，我阅读了xv6的第七章，学习了有关线程调度的知识。在实验三中，我们将熟悉多线程处理，在实验中实现多个线程的切换以及线程屏障的操作。

实验需要切换分支。

```
ubuntu@VM5878-LZY:/home/ubuntu/桌面/xv6-labs-2020$ git fetch
ubuntu@VM5878-LZY:/home/ubuntu/桌面/xv6-labs-2020$ git checkout thread
```

Uthread: switching between threads

该实验需要在用户级的线程系统上实现线程的切换。我们可以在uthread.c和uthread_switch.S两个文件在看到构建uthread程序的规则。实验需要我们补充创建线程和在线程之间切换的代码。

实验代码

在uthread.c文件中，我们定义一个结构体threadcontext，为用于在线程切换时，保存线程上下文的寄存器组。

```
struct threadcontext {
    uint64 ra;
    uint64 sp;
    uint64 s0;
    uint64 s1;
    uint64 s2;
    uint64 s3;
    uint64 s4;
    uint64 s5;
    uint64 s6;
    uint64 s7;
    uint64 s8;
    uint64 s9;
    uint64 s10;
    uint64 s11;
};
```

其中 ra为指向线程要运行的函数的寄存器，sp为指向线程自己的栈的寄存器。

当然，我们也要在struct thread里加上该结构体的定义：

```
struct thread{
    struct threadcontext context;
    }
```

在uthread_switch.S中，我们实现线程上下文的切换，为汇编语言。参考swtch.S：

```
    .globl thread_switch
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)
```

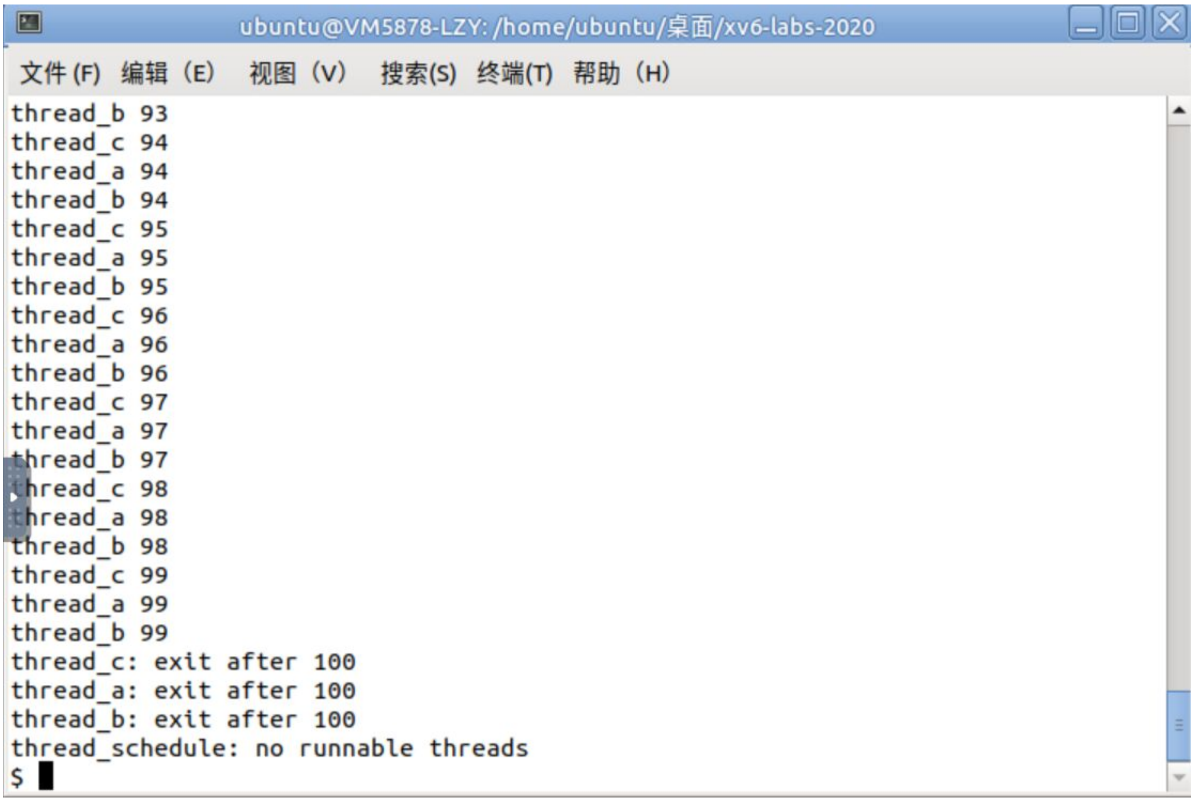
接下来我们就可以实现线程的切换了。首先在线程创建时，我们要令context的寄存器ra和sp指向对应的函数以及栈，因此我们在 thread_create() 函数里添加相关赋值：

```
 t->context.ra = (uint64)func;
 t->context.sp = (uint64)&t->stack[STACK_SIZE-1];
```

最后，我们在thread_schedule()里面调用thread_switch函数：

```
thread_switch(
        (uint64)&t->context,
        (uint64)&next_thread->context
    );
```

测试结果



Using threads

该实验需要探索并解决线程切换过程中序列丢失的问题。在测试中，哈希表的大量keys在线程程切换中丢失。因此，我们要在ph.c文件中的put get函数添加相应的lock和unlock来保护线程序列。

实验代码

实验要求在answers_thread.txt中添加简短说明：

```
answer:In the insert fuction, when a thread, called A, needs to insert a node into the list, at the same time, the B thread need to insert another node. In this case, the node B' address will be covered by A.
```

在函数insert中，我们看到当A,B线程同时需要插入结点时，A线程节点地址将会福官b线程节点地址，因此我们需要在insert处加上锁。在ph.c文件中：

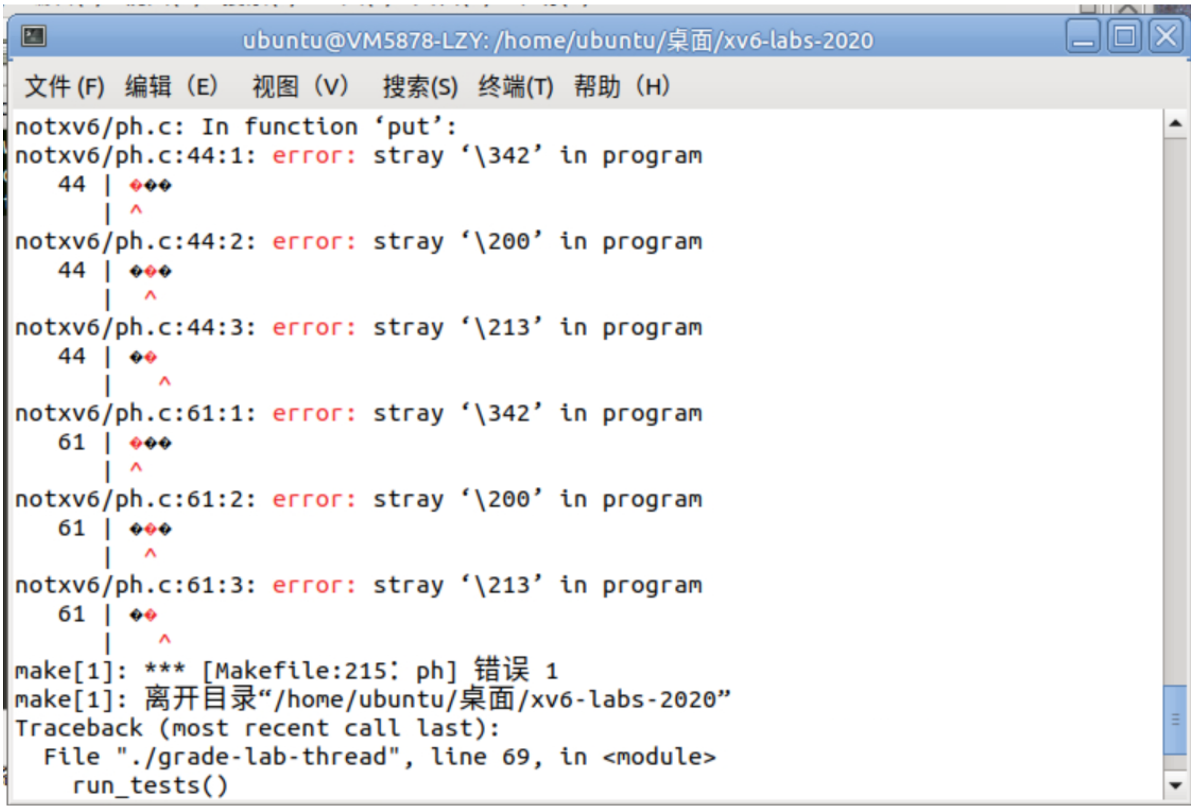
```
pthread_mutex_t lock;

static
void put(int key, int value)
{
  int i = key % NBUCKET;
  struct entry *e = 0;
  for (e = table[i]; e != 0; e = e->next) {
    if (e->key == key)
      break;
  }
  pthread_mutex_lock(&lock);
  if(e){
    e->value = value;
  } else {
    insert(key, value, &table[i], table[i]);
  }
```

```
    pthread_mutex_unlock(&lock);
}
```

与此同时，我们在main函数加上对应声明

```
int main(int argc, char *argv[])
{
    pthread_mutex_init(&lock, NULL);
}
```

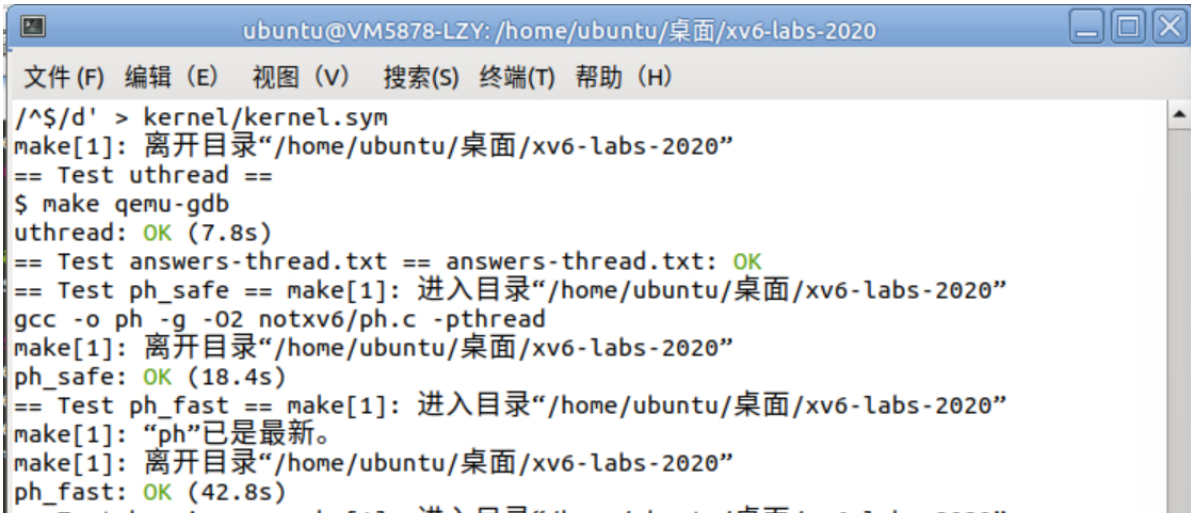
然而这并不能通过：



实验要求我们双线程需要首先并行加速，即两个线程实现时仅能以1.25倍的速度于单线程。在这种情况下，单个锁并不能满足实验要求。因此我们定义多个锁来分别对各个线程插入时进行控制。用一下代码分别替换上述代码：

```
pthread_mutex_t locks[NBUCKET];//在文件开头声明锁

pthread_mutex_lock(locks + i);//在put函数中调用锁
pthread_mutex_unlock(locks + i);

for (int i = 0; i < NBUCKET; i++) {//在main函数初始化锁
        pthread_mutex_init(locks + i, NULL);
    }
```

测试结果



Barrier

该实验要求在进程中实现一个屏障，使得线程在到达该点时要等待其他线程到达才能离开。实验给出了barrier.c文件，需要我们补充相关代码。

实验代码

我们添加一个barrier函数，在bstate.nthread上进行计数，等到所有线程都通过后，即可跳过pthread_cond_wait(&cond, &mutex)，并且调用pthread_cond_broadcast(&cond)释放被屏障的线程

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

运行结果

```
ubuntu@VM5878-LZY: /home/ubuntu/桌面/xv6-labs-2020
文件(F) 编辑(E) 视图(V) 搜索(S) 终端(T) 帮助(H)
riscv64-linux-gnu-objdump -S kernel/kernel > kernel/kernel.asm
riscv64-linux-gnu-objdump -t kernel/kernel | sed '1,/SYMBOL TABLE/d; s/ .* / /;
/^$/d' > kernel/kernel.sym
make[1]: 离开目录“/home/ubuntu/桌面/xv6-labs-2020”
== Test uthread ==
$ make qemu-gdb
uthread: OK (6.5s)
== Test answers-thread.txt == answers-thread.txt: OK
== Test ph_safe == make[1]: 进入目录“/home/ubuntu/桌面/xv6-labs-2020”
make[1]: “ph”已是最新。
make[1]: 离开目录“/home/ubuntu/桌面/xv6-labs-2020”
ph_safe: OK (18.1s)
== Test ph_fast == make[1]: 进入目录“/home/ubuntu/桌面/xv6-labs-2020”
make[1]: “ph”已是最新。
make[1]: 离开目录“/home/ubuntu/桌面/xv6-labs-2020”
ph_fast: OK (43.5s)
== Test barrier == make[1]: 进入目录“/home/ubuntu/桌面/xv6-labs-2020”
make[1]: “barrier”已是最新。
make[1]: 离开目录“/home/ubuntu/桌面/xv6-labs-2020”
barrier: OK (2.6s)
== Test time ==
time: OK
Score: 60/60
ubuntu@VM5878-LZY:/home/ubuntu/桌面/xv6-labs-2020$
```

思考题

1. Uthread: switching between threads: thread_switch needs to save/restore only the callee-save registers. Why?

Because the caller-save register compiler already saves it, as the caller-save register is saved before the thread_switch is called, so we only need to manually save the ones that are not saved.

因为caller-save寄存器编译器会自动保存，在调用thread_switch前caller-save寄存器已经被自动保存了，因此我们只需要手动保存没有被自动保存的。

2. Using threads: 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.

```
30 static void
31 insert(int key, int value, struct entry **p, struct entry *n)
32 {
33   struct entry *e = malloc(sizeof(struct entry));
34   e->key = key;
35   e->value = value;
36   e->next = n;
37   *p = e;
38 }
39
```

In this function, when two threads insert nodes into the list at the same time, A executes to line 37 and points a->next to the linked list header node. At this time, the system switches to the B thread and inserts B into the header of the linked list, at which time the value stored in p is the address of b. A executes line 38, covering *p as the address of a. At this point, we lost B's address and could no longer access B. Thus, the key is missing.

在此函数中，当两个线程同时将节点插入列表时，A 执行到第 37 行并指向链表头节点旁边的 a->。此时，系统切换到 B 线程，将 B 插入链表的标头中，此时存储在 p 中的值就是 b 的地址。A 执行第 38 行，将 *p 作为 a 的地址。此时，我们丢失了 B 的地址，无法再访问 B。因此，将会丢失哈希表的密钥。

实验感受

本次实验带有三个子实验，虽然任务量较大，但由于积累了前面两次实验的经验，加之本人在线程这一部分掌握比较好，本次实验进行得比较顺利，收获颇丰。