@[TOC](Mit6.S081-实验7-Multithreading)

# 一、实验准备

本实验将带你熟悉多线程。

你将实现线程（用户级线程包）切换，使用多线程加速程序，实现一个barrier。

在写代码前，你应该确定你已经读过xv6 book的章节7：scheduling，并且读过相关代码。

开始本lab前，先切换到cow分支

git fetch、git checkout cow、make clean

# 二、Uthread：switching between threads

## 1，实验要求

在本练习中，你将为用户级线程系统设计context切换机制，然后实现它。

开始前，xv6有两个文件user/uthread.c和user/uthread\_switch.S，和一个在Makefile中的规则，来构建一个uthread程序。

uthread.c包含绝大多数用户级线程包，以及3个简单的测试线程代码。

线程包缺少一些代码来创建一个线程和在线程之间切换。

你的工作是提出一个计划：创建线程、存储/恢复寄存器来进行多线程切换，然后实现该计划。

当你做完时，make grade应该说你的方案通过uthread test。

一旦你已经结束了，当你在xv6上执行uthread时，你应该看到下面输出（3个线程可能以不同顺序启动）：

![在这里插入图片描述](https://img-blog.csdnimg.cn/20210304111108547.png)

这个输出来自3个测试线程，每个测试线程有一个循环（打印一行然后让出CPU到其他线程）。

基于这点，如果没有context切换代码，你将看不到输出。

你将需要添加代码到user/uthread.c中的thread\_creat()和thread\_schedule()，user/uthread\_switch.S的thread\_switch。

目标一是确保当thread\_schedule()首次运行一个给定线程时，线程执行传到thread\_create()中的函数，在它自己的栈上。

另外目标是确保thread\_switch保存切换前线程的寄存器，恢复要切换线程的寄存器，返回到切换后的线程上次离开时的指令。

你将不得不决定哪里存储/恢复寄存器；更改struct thread来保存寄存器是一个好计划。

你将需要在thread\_schedule添加thread\_switch调用；你能传递任何需要的参数到thread\_switch，但目的是从一个线程切换到下个线程。

## 2，一些提示

thread\_switch仅需要存储/恢复callee-save寄存器。为什么？

你可以在user/uthread.asm中看到uthread汇编代码，这可能对调试有帮助。

对于测试你的代码，使用riscv64-linux-gnu-gdb单步thread\_switch可能是有帮助的。

(gdb) file user/\_uthread

Reading symbols from user/\_uthread...

(gdb) b uthread.c:60

这会在uthread.c第60行设置一个断点。

这个断点可能会（也可能不会）在你运行uthread之前触发。这如何发生？

一旦你的xv6 shell运行，输入”uthread”，gdb将在60行停住。现在你可以如下输入命令来检测uthread的状态：

(gdb) p/x \*next\_thread

用”x”，你可以检测内存位置的内容：

(gdb) x/x next\_thread->stack

你可以跳到thread\_switch开始处：

(gdb) b thread\_switch

(gdb) c

你可以使用si来单步汇编指令：

(gdb) si

## 3，具体实现

1）修改user/uthread.c，新增头文件引用。

![在这里插入图片描述](https://img-blog.csdnimg.cn/20210304112908509.png)

2）修改user/uthread.c，struct thread增加成员struct context，用于线程切换时保存/恢复寄存器信息。此处struct context即kernel/proc.h中定义的struct context。

![在这里插入图片描述](https://img-blog.csdnimg.cn/20210304113221534.png)

3）修改user/uthread.c，修改thread\_switch函数定义。

![在这里插入图片描述](https://img-blog.csdnimg.cn/20210304113428923.png)

4）修改user/uthread.c，修改thread\_create函数，修改ra保证初次切换到线程时从何处执行，修改sp保证栈指针位于栈顶（地址最高）。

![在这里插入图片描述](https://img-blog.csdnimg.cn/20210304113605286.png?x-oss-process=image/watermark,type\_ZmFuZ3poZW5naGVpdGk,shadow\_10,text\_aHR0cHM6Ly9ibG9nLmNzZG4ubmV0L3UwMTM1Nzc5OTY=,size\_16,color\_FFFFFF,t\_70)

5）修改user/uthread.c，修改thread\_schedule，调用thread\_switch调用，保存当前线程上下文，恢复新线程上下文。

![在这里插入图片描述](https://img-blog.csdnimg.cn/20210304114021937.png?x-oss-process=image/watermark,type\_ZmFuZ3poZW5naGVpdGk,shadow\_10,text\_aHR0cHM6Ly9ibG9nLmNzZG4ubmV0L3UwMTM1Nzc5OTY=,size\_16,color\_FFFFFF,t\_70)

6）修改user/uthread\_switch.S，实现保存/恢复寄存器。

![在这里插入图片描述](https://img-blog.csdnimg.cn/20210304114421838.png)

## 4，执行效果

![在这里插入图片描述](https://img-blog.csdnimg.cn/20210304114645438.png)

![在这里插入图片描述](https://img-blog.csdnimg.cn/20210304114713249.png)

# 三、Using threads

## 1，实验要求

在本次作业中，你将使用哈希表探索并行程序（带有线程和锁）。

你应该在一个真正的Linux或MacOS多核计算机上做此次作业（并非xv6、并非qemu）。

现在的笔记本绝大多数拥有多处理器。

本次作业使用Unix pthread threading库。

你可以从说明页找到关于它的信息，with man pthreads。

你可以在web上看到三个例子：[例一](https://pubs.opengroup.org/onlinepubs/007908799/xsh/pthread\_mutex\_lock.html)、[例二](https://pubs.opengroup.org/onlinepubs/007908799/xsh/pthread\_mutex\_init.html)、[例三](https://pubs.opengroup.org/onlinepubs/007908799/xsh/pthread\_create.html)。

文件notxv6/ph.c包含一个简单的哈希表，如果单线程使用，它是正确的，但多线程使用是不正确的。

在你的xv6主目录，输入：

```bash

$ make ph

$ ./ph 1

```

注意：构建ph，Makefile使用OS的gcc，而不是6.S081工具。

对程序ph的参数明确了，哈希表执行put和get操作的线程数目。

执行一会后，ph1将产生输出，类似：

```bash

100000 puts, 3.991 seconds, 25056 puts/second

0: 0 keys missing

100000 gets, 3.981 seconds, 25118 gets/second

```

你看到的数目可能不同于这个输出，相差2倍或更多，这取决于你的电脑有多快，是否它有多核，是否它正忙于做其他事。

ph执行两个评测。首先它通过调用put()添加大量key到哈希表，打印每秒put的数量。

它用get()从哈希表获取keys。它打印，应该已经因put而存在于哈希表中，但却不在的key数量，打印每秒get数量。

你可以告知ph，同一时间，通过给一个大于1的参数用多线程使用它的哈希表，尝试ph 2：

```bash

$ ./ph 2

100000 puts, 1.885 seconds, 53044 puts/second

1: 16579 keys missing

0: 16579 keys missing

200000 gets, 4.322 seconds, 46274 gets/second

```

ph 2输出的第一行表明：当两个线程并发地添加entry到哈希表，它们实现每秒53044次插入。

这两倍于单线程执行ph 1。这是一个卓越的parallel speedup，2倍于单个（每单元时间，两倍核产生两倍工作）。

然而，两行显示16579 keys missing：表明大量应该在哈希表中的key并不在。

put被期望添加那些key到哈希表，但某些地方出错了。

看notxv6/ph.c，特别是put()和insert()。

为什么两个线程时missing keys，1个线程时没有？说明2个线程的一系列事件（导致missing keys）。

用一个简单的说明（位于answers-thread.txt）提交你的序列。

为了避免一系列事件，插入锁和释放锁语句放在notxv6/ph.c的put和get中，以至于两个线程missing keys的数量总是0。

相关pthread调用是：

```c

pthread\_mutex\_t lock; //定义一个锁

pthread\_mutex\_init(&lock, NULL); //初始化锁

pthread\_mutex\_lock(&lock); //获取锁

pthread\_mutext\_unlock(&lock); //释放锁

```

make grade说你的代码通过ph\_safe测试时，那么可以了，需要两个线程0 missing keys。

在此处没通过ph\_fast测试没关系。

不要忘记调用pthread\_mutex\_init()。

用1个线程测试你的代码，然后用两个线程测试你的代码，正确么（消除了missing keys？）？

与单线程版本相比，两个线程版本实现了并行加速（每单元时间处理更多工作）？

有些情况并发put不会在内存中重叠（读写哈希表时），因此无需锁来保护排斥另外一个。

你可以修改ph.c利用这个情况，来获取put并行加速？提示：每个hash bucket一把锁如何？

更改你的代码以便于一些put操作在保证正确的情况下并行执行。

当make grade说你的代码通过ph\_safe和ph\_fast测试时，就成功了。

ph\_fast测试需要：两个线程比单个线程产生至少1.25倍的每秒put次数。

## 2，具体实现（可通过ph\_safe）

1）修改notxv6/ph.c，定义互斥锁。

![在这里插入图片描述](https://img-blog.csdnimg.cn/20210305202708279.png?x-oss-process=image/watermark,type\_ZmFuZ3poZW5naGVpdGk,shadow\_10,text\_aHR0cHM6Ly9ibG9nLmNzZG4ubmV0L3UwMTM1Nzc5OTY=,size\_16,color\_FFFFFF,t\_70)

2）修改notxv6/ph.c，初始化互斥锁。

![在这里插入图片描述](https://img-blog.csdnimg.cn/20210305203637186.png?x-oss-process=image/watermark,type\_ZmFuZ3poZW5naGVpdGk,shadow\_10,text\_aHR0cHM6Ly9ibG9nLmNzZG4ubmV0L3UwMTM1Nzc5OTY=,size\_16,color\_FFFFFF,t\_70)

3）修改notxv6/ph.c，对put操作加锁。

![在这里插入图片描述](https://img-blog.csdnimg.cn/20210305203957310.png?x-oss-process=image/watermark,type\_ZmFuZ3poZW5naGVpdGk,shadow\_10,text\_aHR0cHM6Ly9ibG9nLmNzZG4ubmV0L3UwMTM1Nzc5OTY=,size\_16,color\_FFFFFF,t\_70)

## 3，执行效果

未加锁：

![在这里插入图片描述](https://img-blog.csdnimg.cn/20210305205315180.png)

简单加锁：

![在这里插入图片描述](https://img-blog.csdnimg.cn/20210305205443329.png)

## 4，具体实现（可通过ph\_fast）

1）修改notxv6/ph.c，定义互斥锁，每个bucket一个锁。

![在这里插入图片描述](https://img-blog.csdnimg.cn/20210305205648437.png)

2）修改notxv6/ph.c，初始化互斥锁。

![在这里插入图片描述](https://img-blog.csdnimg.cn/20210305205834881.png?x-oss-process=image/watermark,type\_ZmFuZ3poZW5naGVpdGk,shadow\_10,text\_aHR0cHM6Ly9ibG9nLmNzZG4ubmV0L3UwMTM1Nzc5OTY=,size\_16,color\_FFFFFF,t\_70)

3）修改notxv6/ph.c，在put中对数组某个bucket加锁。

![在这里插入图片描述](https://img-blog.csdnimg.cn/20210305205947396.png?x-oss-process=image/watermark,type\_ZmFuZ3poZW5naGVpdGk,shadow\_10,text\_aHR0cHM6Ly9ibG9nLmNzZG4ubmV0L3UwMTM1Nzc5OTY=,size\_16,color\_FFFFFF,t\_70)

## 5，执行效果

每个bucket一个锁

![在这里插入图片描述](https://img-blog.csdnimg.cn/20210305210313770.png)

# 三、Barrier

## 1，实验要求

在本作业中，你将实现barrier（应用中的一个点）：所有参与线程必须等待，直到所有其他参与线程也到达该点。

你将使用线程condition变量，一种协作技术，与xv6的sleep和wakeup相似。

你应该在一个真实电脑上做此作业（并非xv6、并非qemu）。

文件notxv6/barrier.c包含一个损坏的barrier。

```bash

$ make barrier

$ ./barrier 2

barrier: notxv6/barrier.c:42: thread: Assertion `i == t' failed.

```

2明确了在barrier同步的线程数（barrier.c中的nthread）。

每个线程执行一个循环。

在每个循环中，迭代一个线程调用barrier()，然后sleep几（一个随机数）毫秒。

断言触发，因为一个线程在另外一个线程到达barrier之前离开barrier。

期望的行为是：每个线程阻塞在barrier()中，直到所有线程调用barrier()。

你的目的是实现期望的barrier行为。

除了ph作业中看到的锁之外，你将需要下面新pthread方法；

看更多细节：[例一](https://pubs.opengroup.org/onlinepubs/007908799/xsh/pthread\_cond\_wait.html)、[例二](https://pubs.opengroup.org/onlinepubs/007908799/xsh/pthread\_cond\_wait.html)。

pthread\_cond\_wait(&cond, &mutex); // go to sleep on cond, releasing lock mutex, acquiring upon wake up

pthread\_cond\_broadcast(&cond); // wake up every thread sleeping on cond

调用pthread\_cond\_wait时，释放mutex，返回前重新获取mutex

我们已经给你barrier\_init()。

你的工作是实现barrier()，以致于panic不会发生。

我们已经为你定义struct barrier；它的属性供你使用。

有两个问题会让你的任务复杂：

你不得不处理连续的barrier调用，我们称之为一轮。

bstate.round记录当前round。你应该提升bstate.round，每次所有线程到达barrier。

你不得不处理这种情况：在其他线程退出barrier之前一个线程进入循环。

特别是，你正在从一轮到下一轮重复使用bstate.nthread变量。

确保，离开barrier、进入循环的线程不会提升bstate.nthread，当之前的round仍在使用它时。

用1、2、更多线程来测试你的代码。

## 2，具体实现

1）修改notxv6/barrier.c，实现barrier()方法。

![在这里插入图片描述](https://img-blog.csdnimg.cn/20210309100605854.png?x-oss-process=image/watermark,type\_ZmFuZ3poZW5naGVpdGk,shadow\_10,text\_aHR0cHM6Ly9ibG9nLmNzZG4ubmV0L3UwMTM1Nzc5OTY=,size\_16,color\_FFFFFF,t\_70)

## 3，执行效果

![在这里插入图片描述](https://img-blog.csdnimg.cn/20210309100855329.png)