Lab5

实验报告

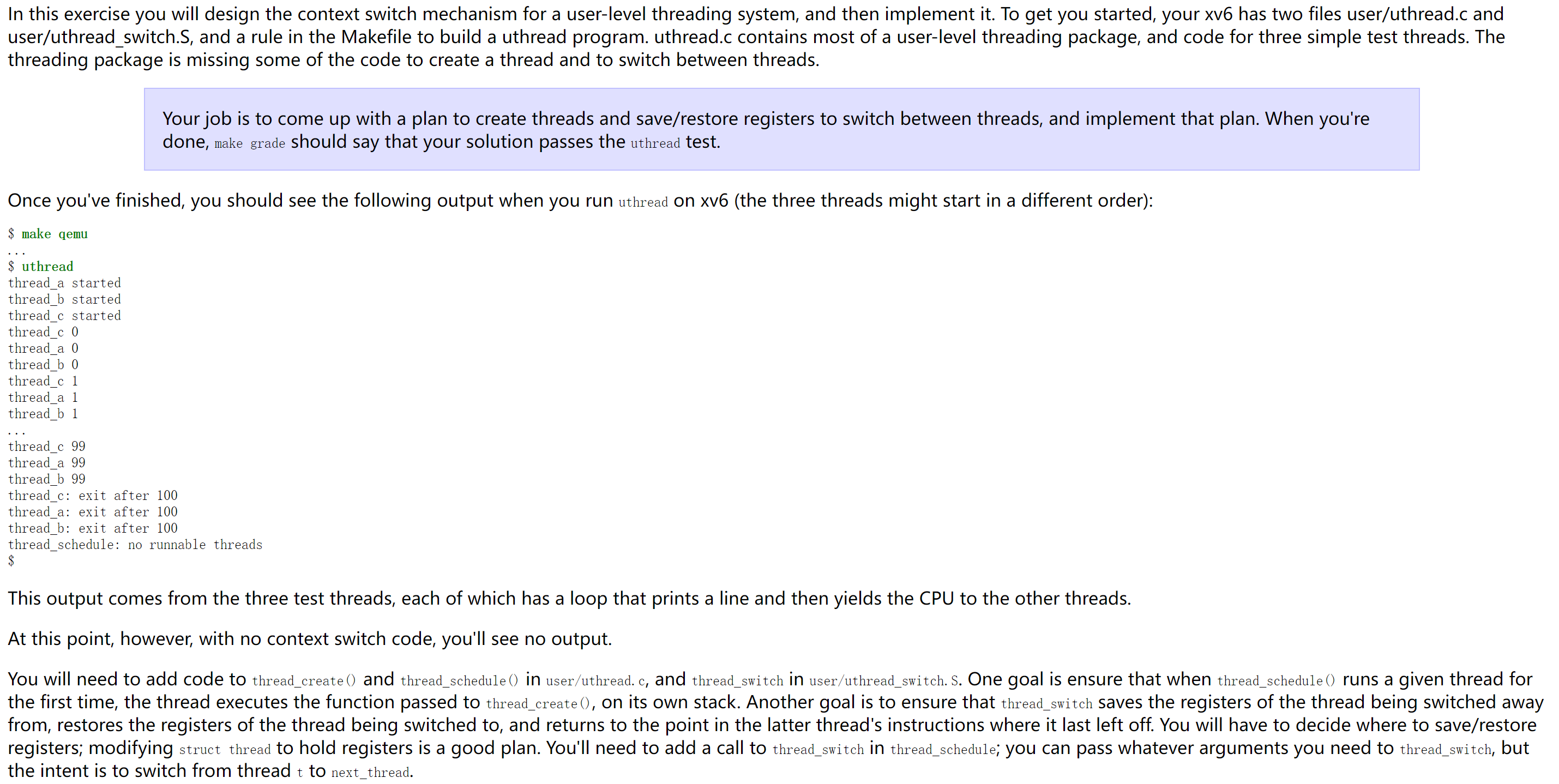
姓名 李晓畅

学号 20307130261

班级 计算机科学技术

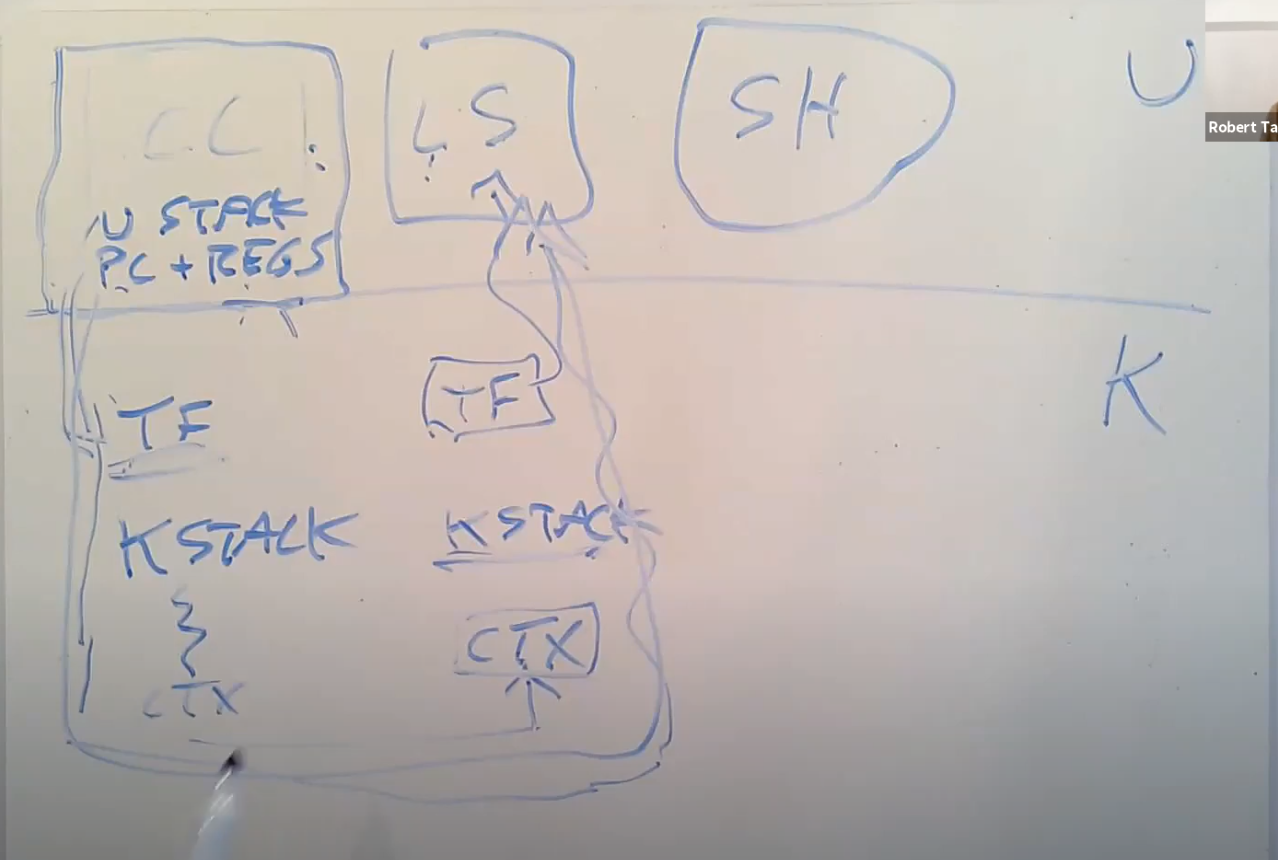
# 实现思路和问题回答

## 3.1 Uthread: switching between threads



其实最关键的一点是要了解线程切换的过程到底是什么样的。

概念上说大概是：



分为下面几个步骤：

1. XV6会首先会将CC程序的内核线程的内核寄存器保存在一个context对象中。

2.类似的，因为要切换到LS程序的内核线程，那么LS程序现在的状态必然是RUNABLE，表明LS程序之前运行了一半。这同时也意味着LS程序的用户空间状态已经保存在了对应的trapframe中，更重要的是，LS程序的内核线程对应的内核寄存器也已经保存在对应的context对象中。所以接下来，XV6会恢复LS程序的内核线程的context对象，也就是恢复内核线程的寄存器。

3.之后LS会继续在它的内核线程栈上，完成它的中断处理程序（注，假设之前LS程序也是通过定时器中断触发的pre-emptive scheduling进入的内核）。

4.然后通过恢复LS程序的trapframe中的用户进程状态，返回到用户空间的LS程序中。

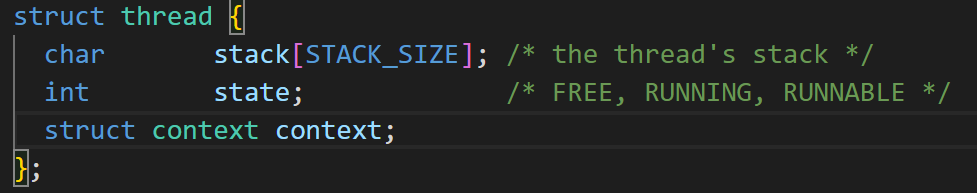
5.最后恢复执行LS

下面可以按照提示进行：

* 

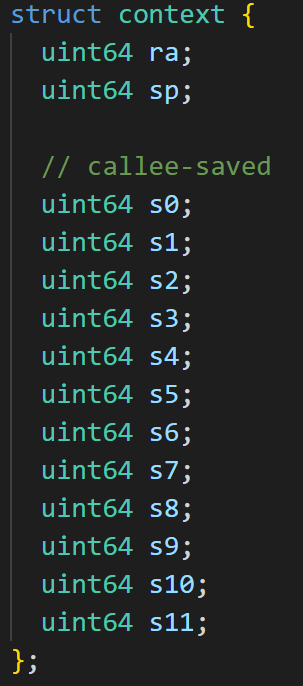
参照上面的流程，Trapframe已经实现了，那么我们在用户态下就拷贝一个寄存器状态就可以了qwq。

个人决定还是把context包含在thread里比较好，包含在trapframe里其实应该也可以，但这样的话我们还要修改对应的代码。



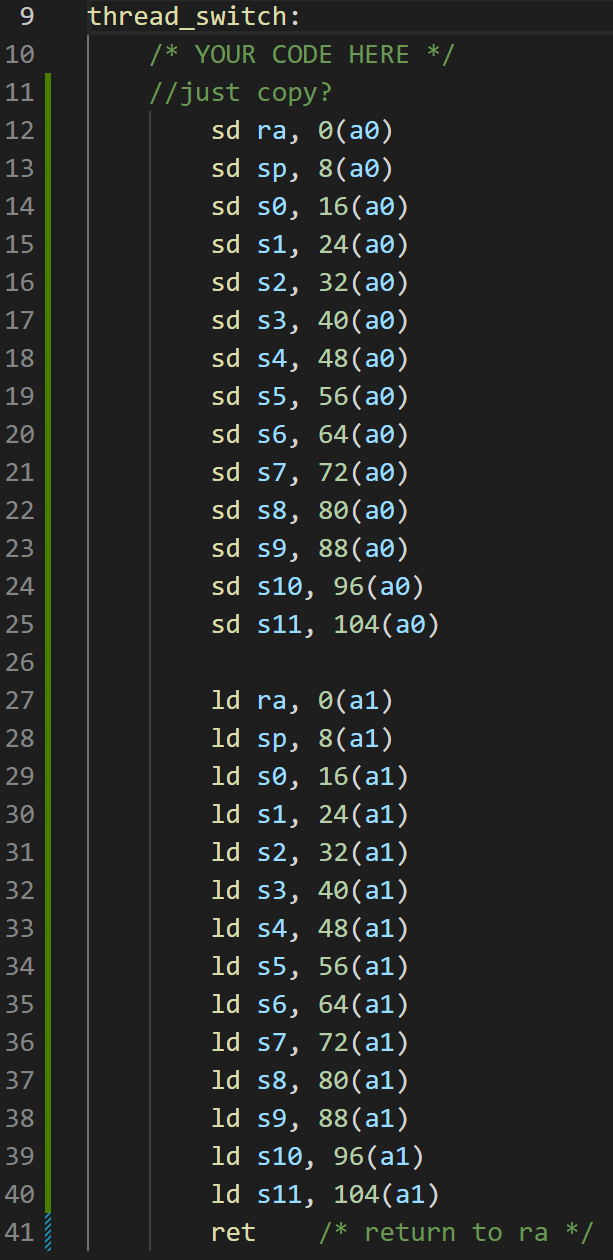
添加一个context。虽然其实这个context和proc.h里包含的一样，实际上也确实只需要这些东西就可以了，不过最好还是不要,

因为还要一些东西才能编译，而且逻辑上关系没有那么大。抄一个过来就可以。



* 

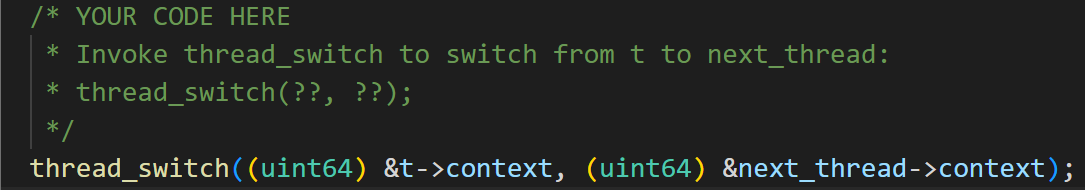
切换context就照抄swtch就可以了。



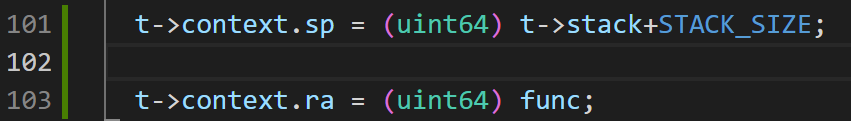
这里注意一下它给的是一个基地址+变址的形式，所以我们应该传入两个地址作为参数，不然就要改代码。

* 具体实现就不难了。

在thread\_schedule里简单调用一下。不用应该也可以，但是直接用内核的swtch肯定会增加切换的成本，实际上也没必要非得进入内核来切换context。



另外在创建的时候添加对context的初始化。



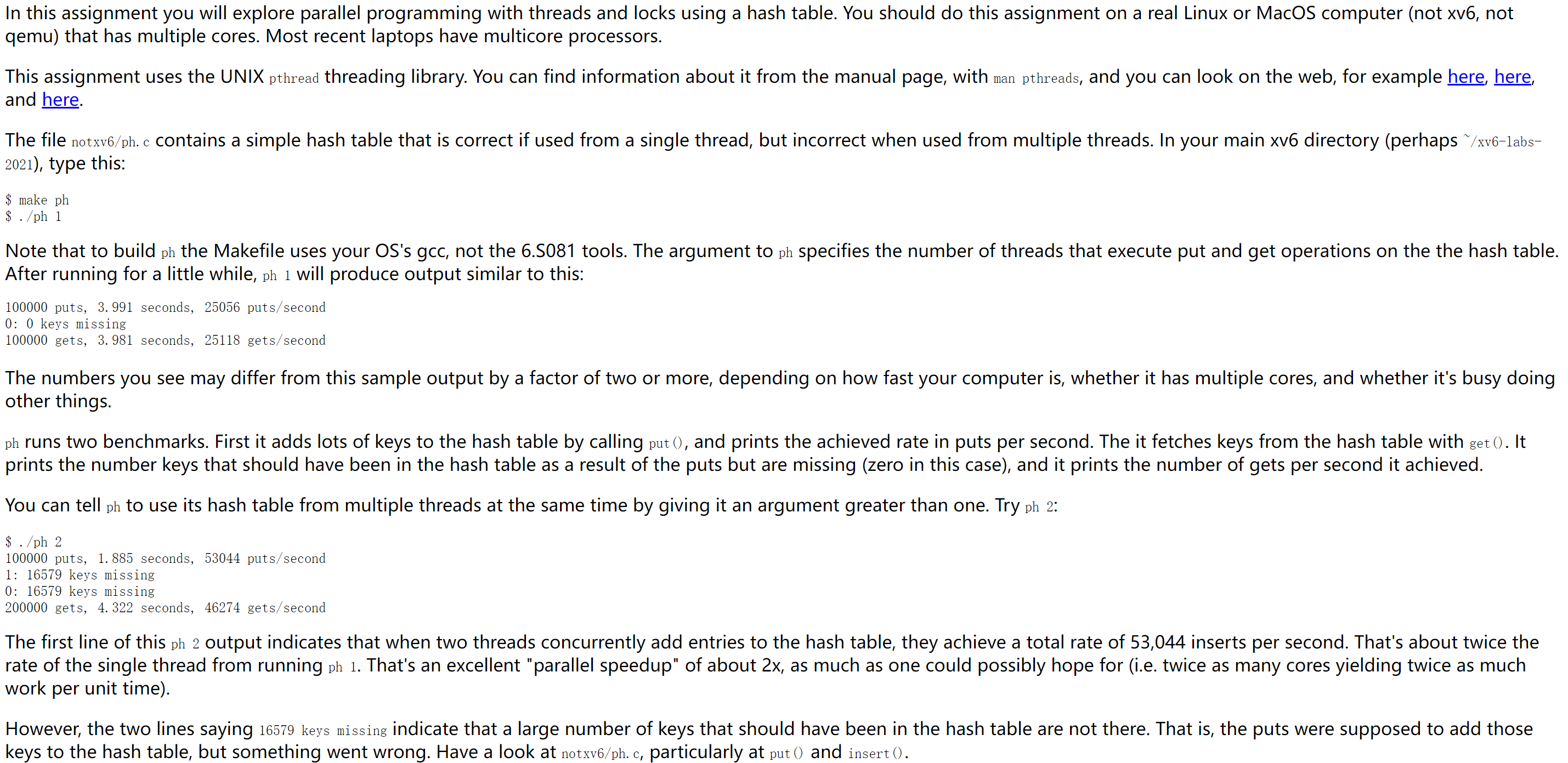
大概就可以了。

测试结果截图：

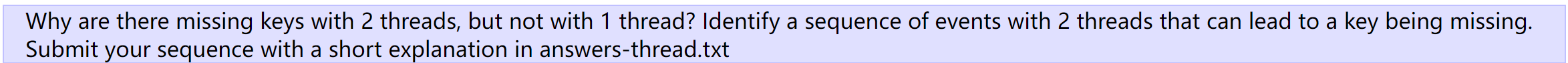


运行成功

## 3.2 Using threads



我们一个个解决问题：

* 

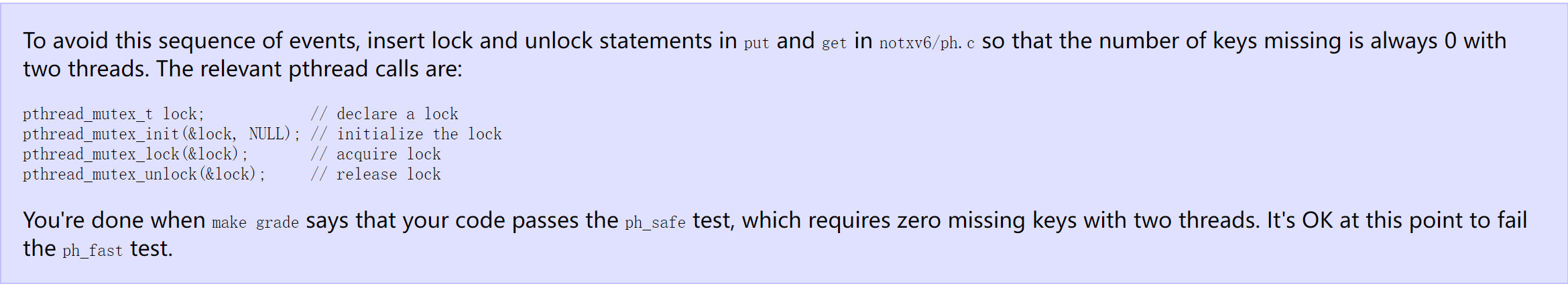
因为临界资源，即对应的桶，没有被保护，导致多线程情况下可能会对同一个桶的同一个链表表元并发写，导致后插入的元素实际上覆盖了之前插入的元素，导致丢失。而在单线程条件下，一切都是顺序进行的，不会有并发问题。

具体说，两个进程可能将两个数差不多同时hash到同一个桶，并且几乎同时执行到这一步，不论是由于他们实际上在不同CPU上运行而实际上同时运行到这里，还是因为他们的时间片耗尽，导致有一个线程被停止都可能造成这样的情况。这时，他们传入insert的都是同一个地址，也就是最后一个表元，对最后一个表元实际上重复添加了两次，前一次被后一次覆盖了。

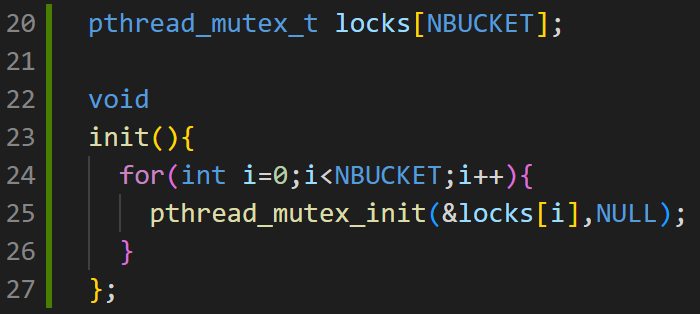
类似的问题还很多，比如重复插入也是有可能的，但概率应该要小很多。所以最终呈现的效果是丢失了不少。

* 

一次只允许一个线程访问临界区就好了。互斥变量为我们提供了这样的方法。



把hash表作为临界资源直接上锁是最直接简单的方法。不过这样做性能堪忧，归根结底我们还是只能一个一个地对hash表写。提示说得很清楚了，可以考虑给每个桶上一个锁。

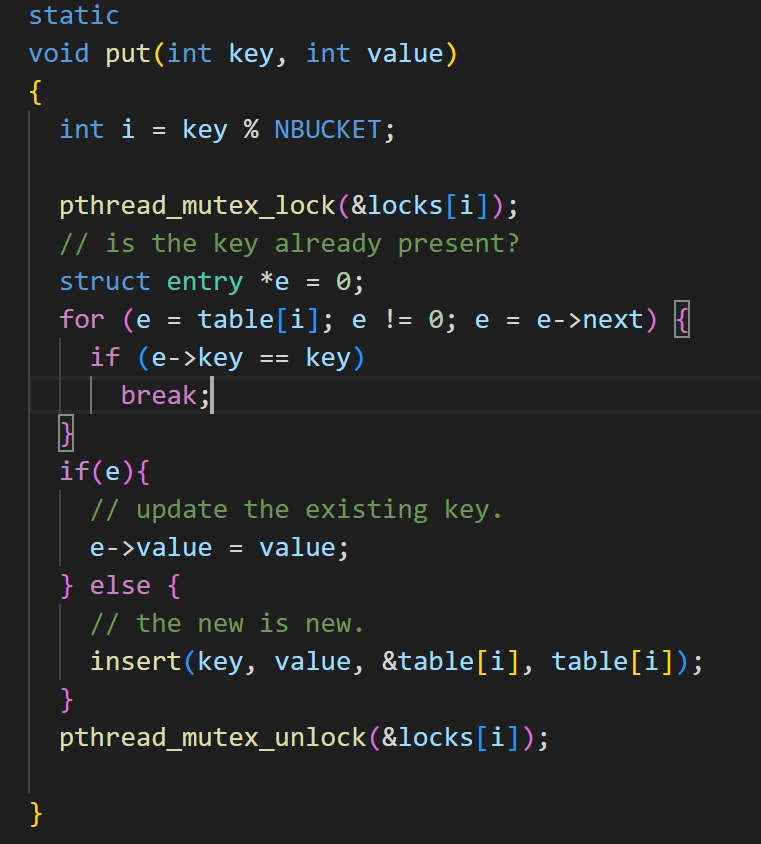


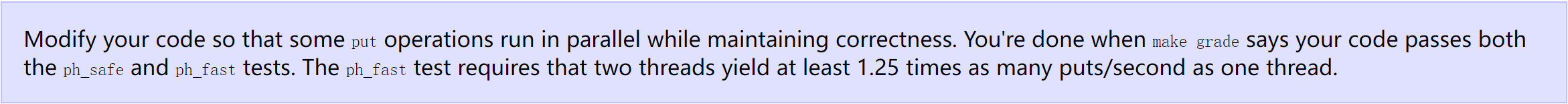
所以要为每个桶创建锁，并初始化。



在进行读写时要先获得锁，简单这样实现就可以实现功能了，而且性能受到的影响也少不少。下面的代码就是这样实现的。

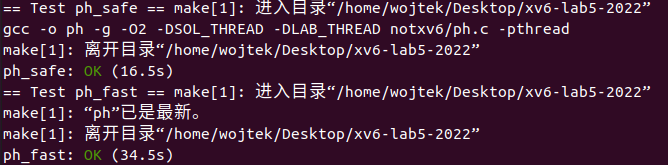
实际上或许我们可以把读和写分开？这样设一把读者写者锁，只是检查hash表的是读者，要修改的是写者，这样允许很多读者同时检查hash表，应该还可以提升性能。





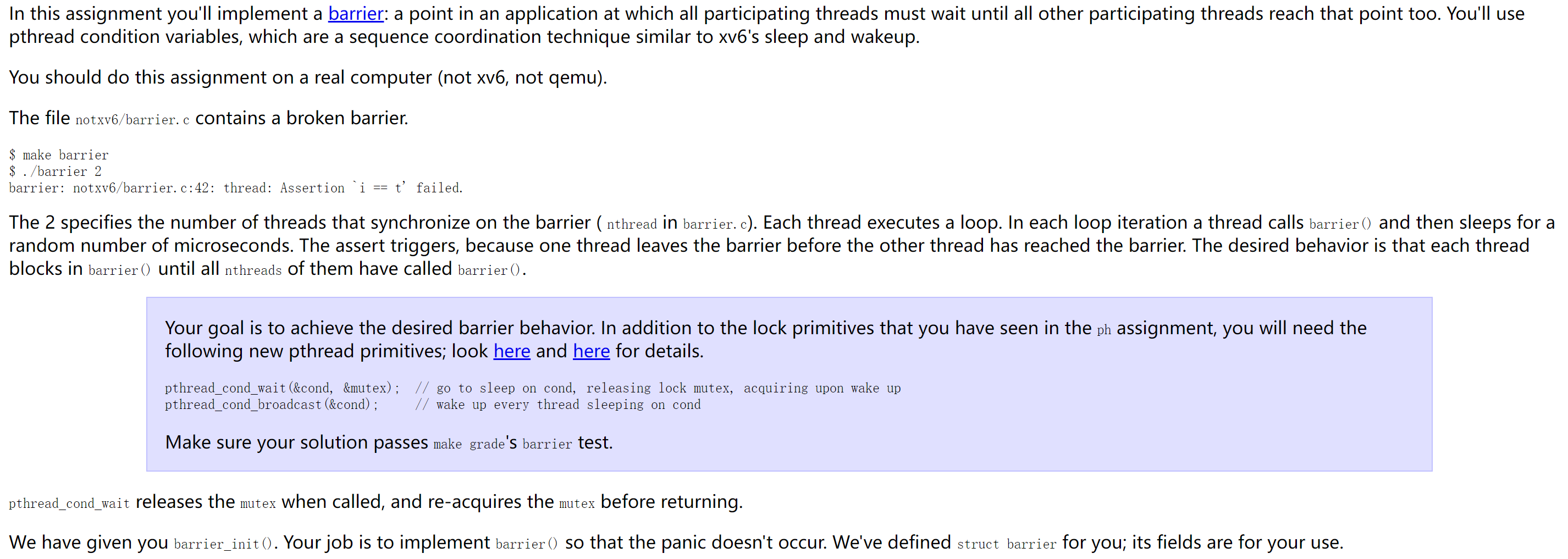
不过其实这样的性能也足够通过测试了qwq。

测试结果截图：



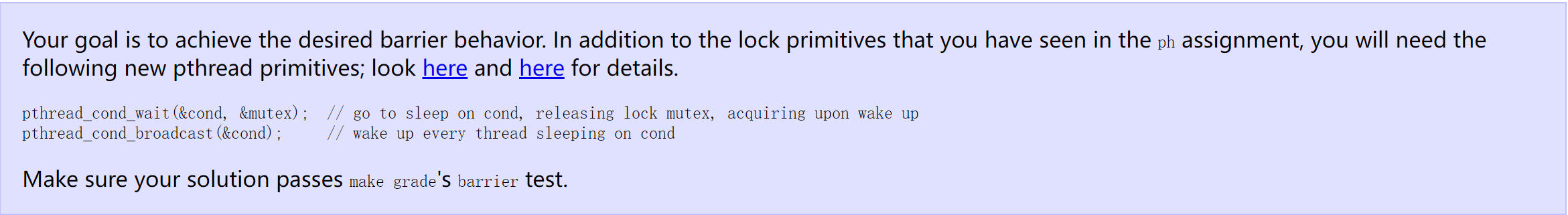
运行成功

## 3.2 Using threads



前面提到的互斥锁有一个明显到缺点: 只有两种状态，锁定和非锁定。

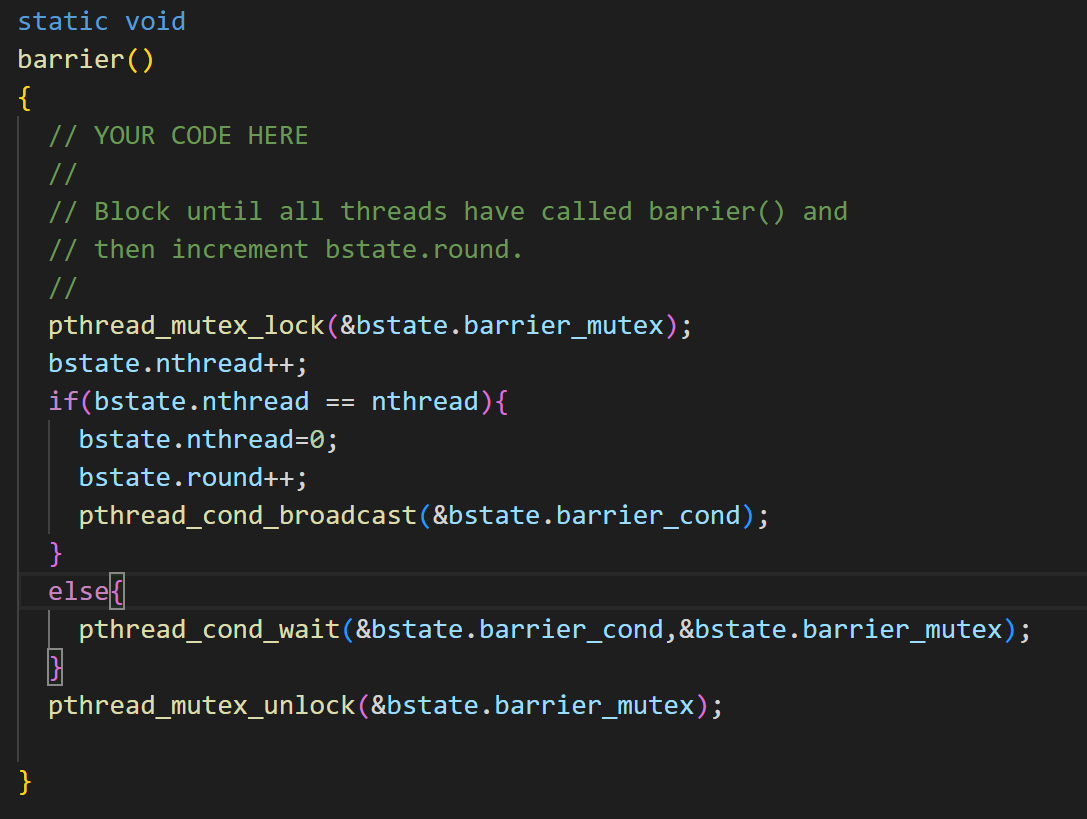
而条件变量则通过允许线程阻塞并等待另一个线程发送唤醒信号的方法弥补了互斥锁的不足，它常和互斥锁一起使用。



再注意考察两个要求：



我们就不难实现了：



遇到路障的线程要更改路障的信息，因而要先获得路障的锁。如果阻塞在路障上的线程足够多，就把他们全部唤醒放行，这时轮数增加，被阻塞线程数清零。不够多线程就必须被阻塞。完成后必须解锁路障信息。

测试结果：

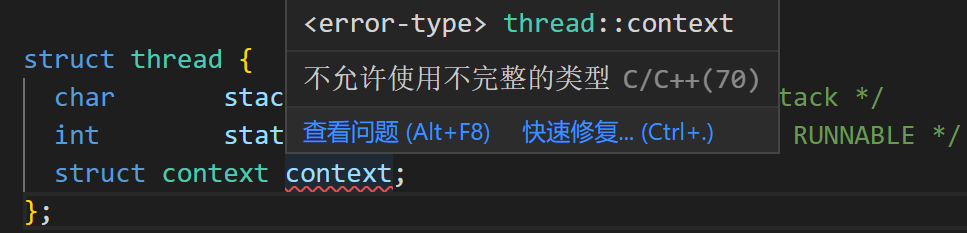


成功。

# 问题和解决

## 如何实现context

在第一部分加入context后就有这样的错：



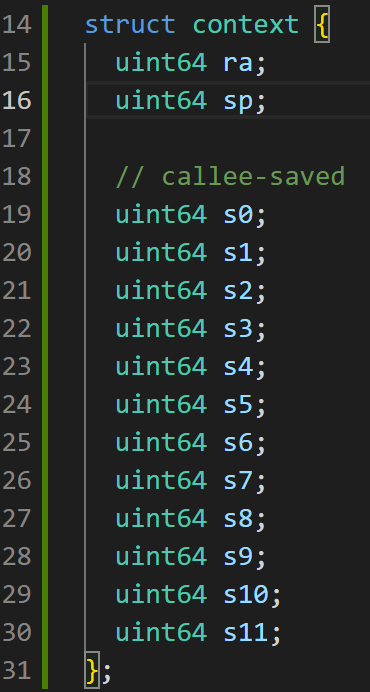
那是因为编译器并不知道我们说的context是什么。

如果直接加入

或许并不是一个好选择，虽然我们只需要context,但C可不允许我只import我想要的东西。这就意味着如果全部加进去的话就有更多相关的部分需要被加进去，即使这些部分和我们的实现一点关系都没有。

所以直接抄一份或许是更好的选择。

然后就可以了。



另外，因为xv6一个进程对应的是一个线程，所以其实就用trapframe里的那个应该也行。但问题在于这就要改其他（很多）地方的代码，而且逻辑上进程和线程也不是同样的东西，所以最好不要这样做。这应该也不是作者希望得到的解决方法。

## 为什么要有线程？

Xv6系统中一个进程只包含一个进程，线程调度实际上还得设计进程调度，在此基础上我们实际上还额外进行了一层抽象，要保存恢复更多信息，即便可能是重复的。那这样的线程有意义吗？

回顾线程的意义：

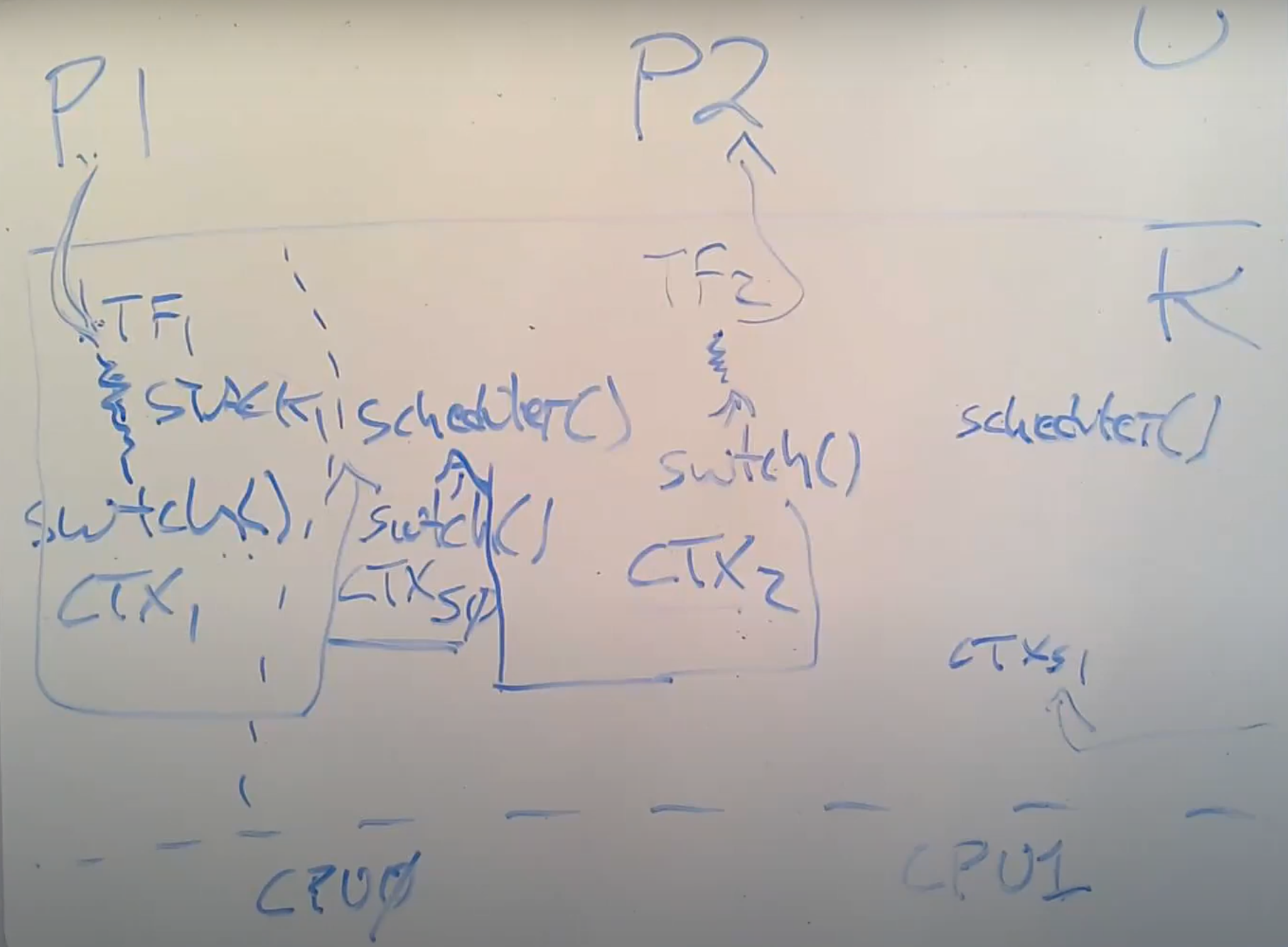
* 首先，人们希望他们的计算机在同一时间不是只执行一个任务。有可能计算机需要执行分时复用的任务。甚至在一个单用户的计算机或你会运行多个进程，并期望计算机完成所有的任务而不仅仅只是一个任务。
* 其次，多线程可以让程序的结构变得简单。线程在有些场合可以帮助程序员将代码以简单优雅的方式进行组织，并减少复杂度。
* 最后，使用多线程可以通过并行运算，在拥有多核CPU的计算机上获得更快的处理速度。常见的方式是将程序进行拆分，并通过线程在不同的CPU核上运行程序的不同部分。你可以认为XV6就是一个多CPU并行运算的程序。

从中我们至少我们可以确定一点，线程用起来比进程方便得多。实际上我们用起来大概也是这样的，创建线程更方便。也就是所谓“简单优雅”。

# 实验总结

本次实验中我们学习了xv6的进程调度。

具体过程大概是这样的：



考虑时间片用尽了被迫切换的线程会经历什么：

* 首先一个定时器中断强迫CPU从用户空间进程切换到内核，trampoline代码将用户寄存器保存于用户进程对应的trapframe对象中；
* 之后在内核中运行usertrap，来实际执行相应的中断处理程序。这时，CPU正在进程P1的内核线程和内核栈上，执行内核中普通的C代码；
* 假设进程P1对应的内核线程决定它想出让CPU，它会做很多工作，这个我们稍后会看，但是最后它会调用swtch函数（译注：switch 是C 语言关键字，因此这个函数命名为swtch 来避免冲突），这是整个线程切换的核心函数之一；
* swtch函数会保存用户进程P1对应内核线程的寄存器至context对象。所以目前为止有两类寄存器：用户寄存器存在trapframe中，内核线程的寄存器存在context中。
* 在schedulder函数中会做一些清理工作，例如将进程P1设置成RUNABLE状态。之后再通过进程表单找到下一个RUNABLE进程。假设找到的下一个进程是P2（虽然也有可能找到的还是P1），schedulder函数会再次调用swtch函数，完成下面步骤：
* 先保存自己的寄存器到调度器线程的context对象
* 找到进程P2之前保存的context，恢复其中的寄存器
* 因为进程P2在进入RUNABLE状态之前，如刚刚介绍的进程P1一 样，必然也调用了swtch函数。所以之前的swtch函数会被恢复， 并返回到进程P2所在的系统调用或者中断处理程序中
* 不论是系统调用也好中断处理程序也好，在从用户空间进入到内核空 间时会保存用户寄存器到trapframe对象。所以当内核程序执行完成 之后，trapframe中的用户寄存器会被恢复
* 最后用户进程P2就恢复运行了

。