# 2023/10/17~2023/10/19

实验内容并不难，做了这么久是因为断断续续拖了几天，一天拿来做了第一个实验，一天拿来走马观花地过了一遍文档，第三天才发现后续两个实验的内容用我在OS课上学到的知识就已经能轻松解决了(本身也只是中等难度)。

实际实验时间不过三小时吧，下周也许会暂停一周进度推进。

我渐渐没有最初的那份劲，解决问题的兴奋感在我想到要完成对应的学习记录文档后也很快被冲散了。

## XV6第七章

### thread switch

线程切换采用的欺骗手段其实和trap如出一辙，代码并不显式地通过函数调用一类的手段切换线程，而是通过对决定线程运行状态的上下文进行保存与恢复来切换进程。这使得对于线程本身而言，线程的切换是完全透明的！在fork中也有类似的方法，通过一个伪造的（通过修改寄存器填入的）a0，来让子线程fork的返回值和父线程不同。

另外，scheduler的的引入也令我感到神奇。scheduler是上下文保存在cpu中而非内存中的线程。线程进入内核->切换上下文到scheduler->scheduler运行->切换上下文到下一线程->线程从内核中返回；scheduler的出现得以让内核获得对线程的整体控制权，scheduler作为一个调度中心出现在这里，而不是让线程们去协调这件事。

本章还详细地讨论了进程锁的获取与释放时机，我实在无法理解到其中的巧妙之处，似乎一切都是顺理成章的……

### Sleep and wakeup

神似pv操作，几乎就是pv的内部实现方式的讨论，不在此赘述。

## Lab7 Multithreads

### uthread

意在实现一个用户级的线程。因为题目明确给出了添加自己的代码的位置……但实际上还需要自己去修改thread结构体，加上自己对进程内存内容的一知半解，竟误以为进程的上下文是要放在本来定义好的thread.stack中去。后来实在没有办法，翻得内核中的进程切换相关的代码，几乎是一五一十抄下来就行。

首先修改thread结构体，添加上下文存储变量context

struct thread {

  char       stack[STACK\_SIZE]; /\* the thread's stack \*/

  int        state;             /\* FREE, RUNNING, RUNNABLE \*/

  struct context ucontext;

};

在用户级scheduler中调用Switch，切换上下文以达成切换线程的目的

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->ucontext,(uint64)&next\_thread->ucontext);

  } else

    next\_thread = 0;

线程初始化，和内核中进程初始化的代码几乎一样，需要注意的是sp指针应该指向栈底。而ra也从forkret转化为了实际传入的调用进程初始化的函数的地址，不难理解。

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

  memset(&t->ucontext,0,sizeof(struct context));

  t->ucontext.ra=(uint64)func;

  t->ucontext.sp=(uint64)&t->stack[STACK\_SIZE];

}

### usingthreads

这个实验重点是在读题啊，大意就是有一个哈希表（大小为5,实现方式为链表数组，一个哈希值对应一个链表），要把一个一维数组下标与对应值的键值对塞进去，key的哈希函数是y=x%5，key范围为0~10000。

在多进程并发的情况下，多个puts进程可能会使对方的修改无效，如下：

//put\_thread1:

//  inserting key1 to bucket 1...

//  step into insert(),asuming current bucket head is p1.

//  thread\_switch()!

//put\_thread2:

//  inserting to bucket 1...

//  step into insert(),finding the current bucket is p1,too.

//  insert the key2 to bucket 1

//  now its key2->p1.

//  and the head poiner points to key2.

//  put\_thread2 finished,thread switch()!

//put\_thread1:

//  supposed to insert before key2,but insert to p1.

//  now its key1->p1

//  and the head pointer points to key1.

//  nothing points to key2,key2 is lost!

在insert之前加锁就行了

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);

    insert(key, value, &table[i], table[i]);

    pthread\_mutex\_unlock(&lock);

  }

}

### barrier

大意是在所有进程调用并进入barrier（）之前，不得有进程离开barrier。

题目本身给出了一个barrier结构体，total是此次程序中进程的总数目。

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

  int total;

} bstate;

于是修改barrier\_init()，以对应地初始化total

static void

barrier\_init(int n)

{

  assert(pthread\_mutex\_init(&bstate.barrier\_mutex, NULL) == 0);

  assert(pthread\_cond\_init(&bstate.barrier\_cond, NULL) == 0);

  bstate.nthread = 0;

  bstate.total=n;

  printf("there are %d threads\n",n);

}

在barrier中，每进入一个进程，就把nthread++,如果nthread++后等于total，就唤醒所有进程，否则睡眠当前进程。

当然，对nthread的加减比较操作是需要上锁的。

static void

barrier()

{

  // YOUR CODE HERE

  pthread\_mutex\_lock(&bstate.barrier\_mutex);

  bstate.nthread++;

  if(bstate.nthread<bstate.total){

// go to sleep on cond, releasing lock mutex, acquiring upon wake up

pthread\_cond\_wait(&bstate.barrier\_cond, &bstate.barrier\_mutex);

  }else{

    pthread\_cond\_broadcast(&bstate.barrier\_cond);

    bstate.round++;

    bstate.nthread=0;

  }

  pthread\_mutex\_unlock(&bstate.barrier\_mutex);

}

### 总结：

这一部分实验比较简单，三个都是中等难度……加上我已经在OS课程中详尽地习得了进程的有关知识，所以做起来相当轻松