**Lab 7 Multithreading**

1. **Uthread: switching between threads (moderate)**
   1. **实验目的**

在已有文件的基础上，为用户级线程系统设计上下文切换机制并实现(代码可以参考内核级的进程切换机制)

* 1. **实验步骤**
     1. 观察kernel/proc.h与swtch.S、proc.c等文件，了解进程切换需要保存的“上下文”是context结构体及其内容，以及切换机制。
     2. 根据kernel/proc.h与实验提示，在user/uthread.c添加用于保存进程/线程上下文的结构体context，并修改线程结构体定义使线程结构体能够包含其上下文信息。

user/uthread.c

context结构体

//从proc.h中复制的结构体内容,用于保存进程/线程上下文内容

struct context {

ra(Return Address Register):返回地址寄存器,存储函数调用返回的地址，即函数调用结束后的下一条指令的地址。

uint64 ra;

sp(Stack Pointer Register):栈指针寄存器,指向当前线程的栈顶。

uint64 sp;

通用寄存器

uint64 s0;

uint64 s1;

uint64 s2;

uint64 s3;

uint64 s4;

uint64 s5;

uint64 s6;

uint64 s7;

uint64 s8;

uint64 s9;

uint64 s10;

uint64 s11;

};

thread结构体

struct thread {

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

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

struct context context; //保存上下文用

};

* + 1. 根据kernel/swtch.S与实验提示，在user/uthread\_switch.S完善thread\_switch

user/uthread\_switch.S

复制swtch.S中记录旧上下文与切换新上下文的汇编指令

thread\_switch:

/\* YOUR CODE HERE \*/

//复制kernel/swtch.S内容,用于切换线程执行的上下文

//ra:Return Address Register:返回地址寄存器,存储函数调用返回的地址，即函数调用结束后的下一条指令的地址。

//sp:Stack Pointer Register:栈指针寄存器,指向当前线程的栈顶。

//在汇编层面实现了保存当前线程的寄存器状态到一个结构体（old），然后从另一个结构体（new）中加载寄存器状态以切换到另一个线程的执行。sd:存储双字/ld:加载双字,用于存储和加载寄存器的值,可以在切换线程时保存当前线程上下文并回复新线程上下文,实现线程的无缝切换

sd ra, 0(a0)

sd sp, 8(a0)

sd s0, 16(a0)

sd s1, 24(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 s10, 96(a1)

ld s11, 104(a1)

ret

对于实验手册的“thread\_switch 需要保存/恢复的只是被调用者保存的寄存器。为什么？”这一问题，我认为是因为因为被调用者保存的寄存器中包含了当前线程的临时数据，切换到另一个线程时需要恢复这些临时数据，以便能够从切换点继续执行。而调用者保存的寄存器一般包括调用的上下文信息，不需要在 thread\_switch 中处理，因为切换到另一个线程时，这些信息不需要变化。

这种设计能够有效地减少上下文切换的开销，使切换过程更加高效。

* + 1. 根据kernel/proc.h、proc.c的scheduler等函数与实验及现有代码提示，在user/uthread.c的thread\_create()中添加新增部分初始化代码；在 thread\_schedule()中的切换线程部分添加保存旧线程上下文与引入新线程上下文的代码。

user/uthread.c

thread\_schedule：将目前线程(current\_thread)状态存储在t中，与内核中的语句类似，调用thread\_switch以保存当前线程状态。

void

thread\_schedule(void)

{

......

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(??, ??);

\*/

//参考kernel/proc.c/scheduler中的swtch(&c->context, &p->context);语句

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

} else

next\_thread = 0;

}

thread\_create：初始调用地址为用户传进来的线程函数的地址，初始sp应该处在栈的最顶部,故需要在新建线程结构体的栈顶地址上加上STACK\_SIZE。

void

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;

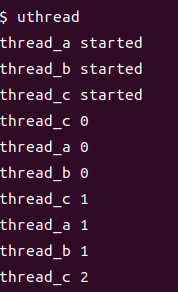
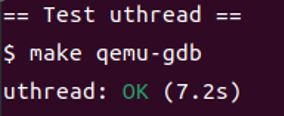
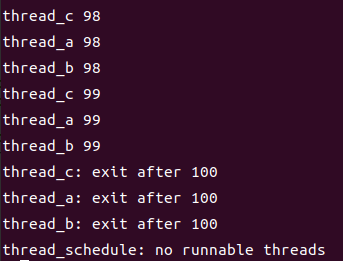
// YOUR CODE HERE

t->context.ra = (uint64) func; // 初始调用地址为用户传进来的线程函数的地址

t->context.sp = (uint64) ((char \*)&t->stack + STACK\_SIZE); // 初始sp在栈顶,故需要加上STACK\_SIZE

}

* + 1. make测试

* 1. **实验中遇到的问题和解决方法**

问题：实验开始时即使有提示页无从下手，面对需要修改的user/uthread\_switch.S文件，不知如何编写汇编语言。

解决方法：参考内核态的进程调度来实现用户态的线程调度，许多代码与思路可以直接复制。

* 1. **实验心得**

这个实验我了解了进程与线程切换的原理。对于不知从何下手的汇编语言，我发现不一定要完全了解其思路或写法，对于该实验只要能理解内核的类似代码并加以同理应用即可。至于实验手册提示的“可以在 user/uthread.asm 中看到 uthread 的汇编代码，这对于调试可能会有所帮助”，我就并没有用到也没有完全看懂。

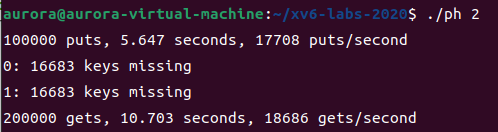
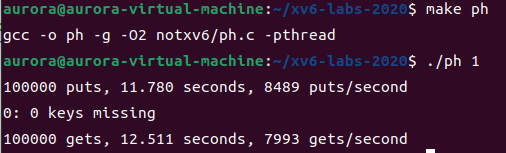
1. **Using threads (moderate)**
   1. **实验目的**

notxv6/ph.c中有一个哈希表，单进程使用时正确但多进程使用时出现错误（详细情况见实验步骤）。为哈希表的部分函数加pthread头文件中的锁相关操作以实现正确的程序。

* 1. **实验步骤**
     1. 按照实验手册运行测试命令

$ make ph

$ ./ph 1



可以看到，2线程使哈希表插入速度增快，但出现大量键值丢失。

* + 1. 根据实验手册的提示，在notxv6/ph.c中函数合理位置添加锁的定义。

pthread锁操作：

pthread\_mutex\_t lock; // 声明一个锁

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

pthread\_mutex\_lock(&lock); // 获取锁

pthread\_mutex\_unlock(&lock); // 释放锁

互斥锁可选设计：①每一位对应一个锁/②每个桶对应一个锁/③所有桶共用一个锁，粒度从小到大。于①而言， NKEY=100000的锁数组大小远比NBUCKET=5的数组大小打，且每一位的锁用过一次就较难、或要等待较久时间才能再被调用；于③而言，每一个插入操作(无论操作的桶是哪个)都需要获取锁，效率低下，甚至不如单线程。优劣显而易见，采取②方式，每个桶对应一个锁

pthread\_mutex\_t locks[NBUCKET];

在int main中初始化锁。

int

main(int argc, char \*argv[])

{

pthread\_t \*tha;

void \*value;

double t1, t0;

if (argc < 2) {

fprintf(stderr, "Usage: %s nthreads\n", argv[0]);

exit(-1);

}

//在main函数开头初始化锁

for(int i = 0; i < NBUCKET; ++i)

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

* + 1. 根据实验手册的提示，在notxv6/ph.c中函数合理位置添加锁的初始化、开关锁等相关操作。

上锁与释放锁语句在代码中的位置极为重要，此处get函数只读，不需要上锁，put函数也只需要在寻找到键值位后再上锁并开始写。。

static

void put(int key, int value)

{

int i = key % NBUCKET;

//pthread\_mutex\_lock(&locks[i]);

// is the key already present?

struct entry \*e = 0;

for (e = table[i]; e != 0; e = e->next) {

if (e->key == key)

break;

}

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

if(e){

// update the existing key.

e->value = value;

} else {

// the new is new.

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

}

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

}

static struct entry\*

get(int key)

{

int i = key % NBUCKET;

//pthread\_mutex\_lock(&locks[i]);

struct entry \*e = 0;

for (e = table[i]; e != 0; e = e->next) {

if (e->key == key) break;

}

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

return e;

}

* + 1. 在根目录新建answer-thread.txt并回答问题：Why are there missing keys with 2 threads, but not with 1 thread?

答案：没有同步机制时,一个线程的插入操作可能会覆盖另一个线程的插入操作,即insert操作非原子性(put和get也是)

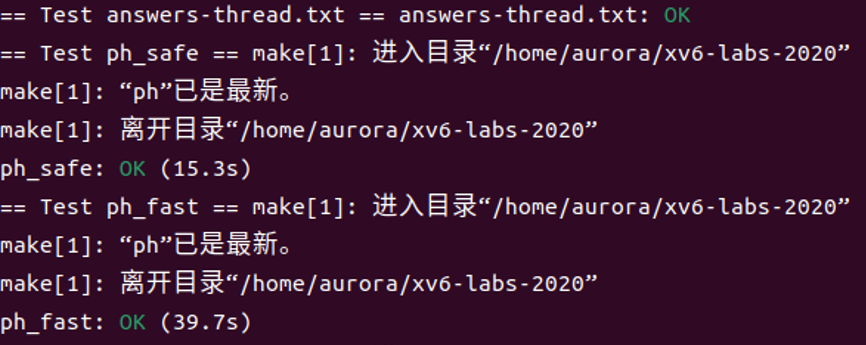
例如，考虑有两个线程同时将键值对 [4, 'd'] 和 [5, 'e'] 添加到同一个桶中：

原始桶内容为 [<1, 'a'>, <2, 'b'>, <3, 'c'>]

在 put() 函数中，它们都会迭代到链表的末尾，并决定在 <3, 'c'> 的后面插入

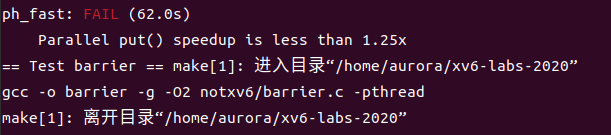
无论哪个线程执行了 '\*p = e' 这一行，都会导致另一边的改变被覆盖，从而丢失数据。

* + 1. make测试



* 1. **实验中遇到的问题和解决方法**

问题：其他实验基本通过而ph\_fast未通过，即互斥锁并没有成功优化加速。

执行$./ph 2时，任务也基本没有加速完成，甚至更慢。

解决方法：思考哈希表插入方式后，发现寻找键值位置部分仅在逐个对比，只读不写，不需要上锁，故去除get函数中的上锁开锁操作，且将put函数在开头的上锁操作移到前部分get操作结束，已经获取到位置准备插入时再上锁。

* 1. **实验心得**

互斥锁的设计需要考虑粒度大小，兼顾效率与简洁。联想到了一些写第一次大作业时遇到的困难与挑战，印象深刻。

1. **Barrier (moderate)**
   1. **实验目的**

在一个多线程环境中，当多个线程需要等待彼此达到某个点后再继续执行，就需要使用屏障。屏障允许线程暂停执行，直到满足特定的条件，然后一起继续执行。

* 1. **实验步骤**
     1. 分析barrier结构体

pthread\_mutex\_t barrier\_mutex: 互斥锁，用于保护对 barrier 结构体中的数据的并发访问。在访问 barrier 结构体中的变量时，需要先获取该锁，以确保多个线程不会同时修改这些变量。

pthread\_cond\_t barrier\_cond: 条件变量，用于实现线程在达到屏障点时的等待和唤醒操作。当一个线程达到屏障点时，它会等待在这个条件变量上，直到所有参与的线程都达到了屏障点，然后通过调用pthread\_cond\_broadcast 唤醒所有等待在该条件变量上的线程。

int nthread: 记录达到当前屏障点的线程数量。每个线程达到屏障点时会增加这个计数器，当计数器达到预期值时，表示所有线程都已经到达，可以执行下一轮的屏障操作。

int round: 记录当前屏障的轮数。每次所有线程都到达屏障点后，会增加这个计数器，表示进入了下一轮的屏障操作。

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;

* + 1. 根据注释及实验手册提示编写barrier函数

notxv6/barrier.c

static void

barrier()

{

// YOUR CODE HERE

//

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

// then increment bstate.round.

//

首先上锁(barrier\_mutex)

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

检查当前已经调用barrier()的线程数量是否达到了总线程数nthread。如果是最后一个线程到达，就意味着所有线程都已经到达屏障点。如果是最后一个线程到达，则将bstate.nthread重置为0，将bstate.round（表示轮数）递增并使用pthread\_cond\_broadcast函数广播信号给所有等待在bstate.barrier\_cond条件变量上的线程，以唤醒它们。

if(++bstate.nthread == nthread) {

bstate.nthread = 0;

bstate.round++;

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

}

// 如果不是最后一个线程到达，则当前线程通过pthread\_cond\_wait函数在bstate.barrier\_cond条件变量上等待，同时释放之前获取的互斥锁。这将使得线程暂停执行，直到被唤醒。

else {

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

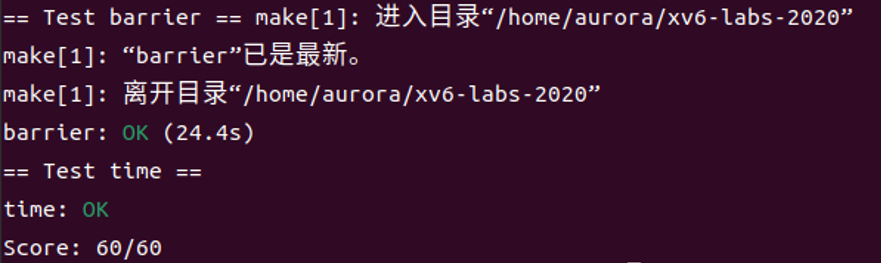
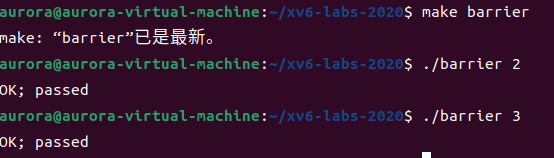
}

最后释放锁(barrier\_mutex)

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

}

* + 1. make测试



* 1. **实验中遇到的问题和解决方法**

问题：理解barrier结构体中的pthread\_cond\_t barrier\_cond及与其相关的广播操作时有些不理解。

解决方式：学习他人笔记完成

* 1. **实验心得**

barrier的机制可以用于协调多个线程的同步操作，在等到必要的线程全部到期前，所有线程都会停留在屏障点，这个机制有点像生产者消费者模式。