Lab 5

Part A: Uthread: switching between threads

本实验实现线程间的环境切换。首先在 thread 结构体中，添加变量 heap 数组，用以存放寄存器数据：

```
struct thread {
  uint64      heap[32];
  char        stack[STACK_SIZE]; /* the thread's stack */
  int         state;             /* FREE, RUNNING, RUNNABLE */
};
```

其中 ,heap 中数值先后对应 ra ,sp ,s0-s11 寄存器中保存内容。由于 caller-saved register，或 volatile register 会在线程切换之前将存储内容转移至堆栈中，因此不需要保存，只需保存 callee-saved register （ non-volatile register)的内容即可。之后 ,在 thread_create，亦即创建线程的阶段，向创建的 thread 结构体中 heap 的第 0、1 个元素（分别对应寄存器 ra、sp ）传入返回地址以及堆栈指针……

```
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->heap[0] = (uint64)func;
  t->heap[1] = (uint64)&t->stack[STACK_SIZE-1];
}
```

……并在 thread_schedule 的相应部分调用线程切换函数。其中 ,函数的两个变量为 thread 结构体对应堆数组的首地址，因此在 thread_switch 中我们可以通过事先约定好的寄存器顺序直接访存相关数据。

```
thread_switch((uint64)t->heap, (uint64)next_thread->heap);
```

thread_switch 代码如下。

```
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。

```
thread_c 98
thread_a 98
thread_b 98
thread_c 99
thread_a 99
thread_b 99
thread_c: exit after 100
thread_a: exit after 100
thread_b: exit after 100
thread_schedule: no runnable threads
$ QEMU: Terminated
```

Part B: Using threads

在多线程的情况下，假设线程 A 与线程 B 交替运行，设想以下情况：线程 A 在执行 put 操作时，在工作指针已经指向数组尾部，但尚未插入新 key 的情况下，系统转而执行线程 B。此时，线程 B 视野下的数组尾和 A 是一致的，因此，如果此时线程 B 先完成了插入表元的操作，系统进而继续执行 A，那么 A 将会在 B 插入表元的位置再插入新表元，B 插入的表元因而被覆盖，从而造成了 key 丢失的现象。

```
(base) justin@chensipengdeMacBook-Pro notxv6 % ./ph 1
100000 puts, 1.887 seconds, 52984 puts/second
0: 0 keys missing
100000 gets, 1.863 seconds, 53671 gets/second
(base) justin@chensipengdeMacBook-Pro notxv6 % ./ph 2
100000 puts, 1.064 seconds, 93987 puts/second
1: 179 keys missing
0: 179 keys missing
200000 gets, 1.930 seconds, 103621 gets/second _
```

在不考虑效率的情况下，解决 key 丢失的方法可以非常直接：只需要让 put 函数成为一个完整的原子操作即可。

```
static
void put(int key, int value)
{
  pthread_mutex_lock(&lock);
  int i = key % NBUCKET;

  // 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);
}
```

值得一提的是，get 函数即使不是原子操作也不会造成 key 丢失，因为 get 只涉及对数组元素的读操作，而不涉及写操作。完成以上修改后，可在如下的输出结果中看到 key 丢失的问题已经解决。

```
(base) justin@chensipengdeMacBook-Pro notxv6 % ./ph 1
100000 puts, 1.892 seconds, 52846 puts/second
0: 0 keys missing
100000 gets, 1.903 seconds, 52560 gets/second
(base) justin@chensipengdeMacBook-Pro notxv6 % ./ph 2
100000 puts, 2.096 seconds, 47706 puts/second
1: 0 keys missing
0: 0 keys missing
200000 gets, 1.973 seconds, 101359 gets/second
```

现在，尝试对锁进行优化，从而提升多线程的工作效率。首先，尽可能缩小上锁部分代码：由于对数组信息读取的部分不必是原子操作，因此仅对写部分上锁即可。另外，由于 table 中不同 bucket 的读写操作相互独立，因此应当允许对不同 bucket 的操作可以交替并行执行，亦即，对每个 bucket 单独分配一个互斥锁，而不应该简单地为整个表分配一个单独的锁。

以下为 main 函数中的互斥锁初始化部分：

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

修改后的 put 函数为如下所示：

```
static
void put(int key, int value)
{
  int i = key % NBUCKET;

  // is the key already present?
  struct entry *e = 0;
  for (e = table[i]; e != 0; e = e->next) {
    if (e->key == key)
      break;
  }
  pthread_mutex_lock(&lock[i]);
```

```
  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]);
}
```

完成以上所有修改后重新运行 ph 脚本。

```
(base) justin@chensipengdeMacBook-Pro notxv6 % ./ph 2
100000 puts, 0.987 seconds, 101307 puts/second
1: 0 keys missing
0: 0 keys missing
200000 gets, 1.912 seconds, 104583 gets/second
```

Part C: Barrier

调用 barrier 功能时，首先获得互斥锁，使得对 nthread 的增加成为原子操作，确保函数能够正确执行。随后进行条件判断：若当前状态下 nthread 变量尚未达到线程总数，则通过 cond_wait 功能解开互斥锁并等待其他线程；若已经达到线程总数，则通过 cond_broadcast 唤醒所有正在等待的线程从中断处继续执行，并将 nthread 值置为零（nthread 表示当前到达 barrier 的线程总数），round 值增加 1。

```
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.round++;
    bstate.nthread = 0;
  }

  else{
    pthread_cond_wait(&bstate.barrier_cond, &bstate.barrier_mutex);
  }
  pthread_mutex_unlock(&bstate.barrier_mutex);
}
```

测试结果如下。

```
(base) justin@chensipengdeMacBook-Pro notxv6 % ./barrier 1
OK; passed
(base) justin@chensipengdeMacBook-Pro notxv6 % ./barrier 2
OK; passed
(base) justin@chensipengdeMacBook-Pro notxv6 % ./barrier 3
OK; passed
(base) justin@chensipengdeMacBook-Pro notxv6 % ./barrier 5
OK; passed
(base) justin@chensipengdeMacBook-Pro notxv6 % ./barrier 10
OK; passed
```

由于个人目前配置的 lab 环境为 amazon aws 单核云服务器，无法正确进行 ph 实验，因此借用了同学电脑运行 make grade。

```
== Test uthread ==
$ make qemu-gdb
uthread: OK (3.3s)
== Test answers-thread.txt == answers-thread.txt: OK
== Test ph_safe == make[1]: 进入目录“/home/wsj/下载/xv6-lab51”
gcc -o ph -g -O2 -DSOL_THREAD -DLAB_THREAD notxv6/ph.c -pthread
make[1]: 离开目录“/home/wsj/下载/xv6-lab51”
ph_safe: OK (12.1s)
== Test ph_fast == make[1]: 进入目录“/home/wsj/下载/xv6-lab51”
make[1]: “ph”已是最新。
make[1]: 离开目录“/home/wsj/下载/xv6-lab51”
ph_fast: OK (26.8s)
== Test barrier == make[1]: 进入目录“/home/wsj/下载/xv6-lab51”
gcc -o barrier -g -O2 -DSOL_THREAD -DLAB_THREAD notxv6/barrier.c -pthread
make[1]: 离开目录“/home/wsj/下载/xv6-lab51”
barrier: OK (12.3s)
== Test time ==
time: OK
Score: 60/60
```

Thoughts

本实验以多线程为主题，实现了进程切换时寄存器数据的存储和 barrier，并通过多线程并

行读写的情境探索了读写操作的原子性及锁对进程执行效率的影响，加深了对线程、锁即条件变量的理解。