

### 一、Uthread: switching between threads

要求实现一个简单的用户级线程的功能，框架已经写好了，只需要填一些关键的代码就行，原理和内核中的原理基本是一致的，需要保存相应的寄存器。

首先定义用户线程的上下文，并将其添加到线程结构中：

```
struct ucontext
{
    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 ucontext context;
};
```

我们要让每个线程在自己的栈上运行，所以需要在线程创建的时候将其上下文的 `sp` 指针设置到自己的栈，为了使调度器第一次切换到线程的时候能够运行相应的线程函数，所以还需要将返回地址寄存器 `ra` 设置到函数地址：

```
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->context.ra = (uint64)func;
    t->context.sp = (uint64)t->stack + STACK_SIZE;
}
```

在线程切换的时候，保存线程 1 的所有寄存器至其上下文中，并恢复线程 2 上下文中的所有寄存器：

```
.globl thread_switch
thread_switch:
/* YOUR CODE HERE */
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 */
```

运行结果如下：

```
init: starting sh
$ uthread
thread_a started
thread_b started
thread_c started
thread_c 0
thread_a 0
thread_b 0
thread_c 1
thread_a 1
thread_b 1
thread_c 2
thread_a 2
thread_b 2
thread_c 3
thread_a 3
thread_b 3
thread_c 4
thread_a 4
thread_b 4
thread_c 5
thread_a 5
thread_b 5
thread_c 6
thread_a 6
thread_b 6
thread_c 7
thread_a 7
thread_b 7
thread_c 8
thread_a 8
thread_b 8
thread_c 9
thread_a 9
thread_b 9
thread_c 10
thread_a 10
thread_b 10
```

## 二、Using threads

题目提供了一份多线程并行存取哈希表的程序，但是没有做同步处理，所以当运行的线程大于 1 时会出现 race condition 并出错，题目要求的是使用 `pthread_mutex` 加锁来消除竞争。

首先定义 `NBUCKET` 把锁（每个桶一把锁，当不同线程存不同桶的时候不会发生竞争，达到并行的效果），并在 `main` 中对锁进行初始化：

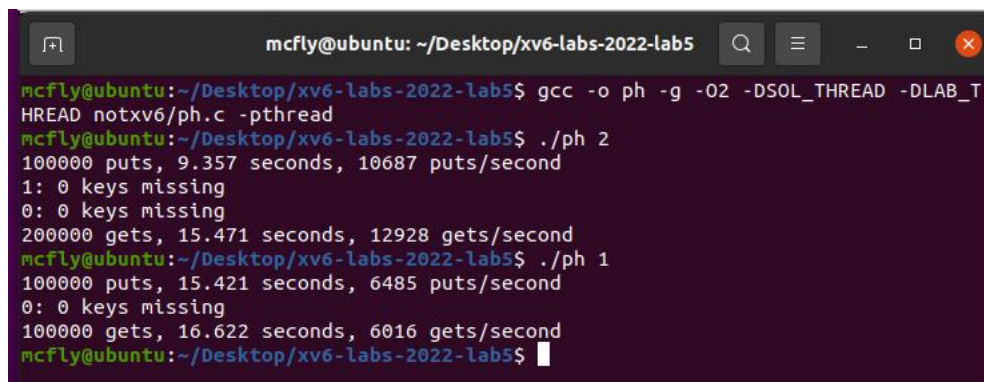
```
pthread_mutex_t locks[NBUCKET];

int
main(int argc, char *argv[])
{
    pthread_t *tha;
    void *value;
    double t1, t0;

    if (argc < 2) {
        fprintf(stderr, "Usage: %s nthreads\n", argv[0]);
        exit(-1);
    }
    nthread = atoi(argv[1]);
    tha = malloc(sizeof(pthread_t) * nthread);
    srand(0);
    assert(NKEYS % nthread == 0);
    for (int i = 0; i < NKEYS; i++) {
        keys[i] = random();
    }

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

运行结果如下：



```
mcfly@ubuntu: ~/Desktop/xv6-labs-2022-lab5
mcfly@ubuntu:~/Desktop/xv6-labs-2022-lab5$ gcc -o ph -g -O2 -DSOL_THREAD -DLAB_T
HREAD notxv6/ph.c -pthread
mcfly@ubuntu:~/Desktop/xv6-labs-2022-lab5$ ./ph 2
100000 puts, 9.357 seconds, 10687 puts/second
1: 0 keys missing
0: 0 keys missing
200000 gets, 15.471 seconds, 12928 gets/second
mcfly@ubuntu:~/Desktop/xv6-labs-2022-lab5$ ./ph 1
100000 puts, 15.421 seconds, 6485 puts/second
0: 0 keys missing
100000 gets, 16.622 seconds, 6016 gets/second
mcfly@ubuntu:~/Desktop/xv6-labs-2022-lab5$
```

可以看到速度提升了，但是不到 2 倍。

## 三、Barrier

本题主要使用条件变量，设置 `barrier` 的主要目标是为了同步线程。

当线程进入 `barrier` 时，我们先加锁，然后将线程计数加 1，若当前线程不是最后一个到达的则让其等待(wait)，最后一个到达的将 `round` 加 1，然后叫醒其它所有的线程：

```
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) {
        pthread_cond_broadcast(&bstate.barrier_cond);
        bstate.nthread = 0;
        bstate.round++;
    } else {
        // go to sleep on cond, releasing lock mutex, acquiring upon wake up
        pthread_cond_wait(&bstate.barrier_cond, &bstate.barrier_mutex);
    }

    pthread_mutex_unlock(&bstate.barrier_mutex);
}
```

#### 四、问题回答

为什么两个线程会丢失 keys，但是一个线程不会？

当两个线程同时进入 insert，并且 put 中的变量 i 相同，两个线程就会先后对于 table[i] 进行赋值，后插入的 key 会覆盖掉前一个线程插入的 key，从而产生 key 丢失的情况。