# Lab7 Multithreading

This lab will familiarize you with multithreading. You will implement switching between threads in a user-level threads package, use multiple threads to speed up a program, and implement a barrier.

本实验将使您熟悉多线程。您将在用户级线程包中实现线程之间的切换，使用多个线程来加速程序，并实现一个屏障。

## Uthread: switching between threads

### 实验目的：

在本练习中，您将为用户级线程系统设计上下文切换机制，然后实现它。为了让您开始，您的xv6有两个文件：`user/uthread.c`和`user/uthread\_switch.S`，以及一个规则：运行在`Makefile`中以构建`uthread`程`序uthread.c`包含大多数用户级线程包，以及三个简单测试线程的代码。线程包缺少一些用于创建线程和在线程之间切换的代码。

您的工作是提出一个创建线程和保存/恢复寄存器以在线程之间切换的计划，并实现该计划。完成后，`make grade`应该表明您的解决方案通过了`uthread`测试。

这个实验其实相当于在用户态重新实现一遍 xv6 内核中的 scheduler() 和 swtch() 的功能，许多代码都可以模仿前面的代码来实现。

### 实验步骤：

1. 参考’kernel/proc.h’中的’struct context’结构体，在’kernel/uthread.c’中设计’struct ctx’结构体用于保存寄存器

struct ctx {

    uint64 ra;

    uint64 sp;

    uint64 s0;

    uint64 s1;

    uint64 s2;

    uint64 s3;

    uint64 s4;

    uint64 s5;

    uint64 s6;

    uint64 s7;

    uint64 s8;

    uint64 s9;

    uint64 s10;

    uint64 s11;

};

1. 为线程结构体添加上下文字段

struct thread {

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

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

  struct thread\_context context;       // 线程上下文

};

1. 修改thread\_create()函数，使其能在线程数组中遍历，发现一个未初始化的线程就执行初始化。

  // YOUR CODE HERE

  t->context.ra = (uint64)func;

  t->context.sp = (uint64)&t->stack[STACK\_SIZE - 1];

1. 修改thread\_schedule()函数，它负责负责用户多线程间的调度。它从当前线程的位置开始，在线程数组中寻找一个 `RUNNABLE` 状态的线程进行运行，在找到线程后调用函数 `thread\_switch()`进行线程的切换。

    thread\_switch((uint64)&t->context, (uint64)&current\_thread->context);

5.参考 `swtch` 代码在user/uthread\_switch.S中实现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)

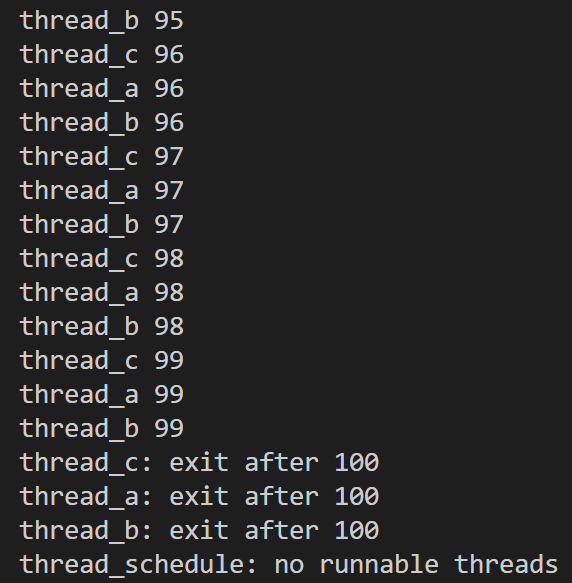
ld s8, 80(a1)

ld s9, 88(a1)

ld s10, 96(a1)

ld s11, 104(a1)

6.进入xv6，输入uthread后得到预期结果：



### 实验中遇到的问题和解决办法

在修改thread\_schedule()时要从t切换到next\_thread，不是从current\_thread切换到next\_thread

### 实验心得

熟悉了用户级线程的上下文切换机制，也是为后续多线程的实现做了铺垫。

## Using threads

### 实验目的

In this assignment you will explore parallel programming with threads and locks using a hash table. You should do this assignment on a real Linux or MacOS computer (not xv6, not qemu) that has multiple cores. Most recent laptops have multicore processors.

在本作业中，您将探索使用哈希表的线程和锁的并行编程。您应该在具有多个内核的真实Linux或MacOS计算机（不是xv6，不是qemu）上执行此任务。最新的笔记本电脑都有多核处理器。这个作业使用UNIX的pthread线程库。您可以使用man pthreads在手册页面上找到关于它的信息。

实际上就是给每个桶加上一个锁来防止同时进行操作。

### 实验步骤

1. 在ph.c中声明多线程的锁，并且在main函数里初始化

pthread\_mutex\_t lock[NBUCKET];

  // 初始化锁

  for (int i = 0; i < NBUCKET; i++){

    pthread\_mutex\_init(&lock[i], NULL);

  }

1. 在put函数里上锁

线程的安全问题是对桶中的链表进行操作而导致的，需要在链表操作的前后加锁，在`put`函数读写bucket之前加锁，在函数结束时释放锁。

  if(e){

    // update the existing key.

    e->value = value;

  } else {

    pthread\_mutex\_lock(&lock[i]);

    // the new is new.

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

    pthread\_mutex\_unlock(&lock[i]);

  }

### 实验中遇到的问题和解决办法

当不同线程对同一个bucket进行put操作时，可能会覆盖前一个put的结果，造成错误。

当整个哈希表被加锁后整体的性能甚至不如之前，原因是每一时刻只能有一个线程进行操作，相当于单线程了，优化思路是降低锁的粒度，从对于整个哈希表加锁到对于每个bucket加锁。

### 实验心得

整体实验还是比较简单的，稍微回顾一下锁的概念坐下来并不难，难的主要是在已经实现了锁的基础上怎么进行进一步优化。

## Barrier

### 实验目的

In this assignment you'll implement a [barrier](http://en.wikipedia.org/wiki/Barrier_(computer_science)): a point in an application at which all participating threads must wait until all other participating threads reach that point too. You'll use pthread condition variables, which are a sequence coordination technique similar to xv6's sleep and wakeup.

You should do this assignment on a real computer (not xv6, not qemu).

在本作业中，您将实现一个屏障`Barrier`：应用程序中的一个点，所有参与的线程在此点上必须等待，直到所有其他参与线程也达到该点。您将使用`pthread`条件变量，这是一种序列协调技术，类似于xv6的`sleep`和`wakeup`。

实验步骤：

1. 实验一个屏障’barrier()’函数，它首先会加锁然后判断到达屏障点的线程数量，如果所有线程都到达了就唤醒其他线程，否则就wait.

static void

barrier()

{

  // YOUR CODE HERE

  //

  // Block until all threads have called barrier() and

  // then increment bstate.round.

  //

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

  // 所有进程都要调用一次barrier

  if (++bstate.nthread == nthread) {

    ++bstate.round; // 轮数++

    bstate.nthread = 0; // 重置nthread

    pthread\_cond\_broadcast(&bstate.barrier\_cond); // 唤醒所有的睡眠线程

  }

  else {

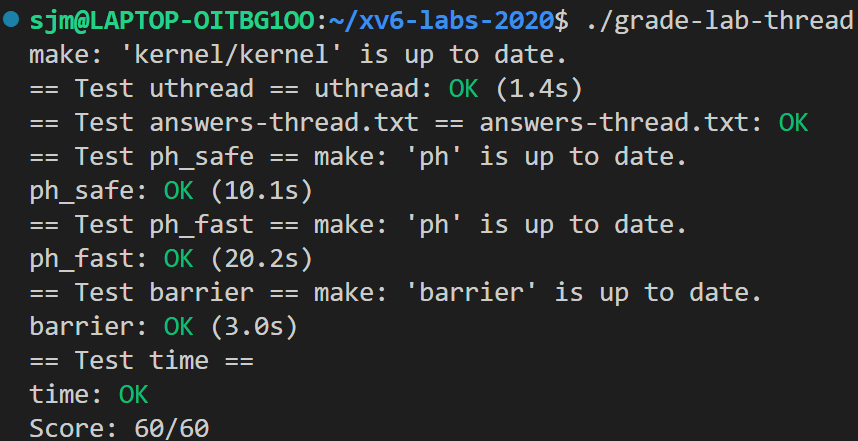
    pthread\_cond\_wait(&bstate.barrier\_cond, &bstate.barrier\_mutex); // 等待

  }

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

}

退出xv6，在终端运行./grade-lab-thread



得到的结果符合预期。

### 实验中遇到的问题和解决办法

问题不大，主要就是不明确上锁的时机要去不断调试，看看在什么时候加锁的效率会高一些。

### 实验小结

运用锁来提高效率是一种很重要的方法，其中选取上锁的粒度和上锁的时机很重要，关乎着能否提高效率，如果粒度选取不好，甚至会加大时间和空间开销。