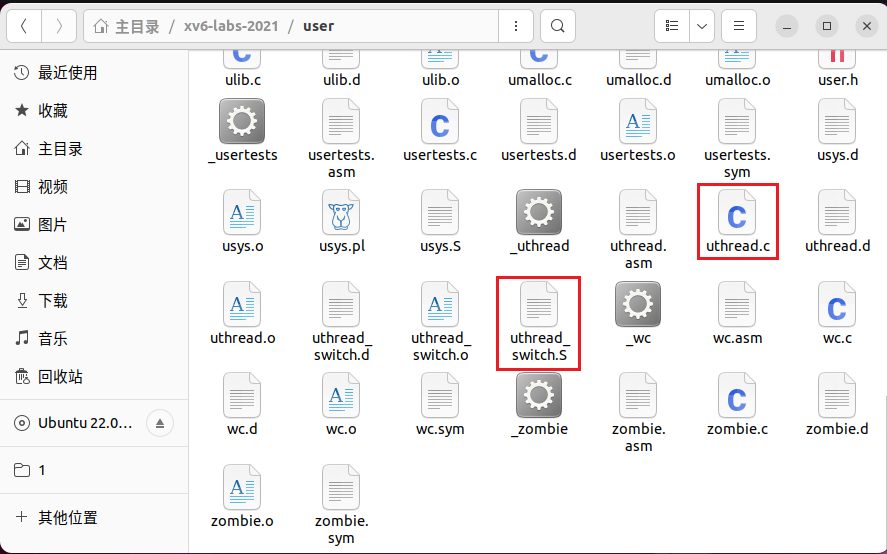
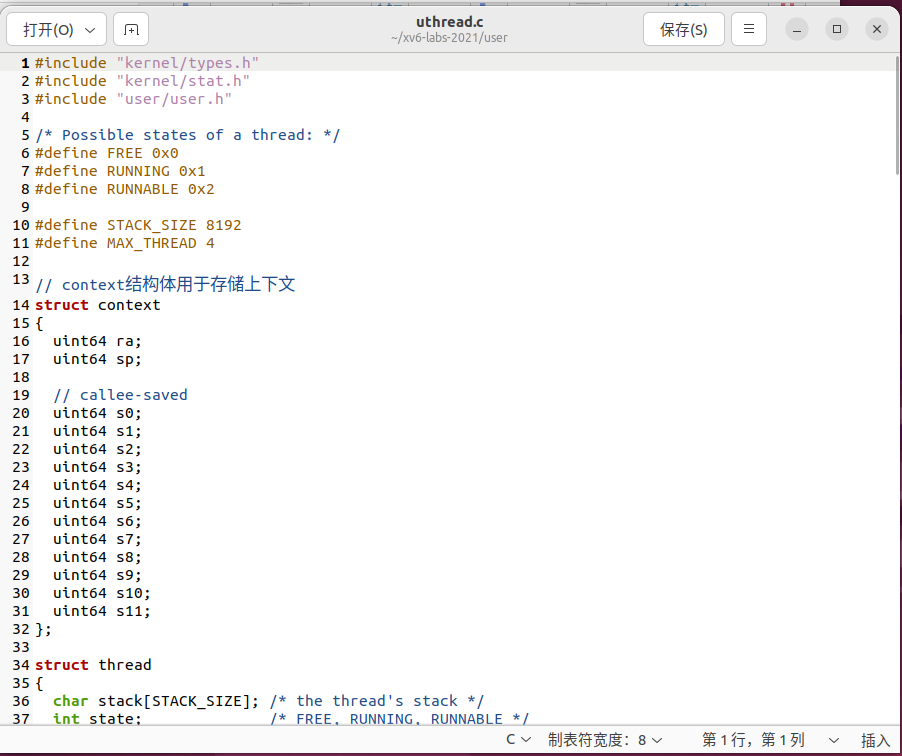
# 1 Uthread: switching between threads

## 1.1实验内容

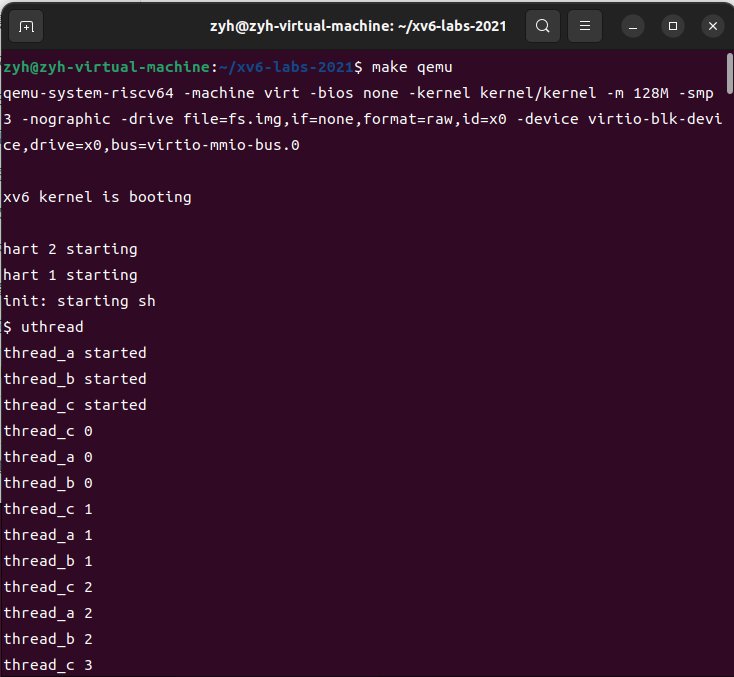
在本练习中，我们要参考xv6的切换上下文的代码，在用户空间中模拟上下文切换的机制。我们的部分代码需要写在user/uthread.c文件里，其内已有模拟线程的代码框架，根据提示，我们要做的就是修改其中的thread\_create函数和thread\_schedule函数，以实现切换线程时的上下问切换。我们还将修改user/uthread\_switch.S里的thread\_switch函数，这是一个汇编代码。简单来说，我们需要完成创建线程、存储/恢复寄存器来进行多线程切换的代码。当你做完时，make grade应该说你的方案通过uthread test。编译xv6后，执行uthread也能检验代码是否编写正确。

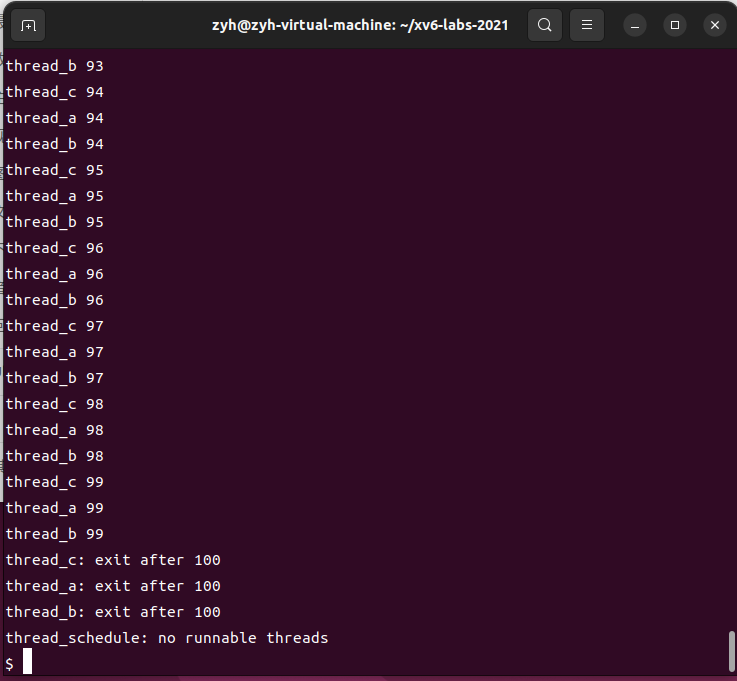
## 1.2代码位置与截图

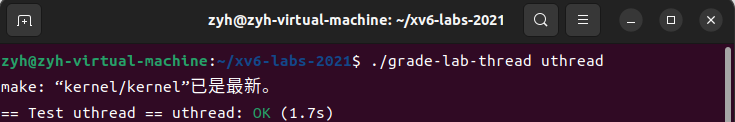




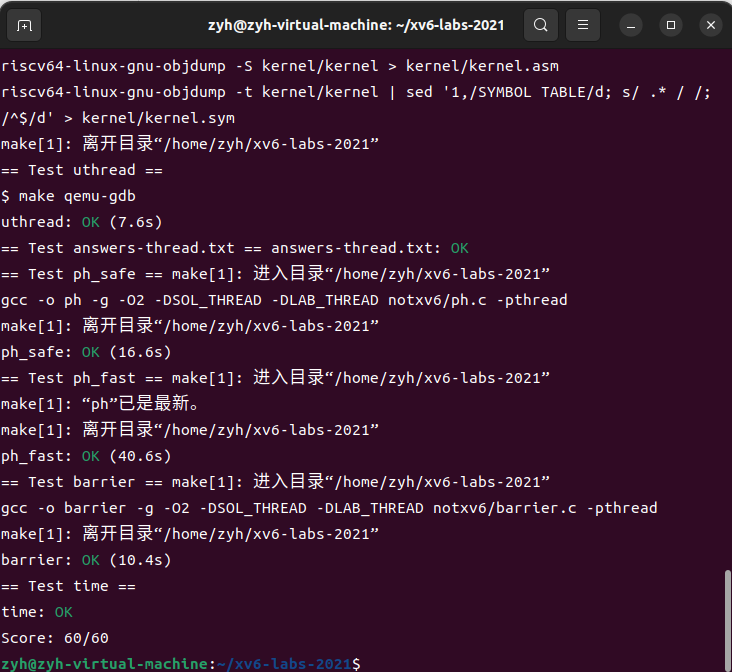
## 1.3实验结果







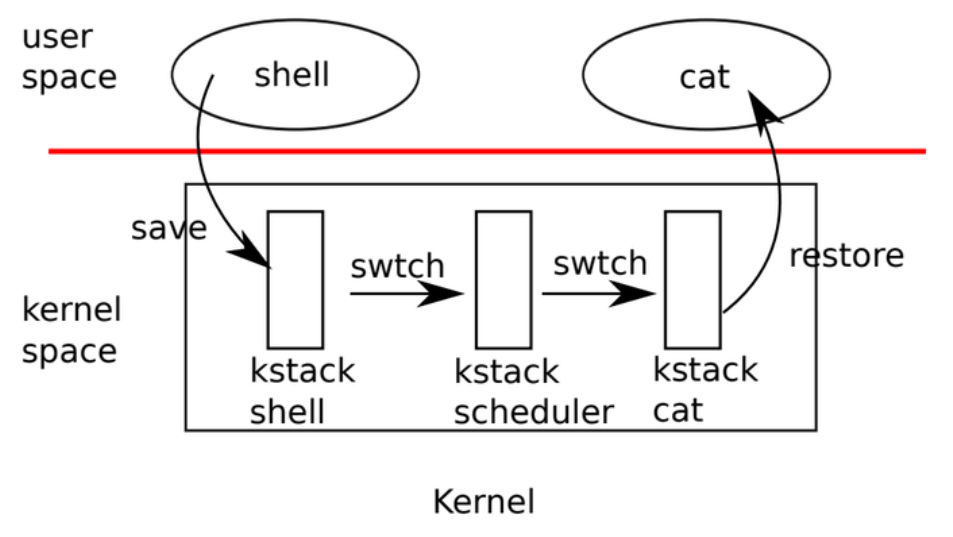
make grade:



## 1.4代码实现

### 1.4.1实现思路

这个实验其实相当于在用户态重新实现一遍 xv6 kernel 中的 scheduler() 和 swtch() 的功能，所以大多数代码都是可以借鉴的。下图是xv6的上下文切换：



这里面涉及到几个重要的函数，在本实验中会用到甚至修改它们的仿造函数.

uthread\_switch.S中需要实现上下文切换的代码，这里借鉴swtch.S，在thread\_switch函数里的/\* YOUR CODE HERE \*/下添加下面的代码：

/\* 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)

这里是参考了xv6的kernel/swtch.S文件的代码，将切换上下文中涉及到的寄存器的数据先保存到了a0为首地址的一个数据结构中（模仿xv6，后面我们定义了一个context结构体用来保存这些寄存器的内容），然后将a1为首地址的一个数据结构的数据加载到这些寄存器中。

线程结构体需要添加一个属性，就是我们定义的context结构体。同时，uthread.c文件里的thread\_switch函数的声明也要修改，它的两个参数要改为context结构体。

我们还需要修改thread\_create函数和thread\_schedule函数，但只需要添加一些代码即可。对于线程调度函数thread\_schedule，我们只需要在/\* YOUR CODE HERE \*/下加上一行代码：

thread\_switch(&t->ctx, &next\_thread->ctx);

表示切换上下文。对于线程创建函数thread\_create，我们要添加两行代码：

t->ctx.ra = (uint64)func;

t->ctx.sp = (uint64)&t->stack + (STACK\_SIZE - 1);

创建该进程是为了执行func函数，所以需要将ra执行func函数的代码，因为swtch函数末尾ret执行后就是往下执行ra指向的代码。另外，一个进程创建后会有自己的栈，这里让线程的栈指针sp指向了自己的栈的最高地址（因为栈是由高地址向低地址生长的）。

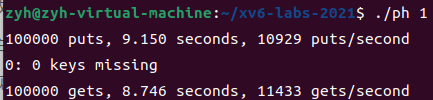
于是，该实验就完成了。

# 2 Using threads

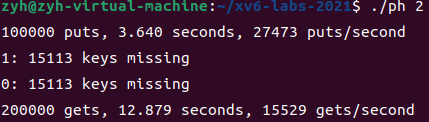
## 2.1实验内容

该实验是独立于xv6操作系统的，因此代码的执行页不需要先make qemu编译xv6。

在进行该实验前，我们需要使用make ph命令构建 ph 程序（修改ph.c的代码后，也需要make ph才会执行新的代码），该程序包含一个线程不安全的哈希表。运行 ./ph 1 即使用单线程运行该哈希表，输出如下：



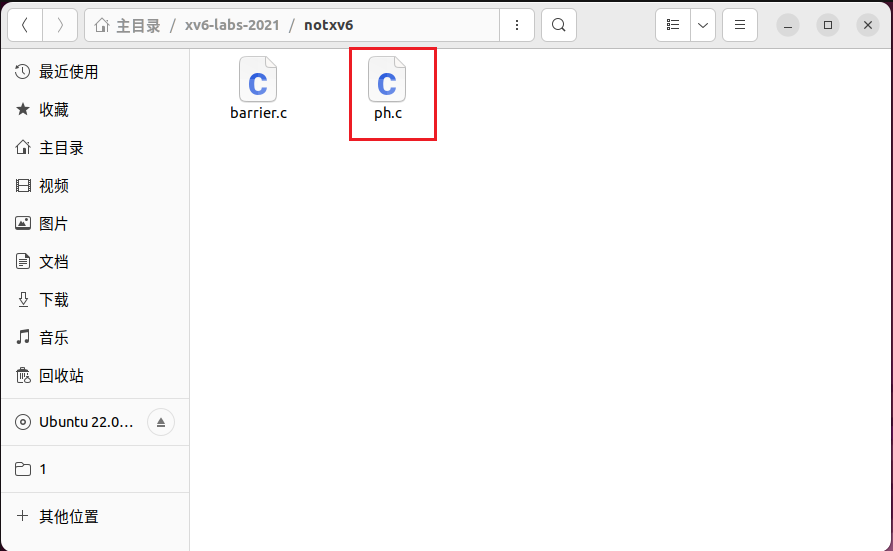
运行 ./ph 2 即使用单线程运行该哈希表，输出如下：



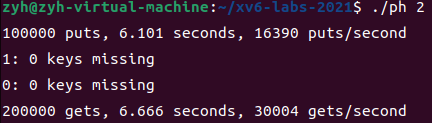
可以看到其 put 速度近乎先前 2 倍, 但是有 15113 个键丢失, 也说明了该哈希表非线程安全。实际上，该程序未采用任何机制来保护向哈希表中并行插入新值的过程。具体来说，如果两个线程正好同时执行put函数，并且对哈希表同一个索引处插入新值，又正好同时执行到insert函数处，它们要插值得位置相同，那么先执行insert函数插入的新值，就会被即后执行insert函数插入的值给覆盖掉。上述问题称为race condition，当两个线程交错执行非原子化操作时，就容易发生结果的丢失。

本实验需要我们通过锁机制来解决上述问题，并且尽可能保留程序执行的效率。

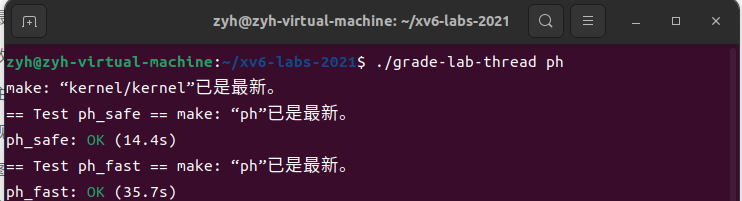
## 2.2代码位置与截图



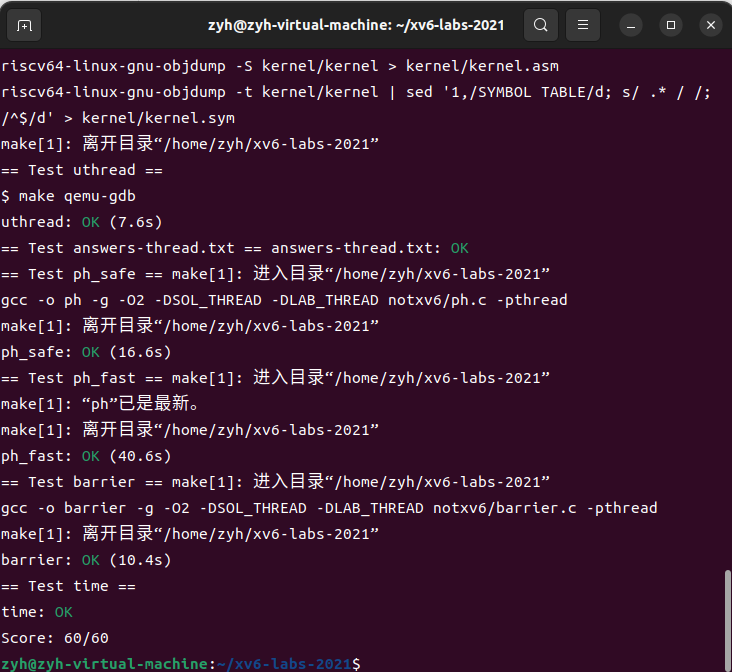
## 2.3实验结果



与完善前的程序相比，put的速度慢了



make grade:



## 2.4代码实现

### 2.4.1实现思路

我们需要给put函数上锁。最简单的，可以定义一个锁：

pthread\_mutex\_t lock;

再在main函数里初始化化后，直接在put函数的首尾加上锁：

static void put(int key, int value)

{

pthread\_mutex\_unlock(&lock);

……

pthread\_mutex\_unlock(&lock);

}

但是本实验的测试有一项ph\_fast，判断测试程序的执行是否过长。如果采用上述简单粗暴的方法，那么执行时间必然过程，导致无法通过测试，所以我们需要再挖掘一下put函数的并行性。可以发现，不同bucket的insert是可以并行执行的，于是我们需要多个锁，执行一次put函数只需要上特定的锁。于是：

pthread\_mutex\_t locks[NBUCKET];

这里NBUCKET表示哈希表的bucket的个数，我们给每个bucket定义独属于自己的lock。相应的，put函数里的代码改为：

static void put(int key, int value)

{

int i = key % NBUCKET;

pthread\_mutex\_unlock(&locks[i]);

……

pthread\_mutex\_unlock(&locks[i]);

}

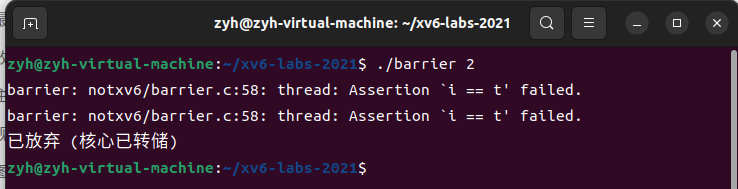
获取i的代码是原来put函数就带有的，用于表示键对应的bucket。

# 3 Barrier

## 2.1实验内容

该实验是独立于xv6操作系统的，因此代码的执行页不需要先make qemu编译xv6。

在进行该实验前，我们需要使用make barrier命令构建 barrier 程序（修改barrier.c的代码后，也需要make barrier才会执行新的代码）。此时会有如下报错：

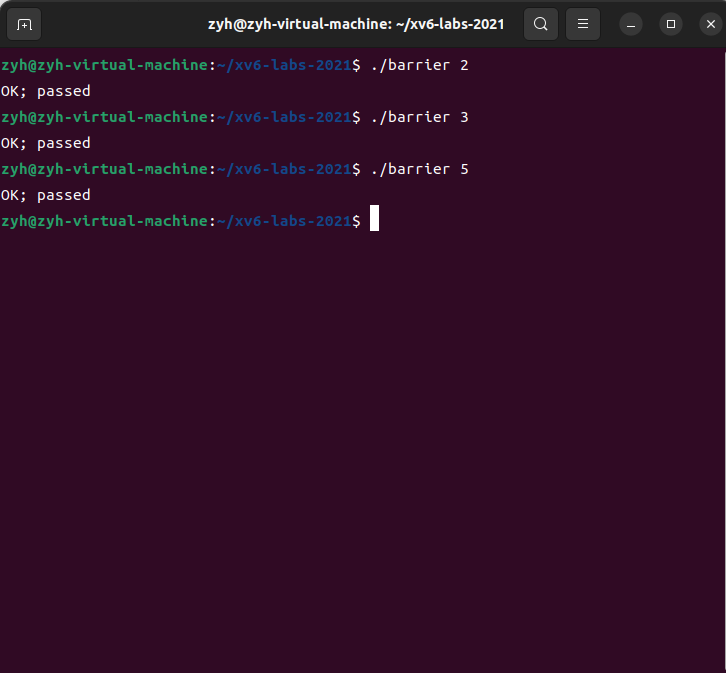


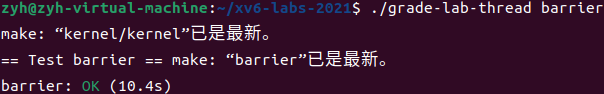
本次实验，需要我们完善barrier.c程序中的barrier函数，实现该功能：个别线程执行到barrier函数时进入等待，所有线程都指向到barrier时，才唤醒所有线程。

## 2.2代码位置与截图

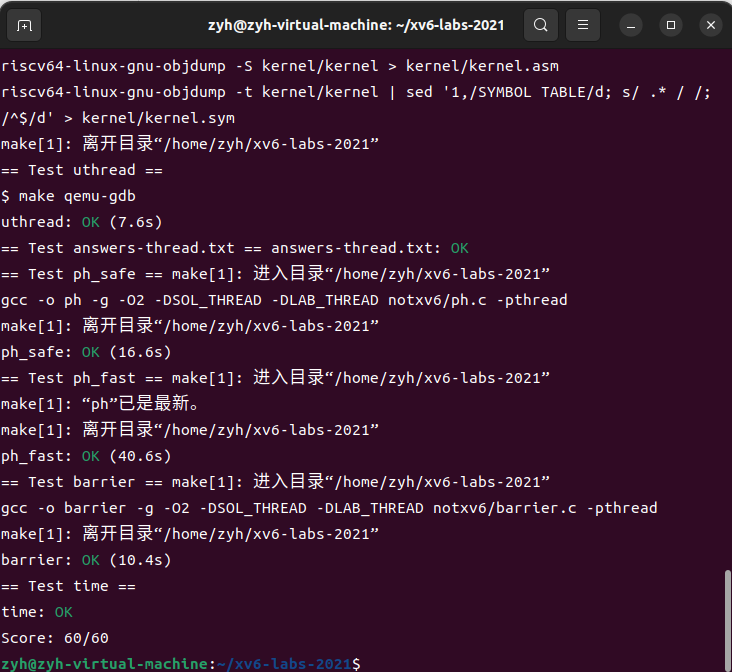


## 2.3实验结果





make grade:



## 2.4代码实现

### 2.4.1实现思路

为完成本次实验，我们需要利用好文件里定义的barrier结构体：

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;

其中nthread属性用于记录本次round（所有线程执行一次barrier函数称为一个round）中有几个线程到达了barrier函数。round属性则记录程序执行了几个round。

pthread\_mutex\_t表示互斥量，pthread\_cond\_t表示共享变量，它们是实现本次实验的关键。pthread\_mutex\_t不方便实现线程的有条件的阻塞，而共享变量使用pthread\_cond\_wait实现起来很简单。另外，函数pthread\_cond\_wait必须与pthread\_mutex\_t配套使用。因为线程通过pthread\_cond\_wait进入阻塞状态的同时，会主动调用pthread\_mutex\_unlock释放掉互斥量。而当线程通过pthread\_cond\_broadcast被唤醒时，该线程又会主动调用 pthread\_mutex\_lock来上锁。

那么我们实现思路就很简单了：barrier函数里，先通过互斥量上锁，然后通过nthread判断所有线程是否都执行到了barrier函数，若没有，则阻塞进程；反之，唤醒所有线程。

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) // 每有一个线程到达bstate.nthread便加一，并判断是否所有线程都到达

{

pthread\_cond\_wait(&bstate.barrier\_cond, &bstate.barrier\_mutex); // 先到的线程等待

}

else // 所有线程到达

{

bstate.nthread = 0; // 重置bstate.nthread

++bstate.round; // 所有进程执行过一次barrier称为一个round

pthread\_cond\_broadcast(&bstate.barrier\_cond); // 唤醒所有线程

}

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

}