实验报告

Lab 5

1. Uthread:switching between threads

1.1 实验要求

我们需要实现一个用户级的线程系统，包括线程创建、上下文切换机制。
xv6中，`user/uthread.c`和`user/uthread_switch.S`是我们需要修改的文件。主要修改以下几个方面：

1. 修改`thread_create()`函数，使得线程在自己的堆栈上允许传入的函数；
2. 修改`thread_schedule()`，完成线程调度的工作；
3. 修改`thread_switch()`(汇编代码)，完成上下文切换的工作。

1.2 实验基础

在正式开始修改前，我们先了解一下`uthread`是怎样实现线程内容的。
线程有3个状态:FREE、RUNNING和RUNNABLE。线程未初始化时状态为FREE，初始化后为RUNNABLE。当线程正在运行时状态为RUNNING，切换后，状态改为RUNNABLE。调度函数的工作就是寻找RUNNABLE的线程，并切换到该线程上运行。

```
#define STACK_SIZE  8192
#define MAX_THREAD  4



struct thread {
  char       stack[STACK_SIZE]; /* the thread's stack */
  int        state;             /* FREE, RUNNING, RUNNABLE */
};
struct thread all_thread[MAX_THREAD];
struct thread *current_thread;
```

线程用结构体`thread`表示，由一个大小为8192B(即8KB)的线程栈和状态构成。所有线程被记录于大小为4的数组`all_thread`中，`current_thread`是指向当前线程的指针。

1.3 thread_create()

线程创建时，传入了一个函数指针参数`func`。首先遍历整个`all_thread`数组，并找到状态为FREE的线程用于创建线程，将线程状态设置为RUNNABLE。

接下来。我们应该将函数指针保存到ra寄存器上，这样在`thread_switch()`后，新线程将ra的内容复制到实际的ra寄存器上，从而通过`ret`指令返回时可以进入函数`func()`并执行。于是线程结构`thread`需要新添加一系列用于保存的寄存器，内容可以参考`kernel/proc.h`中上下文结构`context`的内容。

```
struct thread {
  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;

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

这些寄存器除了返回地址寄存器ra和栈指针sp外，还要保存相应的被调用者寄存器，这样就能保证线程切换后，仍能保存线程上的临时变量。

此外，还需要设置栈指针sp的位置。它初始应指向线程栈的底部，这样在具体的函数执行中，能够利用栈指针定位到线程栈，并不断扩张以存放线程上执行的函数的临时变量。(可以make后查

看`uthread.asm`进行对照查看)

以上的部分都可以参考`kernel/proc.c allocproc()`实现。

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

1.4 thread_switch()

switch所做的工作非常简单：将旧线程的寄存器保存到旧线程结构上，从新线程结构上的寄存器恢复新线程的寄存器。这部分代码可以直接参考xv6的内核实现`kernel/swtch.S`。

```
    .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 */
```

1.5 thread_schedule()

在找到可切换的线程后，只需要调用`thread_switch()`切换线程即可。

```
    ...
    /* YOUR CODE HERE
     * Invoke thread_switch to switch from t to next_thread:
     * thread_switch(??, ??);
     */
    thread_switch((uint64)t,(uint64)next_thread);
    ...
```

1.6 实验结果

`make qemu`后执行`uthread`结果如图:

```
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
```

`make grade`的结果为:

```
== Test uthread ==
$ make qemu-gdb
uthread: OK (5.3s)
```

2. Using threads

2.1 实验目的

本实验基于hash table来探究线程和锁的并行编程。通过向hash table插入100000个键，统计时间，再检查hash table的key，看有多少键丢失了。

在1个线程下向hash table添加表项，不会产生任何丢失；在2个线程下，添加速率更快了，但产生了很多表项的丢失。

我们需要:

1. 在`notxv6/ph.c`中的`put()`和`get()`中添加lock和unlock语句，使得并发安全。
2. 加锁的同时保证高性能，即尽可能少的使用锁。

2.2 实验基础

先看一下`ph.c`的具体实现。

```
#define NBUCKET 5
#define NKEYS 100000

struct entry {
  int key;
  int value;
  struct entry *next;
};
struct entry *table[NBUCKET];
int keys[NKEYS];
int nthread = 1;
```

`entry`是一个链表，包含key和value两个属性。`table`是大小为5的链表数组，即一个`table`有5个entry(5个不同的桶)。总键数为100000。

`insert()`函数将一个新表项(指定key和value)插入到链表的头部，并更新桶的内容(更新头部)。

`put()`函数先对key取NBUCKET的模(**这样，除以5余数不同的key被分配到不同的桶上**)。先遍历一遍这个桶看是否有该key了。如果找到，就更新桶中该key的value为新value，否则插入该表项到桶上。

`get()`函数简单地遍历桶查看是否有指定key对应的表项，如果有就返回，否则返回0。

`put_thread()`首先将总的key按线程数平分(`main()`函数确保可以平分)，然后每个线程通过`put()`添加表项，表项的value都是线程编号。

`get_thread()`通过`get()`检查所有的100000个key，记录返回值为0的数目(即缺失项)。

`main()`函数为100000个key分配随机值，然后利用`pthread`库函数创建线程并执行put和get。

2.3 为什么丢失key?

单个线程执行时，显然是顺序执行，并不会产生任何丢失。

当2个线程执行时，我们思考一下可能丢失key的原因：

首先，2个线程put的key应当是不同的(假定随机数基本不会产生相同的数，因为范围很大)，第一个线程put keys[0:49999]，第二个线程put keys[50000:99999]。当执行`put()`函数时，各自获得各自的桶号(**函数内部的局部变量是线程安全的**)，没有找到key时为这个桶添加表项。

关键就在这里：`insert()`函数是一个简单的插入链表头部，且对函数参数e和p进行了全局的修改，那么考虑如下情形：假设两个线程的桶号一致，insert时它们的key被插入到同一个桶，当`insert()`函数如下执行时：

```
  struct entry *e = malloc(sizeof(struct entry));    // t1
  e->key = key;                                      // t1
  e->value = value;                                  // t1
  e->next = n;                                       // t1
  struct entry *e = malloc(sizeof(struct entry));    // t2
  e->key = key;                                      // t2
  e->value = value;                                  // t2
  e->next = n;                                       // t2
  *p = e;                                            // t2
  *p = e;                                            // t1
```

可以看到，线程1的表项被遗弃，在这个桶上只插入了线程2的表项，从而丢失了key。

2.4 添加lock

为了避免上述错误，应该为`put()`函数添加锁。根据提示，我们只需要为每个桶维护一个锁，因为不同的桶之间不会出现上述的问题。

在`notxv6/ph.c`开头声明一个锁数组：

```
// my edit
pthread_mutex_t lock[NBUCKET];
// edit ends
```

在`main()`中初始化锁：

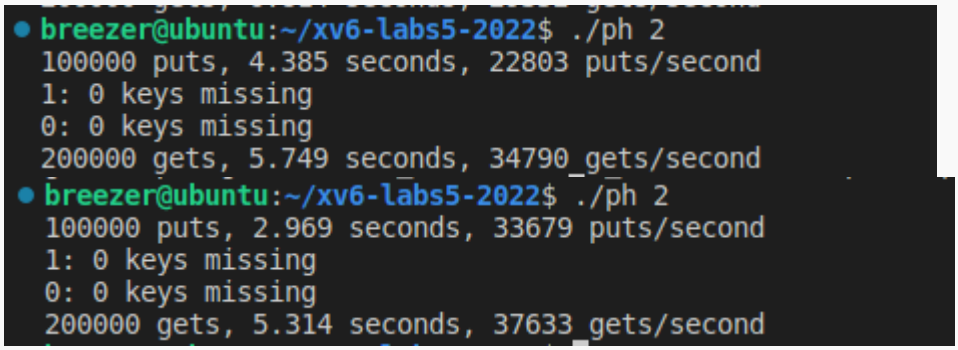
```
// my edit
  for(int i = 0;i < NBUCKET;++i)
    pthread_mutex_init(&lock[i],NULL);
// edit ends
```

在`put()`函数中添加相应的lock和unlock函数。

```
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;
  }
  if(e){
    // update the existing key.
    e->value = value;
  } else {
    // the new is new.
    pthread_mutex_lock(&lock[i]);
    insert(key, value, &table[i], table[i]);
    pthread_mutex_unlock(&lock[i]);
  }
}
```

这里只需要在需要insert的情况下进行lock和unlock即可，能提升一定性能。下面是对全局加锁和只对insert加锁的性能对比:



另外也不需要为`get()`函数加锁，该函数并不会产生并发错误。

2.5 实验结果

执行`make grade`的结果如图:

```
== Test answers-thread.txt == answers-thread.txt: OK
== Test ph_safe == make[1]: Entering directory '/home/breezer/xv6-labs5-2022'
gcc -o ph -g -O2 -DSOL_THREAD -DLAB_THREAD notxv6/ph.c -pthread
make[1]: Leaving directory '/home/breezer/xv6-labs5-2022'
ph_safe: OK (8.0s)
== Test ph_fast == make[1]: Entering directory '/home/breezer/xv6-labs5-2022'
make[1]: 'ph' is up to date.
make[1]: Leaving directory '/home/breezer/xv6-labs5-2022'
ph_fast: OK (18.2s)
```

3. Barrier

3.1 实验目的

需要实现一个barrier，使得当一个线程到达barrier时，必须要等待其他所有的线程都到达这个barrier。

我们需要使用条件变量来进行同步。

3.2 实验基础

先看一下`barrier.c`的实现。

```
struct barrier {
  pthread_mutex_t barrier_mutex;
  pthread_cond_t barrier_cond;
  int nthread;      // Number of threads that have reached this round of the barrier
  int round;     // Barrier round
} bstate;
```

bstate结构体由一个互斥锁、一个条件变量、记录到达本轮barrier的线程数和barrier round号组成。

通过`barrier_init()`函数进行初始化：初始化锁和条件变量，设置nthread=0。

`thread()`函数检查20000轮barrier。

`main()`函数将多个线程绑定到`thread()`函数上以检查barrier。

3.3 `barrier()`函数实现

这个函数的实现思路很简单，每调用一次就递增`bstate.nthread`，然后检查是否有`bstate.nthread == nthread`以确定是否所有线程都已到达barrier。如果都已到达，

将`bstate.nthread`清0，round加1，并调用`pthread_cond_broadcast()`函数唤醒所有等待的线程。否则，调用`pthread_cond_wait()`函数使当前线程进入睡眠状态。

注意仍然需要为每次barrier用锁保护，因为涉及到对bstate的修改。这样有一个问题出现：调用`pthread_cond_wait()`令线程睡眠后，后续是否无法获取锁？这是不会的，因为调用`pthread_cond_wait()`时会释放锁，且返回时再次获取锁，从而不会产生相应问题。

3.4 实验结果

执行`make grade`的结果如图:

```
== Test barrier == make[1]: Entering directory '/home/breezer/xv6-labs5-2022'
gcc -o barrier -g -O2 -DSOL_THREAD -DLAB_THREAD notxv6/barrier.c -pthread
make[1]: Leaving directory '/home/breezer/xv6-labs5-2022'
barrier: OK (11.2s)
== Test time ==
time: OK
Score: 60/60
```

4. 实验总结

本次实验主要研究多线程、锁和条件变量相关内容。实验1模仿内核实现上下文切换的方式，完成用户级的线程切换程序；实验2和实验3分别简单地利用锁和条件变量解决相关问题。