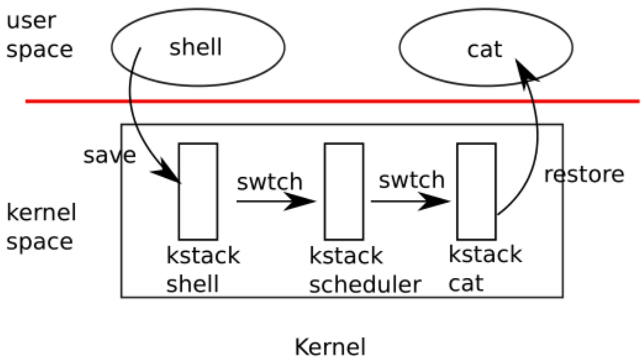
Lab7: Multithreading 实验报告

线程调度过程如下图所示：



线程调度的过程主要是保存 `context` 上下文状态，因为这里的切换全都是以函数调用的形式，因此这里只需要保存被调用者保存的寄存器（Callee-saved register）即可，调用者的寄存器会自动保存。

---

1. 首先切换分支：

> ```
> $ git fetch
> $ git checkout thread
> $ make clean
> ```

Uthread: switching between threads (moderate)

实现一个用户线程调度的方法。

> 这里的“线程”是完全用户态实现的，多个线程也只能运行在一个 CPU 上，并且没有时钟中断来强制执行调度，需要线程函数本身在合适的时候主动 yield 释放 CPU。

1. 上下文切换：

> 借鉴 *kernel/swtch.S*，在*user/uthread_switch.S* 中需要实现上下文切换的代码
>
> 调用者寄存器被c 编译器保存在thread_schedule 的堆栈中

```
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 */
```

2. 定义上下文字段：

   借鉴 *kernel/proc.h* 中的上下文结构体，在 *user/uthread.c* 中定义一个 `context` 结构体，用于保存 ra、sp 以及 callee-saved registers，并加入到 `thread` 结构体中。

```c
// Saved registers for user 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;            /* 存储线程上下文切换时保存的寄存器 */
};
```

3. 在*kernel/uthread.c*中的thread_schedule 函数中调用 thread_switch 进行上下文切换

```
  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) &current_thread->context);
  } else
    next_thread = 0;
}
```

4. 补充*kernel/uthread.c*中的 thread_create 函数：将线程函数的入口地址保存到返回地址 ra 中， 使得 `thread_switch` 最后返回到该地址，从而运行线程代码；将 sp 指向栈底（由于栈从高到低生长，所以 sp 应该指向栈的最高地址）

```
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 - 1);
}
```

5. 执行测试命令： `uthread`

```
220110512@comp6:~/xv6-labs-2020$ make qemu
qemu-system-riscv64 -machine virt -bios none -kernel kernel/kernel -m 128M -smp 3 -nographic -drive fil
e=fs.img,if=none,format=raw,id=x0 -device virtio-blk-device,drive=x0,bus=virtio-mmio-bus.0

xv6 kernel is booting

hart 1 starting
hart 2 starting
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 96
thread_b 96
thread_c 97
thread_a 97
thread_b 97
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
$ 
```

Using threads (moderate)

1. 文件*notxv6/ph.c*包含一个简单的哈希表，如果单个线程使用，该哈希表是正确的，但是多个线程使用时，该哈希表是不正确的。

   键入 `make ph` ，`./ph 1` 和 `./ph 2` 可以看到，当有两个线程同时向哈希表中添加条目时，实现了较优的并行性能，但两个线程均丢失了16550个键，表明大量本应在哈希表中的键不存在在哈希表中

   ```
   220110512@comp6:~/xv6-labs-2020$ make ph
   gcc -o ph -g -O2 notxv6/ph.c -pthread
   220110512@comp6:~/xv6-labs-2020$ ./ph 1
   100000 puts, 11.695 seconds, 8551 puts/second
   0: 0 keys missing
   100000 gets, 12.050 seconds, 8299 gets/second
   220110512@comp6:~/xv6-labs-2020$ htop
   220110512@comp6:~/xv6-labs-2020$ ./ph 2
   100000 puts, 5.637 seconds, 17739 puts/second
   1: 16550 keys missing
   0: 16550 keys missing
   200000 gets, 12.814 seconds, 15608 gets/second
   ```

2. 回答问题：为什么多线程会导致键的缺失，而单键值不会？

   设计这样一个序列：

   > [假设键 k1、k2 属于同个 bucket]
   >
   > - thread 1: 尝试设置 k1
   > - thread 1: 发现 k1 不存在，尝试在 bucket 末尾插入k1
   >
   > ---scheduler 切换到 thread 2---
   >
   > - thread 2: 尝试设置 k2
   > - thread 2: 发现 k2 不存在，尝试在 bucket 末尾插入 k2
   > - thread 2: 分配 entry，在桶末尾插入 k2

> ---scheduler 切换回 thread 1---
>
> - thread 1: 分配 entry，没有意识到 k2 的存在，在其认为的 “桶末尾” （实际为 k2 所处位置）插入 k1
>
> [k1 被插入，但是由于被 k1 覆盖，k2 从桶中消失了，引发了键值丢失]

由此造成了键值缺失

3. 利用加锁操作，解决哈希表 race-condition 导致的数据丢失问题

如果只加一个锁，锁的粒度很大，会导致丢失性能，结果还不如不加锁的单线程。因此需要将锁的粒度减小，为每个槽位（bucket）加一个锁。

在 ***notxv6/ph.c*** 中定义lock锁，并修改put、get、main函数，在适当位置加锁和解锁：

```
pthread_mutex_t lock[NBUCKET];
```

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

  pthread_mutex_lock(&lock[i]);
  // 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[i]);
}
```

```
static struct entry*
get(int key)
{
  int i = key % NBUCKET;

  pthread_mutex_lock(&lock[i]);
  struct entry *e = 0;
  for (e = table[i]; e != 0; e = e->next) {
    if (e->key == key) break;
  }
  pthread_mutex_unlock(&lock[i]);
  return e;
}
```

```c
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);
  srandom(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(&lock[i], NULL);
  }
```

4. 键入 `make ph` ，`./ph 1` 和 `./ph 2` ：

```
220110512@comp6:~/xv6-labs-2020$ make ph
gcc -o ph -g -O2 notxv6/ph.c -pthread
220110512@comp6:~/xv6-labs-2020$ ./ph 1
100000 puts, 11.910 seconds, 8396 puts/second
0: 0 keys missing
100000 gets, 11.591 seconds, 8627 gets/second
220110512@comp6:~/xv6-labs-2020$ ./ph 2
100000 puts, 6.822 seconds, 14658 puts/second
0: 0 keys missing
1: 0 keys missing
200000 gets, 13.384 seconds, 14943 gets/second
220110512@comp6:~/xv6-labs-2020$
```

实现了较优的并行性能，且不存在键值缺失

Barrier(moderate)

实验实现思是：加锁，然后判断到达屏障点的线程数，如果所有线程都已到达就调用pthread_cond_broadcast接口唤醒其他线程，否则就调用pthread_cond_wait进行等待。

1. 在 *notxv6/barrier.c* 文件中完成barrier函数：

   采用生产者消费者模式，如果还有线程没到达，就加入到队列中，等待唤起；如果最后一个线程到达了，就将轮数加一，然后唤醒所有等待这个条件变量的线程

```c
static void
barrier()
{
  // YOUR CODE HERE
  //
  // Block until all threads have called barrier() and
  // then increment bstate.round.
  //
  pthread_mutex_lock(&bstate.barrier_mutex);
  if(++bstate.nthread == nthread) {
    // 记录等待进程数的计数器清零
    bstate.nthread = 0;
    // 屏障轮数加一
    bstate.round++;
    // 唤醒等待在此点上的所有进程
    pthread_cond_broadcast(&bstate.barrier_cond);
  } else {
    // 等待
    pthread_cond_wait(&bstate.barrier_cond, &bstate.barrier_mutex);
  }
  pthread_mutex_unlock(&bstate.barrier_mutex);
}
```

2. 键入`make barrier`，`./barrier 1`、`./barrier 2`和`./barrier 4`：

```
220110512@comp6:~/xv6-labs-2020$ make barrier
gcc -o barrier -g -O2 notxv6/barrier.c -pthread
220110512@comp6:~/xv6-labs-2020$ ./barrier 1
OK; passed
220110512@comp6:~/xv6-labs-2020$ ./barrier 2
OK; passed
220110512@comp6:~/xv6-labs-2020$ ./barrier 4
OK; passed
220110512@comp6:~/xv6-labs-2020$
```

结果截图

执行命令`make grade`

```
make[1]: Leaving directory '/home/students/220110512/xv6-labs-2020'
== Test uthread ==
$ make qemu-gdb
uthread: OK (4.7s)
== Test answers-thread.txt == answers-thread.txt: OK
== Test ph_safe == make[1]: Entering directory '/home/students/220110512/xv6-labs-2020'
make[1]: 'ph' is up to date.
make[1]: Leaving directory '/home/students/220110512/xv6-labs-2020'
ph_safe: OK (20.5s)
== Test ph_fast == make[1]: Entering directory '/home/students/220110512/xv6-labs-2020'
make[1]: 'ph' is up to date.
make[1]: Leaving directory '/home/students/220110512/xv6-labs-2020'
ph_fast: OK (43.2s)
== Test barrier == make[1]: Entering directory '/home/students/220110512/xv6-labs-2020'
make[1]: 'barrier' is up to date.
make[1]: Leaving directory '/home/students/220110512/xv6-labs-2020'
barrier: OK (3.9s)
== Test time ==
time: OK
Score: 60/60
220110512@comp6:~/xv6-labs-2020$
```