Lab 6 Multithreading

目录

[1. Uthread: switching between threads (moderate) 1](#_Toc174491325)

[1.1 实验目的 1](#_Toc174491326)

[1.2 实验步骤 1](#_Toc174491327)

[1.3 实验结果 4](#_Toc174491328)

[1.4 遇到的问题 5](#_Toc174491329)

[1.5 实验心得 6](#_Toc174491330)

[2. Using threads (moderate) 7](#_Toc174491331)

[2.1 实验目的 7](#_Toc174491332)

[2.2 实验步骤 7](#_Toc174491333)

[2.3 实验结果 10](#_Toc174491334)

[2.4 实验中遇到的问题和解决方法 10](#_Toc174491335)

[2.5 实验心得 10](#_Toc174491336)

[3. Barrier (moderate) 11](#_Toc174491337)

[3.1 实验目的 11](#_Toc174491338)

[3.2 实验步骤 12](#_Toc174491339)

[3.3 实验结果 13](#_Toc174491340)

[3.4 遇到的问题 13](#_Toc174491341)

[3.5 实验心得 14](#_Toc174491342)

[4. 测试结果 15](#_Toc174491343)

Uthread: switching between threads (moderate)

* 1. 实验目的
* 理解用户级线程系统的概念
* 理解上下文切换的概念
* 设计线程切换机制,需要设计一个机制来保存和恢复线程的寄存器状态，以便在线程之间进行切换
* 理解寄存器的保存和恢复
  1. 实验步骤
* 修改 struct thread ：在 struct thread 中增加保存寄存器的字段，同时新建线程上下文结构体的定义 thread\_context

// 线程的上下文

struct thread\_context {

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;

};

struct thread {

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

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

struct thread\_context context; // 线程的上下文

};

* 实现 thread\_create() ：创建线程并初始化其栈和寄存器。为新创建的线程初始化必要的上下文，使其能够正确执行传入的函数并在自己的栈空间上运行。这样，当调度器第一次调度该线程时，线程会从传入的函数开始执行，并且使用分配给它的栈空间。

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; // 执行传入的函数（thread\_switch）

t->context.sp = (uint64)(t->stack + STACK\_SIZE); // 复制栈顶指针

}

* 实现 thread\_schedule() ：调度可运行的线程，调用 thread\_switch 进行上下文切换。

void

thread\_schedule(void)

{

struct thread \*t, \*next\_thread;

/\* Find another runnable thread. \*/

next\_thread = 0;

t = current\_thread + 1;

for(int i = 0; i < MAX\_THREAD; i++){

if(t >= all\_thread + MAX\_THREAD)

t = all\_thread;

if(t->state == RUNNABLE) {

next\_thread = t;

break;

}

t = t + 1;

}

if (next\_thread == 0) {

printf("thread\_schedule: no runnable threads\n");

exit(-1);

}

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(??, ??);

\*/

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

} else

next\_thread = 0;

}

* 实现 thread\_switch ：在汇编中保存当前线程的寄存器，恢复下一个线程的寄存器，完成上下文切换。

swtch:

sd ra, 0(a0) // 保存返回地址寄存器

sd sp, 8(a0) // 保存栈指针寄存器

sd s0, 16(a0) // 保存s0寄存器

sd s1, 24(a0) // 保存s1寄存器

sd s2, 32(a0) // 保存s2寄存器

sd s3, 40(a0) // 保存s3寄存器

sd s4, 48(a0) // 保存s4寄存器

sd s5, 56(a0) // 保存s5寄存器

sd s6, 64(a0) // 保存s6寄存器

sd s7, 72(a0) // 保存s7寄存器

sd s8, 80(a0) // 保存s8寄存器

sd s9, 88(a0) // 保存s9寄存器

sd s10, 96(a0) // 保存s10寄存器

sd s11, 104(a0) // 保存s11寄存器

ld ra, 0(a1) // 恢复返回地址寄存器

ld sp, 8(a1) // 恢复栈指针寄存器

ld s0, 16(a1) // 恢复s0寄存器

ld s1, 24(a1) // 恢复s1寄存器

ld s2, 32(a1) // 恢复s2寄存器

ld s3, 40(a1) // 恢复s3寄存器

ld s4, 48(a1) // 恢复s4寄存器

ld s5, 56(a1) // 恢复s5寄存器

ld s6, 64(a1) // 恢复s6寄存器

ld s7, 72(a1) // 恢复s7寄存器

ld s8, 80(a1) // 恢复s8寄存器

ld s9, 88(a1) // 恢复s9寄存器

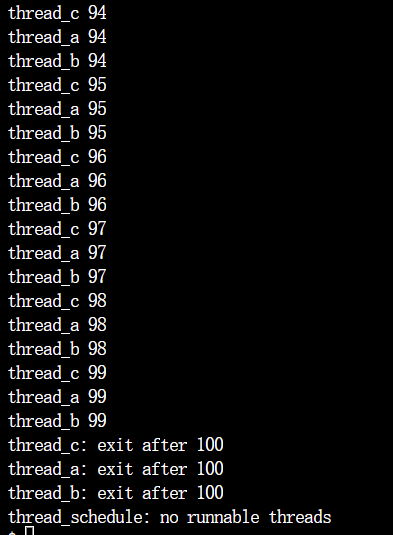
ld s10, 96(a1) // 恢复s10寄存器

ld s11, 104(a1) // 恢复s11寄存器

ret // 返回到ra指示的地址

* 1. 实验结果
* 运行 make qemu 后执行 uthread ：

·······



* 1. 遇到的问题
* 问题：

在进行线程切换时，需要正确保存当前线程的寄存器状态，并能够恢复到切换到的线程的寄存器状态。但是怎么确保寄存器状态保存或恢复正确呢？这个我在一开始并没有什么思路。在理论课的学习中老师讲过什么叫线程的上下文切换，我大概知道是要把当前线程的相关信息和CPU状态存储起来，然后把要换进来的线程的相关信息导入，但是对于是什么信息，怎么保存，存到哪这些细节我是不懂的，而且这次修改的文件是.S文件，有点特别，就让我感到更加疑惑了。

* 解决方法：

后来我看到了实验提示提示“ thread\_switch needs to save/restore only the callee-save registers.”思考了一下，这是因为在函数调用中，寄存器分为两类：调用者保存寄存器（caller-save registers）和被调用者保存寄存器（callee-save registers）。调用者保存寄存器在函数调用前由调用者保存，函数返回后由调用者恢复。被调用者保存寄存器在函数调用前由被调用者保存，函数返回后由被调用者恢复。在上下文切换时，thread\_switch 函数只需要保存和恢复被调用者保存寄存器。这是因为在线程切换之前，当前线程的调用者保存寄存器已经由编译器生成的代码保存了。当线程切换回来时，这些寄存器会由编译器生成的代码恢复。因此， thread\_switch 函数只需要负责保存和恢复被调用者保存寄存器，以确保线程切换回来时能够正确恢复它们的值。有了这个提示，我的方向就比较清晰了。

* 1. 实验心得

通过实现用户级线程系统，我了解到线程是操作系统调度的基本单位，而线程的状态及上下文（寄存器值）的保存与恢复是线程切换的核心。线程的上下文包含了线程执行的必要信息，例如返回地址、栈指针以及其他寄存器的值。

上下文切换的核心在于准确地保存当前线程的所有寄存器状态，并在切换到下一个线程时恢复这些状态。通过自己编写 thread\_switch 汇编函数并理解其工作原理，我更清晰地认识到上下文切换的复杂性和必要性。

Using threads (moderate)

* 1. 实验目的
* 理解并行编程的概念
* 理解线程和锁的概念
* 掌握并发编程中的竞态条件问题
* 学会使用互斥锁解决竞态条件问题
  1. 实验步骤
* 学习 pthreads 的使用方法
* 学习 ph.c 文件中哈希表的实现并进行调整，ph.c 文件实现了一个简单的哈希表，并且使用多线程进行并行操作
* 数据结构

#define NBUCKET 5

#define NKEYS 100000

struct entry {

int key;

int value;

struct entry \*next;

};

struct entry \*table[NBUCKET];

int keys[NKEYS];

int nthread = 1;

* 插入函数

static void

insert(int key, int value, struct entry \*\*p, struct entry \*n)

{

struct entry \*e = malloc(sizeof(struct entry));

e->key = key;

e->value = value;

e->next = n;

\*p = e;

}

* put函数

static void put(int key, int value)

{

int i = key % NBUCKET;

// is the key already present?

struct entry \*e = 0;

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

if (e->key == key)

break;

}

if (e) {

// update the existing key.

e->value = value;

}

else {

// the new is new.

pthread\_mutex\_lock(&lock[i]); // 对第i个bucket加锁

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

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

}

}

* get函数

static struct entry\*

get(int key)

{

int i = key % NBUCKET;

struct entry \*e = 0;

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

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

}

return e;

}

* 线程函数

static void \*

put\_thread(void \*xa)

{

int n = (int) (long) xa; // thread number

int b = NKEYS/nthread;

for (int i = 0; i < b; i++) {

put(keys[b\*n + i], n);

}

return NULL;

}

static void \*

get\_thread(void \*xa)

{

int n = (int) (long) xa; // thread number

int missing = 0;

for (int i = 0; i < NKEYS; i++) {

struct entry \*e = get(keys[i]);

if (e == 0) missing++;

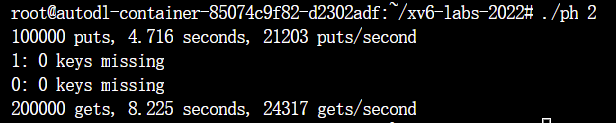
}

printf("%d: %d keys missing\n", n, missing);

return NULL;

}

* 在notxv6/ph.c中加入pthread\_mutex\_t locks[NBUCKET];
* 在notxv6/ph.c中修改put函数。实现了一个哈希表的 put 操作。它接受一个键和一个值作为参数。函数首先计算键所在的桶，并锁定该桶以保证线程安全。然后，函数遍历该桶中的所有条目，查找是否存在与键匹配的条目。如果找到匹配的条目，则更新其值；否则，插入一个新条目。最后，函数解锁该桶。
* 在notxv6/ph.c的main函数中初始化 locks
  1. 实验结果



* 1. 实验中遇到的问题和解决方法
* 问题：

由于多个线程同时访问和修改共享资源，我需要考虑数据的一致性或者丢失的隐患。

* 解决方法：

通过上网查阅资料和之间课程的学习，我发现使用互斥锁来保护共享资源是一个很好的方法，这样金可以确保同一时间只有一个线程可以访问或修改。例如，在 put 函数中，可以在关键部分之前获取互斥锁，执行插入操作后释放锁。这样，每个线程在操作哈希表时都会先获取锁，避免了竞态条件问题。

* 1. 实验心得

通过学习和使用 pthreads 库，我理解了如何在程序中创建和管理多个线程来实现并行操作。并行编程能够充分利用多核处理器的优势，加速任务的执行。在这次实验中，通过多线程对哈希表进行操作，我体会到了并行编程的实际应用场景和重要性。

竞态条件是并发编程中常见的问题之一。当多个线程同时操作共享资源时，如果不加以控制，可能会导致数据的竞态条件问题。在实验中，如果不使用互斥锁，可能会出现多个线程同时插入数据的情况，从而导致哈希表结构损坏。通过实验，我掌握了如何识别和避免竞态条件问题。

在实验中，我使用了 pthread\_mutex\_lock 和 pthread\_mutex\_unlock 对哈希表的桶进行加锁和解锁操作。这让我理解了互斥锁的作用，即通过确保只有一个线程能够进入临界区来解决竞态条件问题。互斥锁的使用大大提高了程序的安全性和可靠性，保证了并发操作的正确性。

Barrier (moderate)

* 1. 实验目的
* 理解屏障的概念
* 学习使用pthread条件变量，学习如何使用pthread\_cond\_wait函数使线程等待条件满足，并使用pthread\_cond\_broadcast函数通知所有等待线程继续执行
* 理解并处理竞态条件，学习如何使用互斥锁来保护共享变量，并确保在关键时刻只有一个线程可以访问或修改变量，以避免竞态条件的发生
* 掌握多线程编程技巧，学会如何协调和同步多个线程的执行顺序，以实现预期的并发行为
  1. 实验步骤
* 理解屏障：屏障(barrier)，即一个应用程序中的同步点，所有参与的线程必须在该点等待，直到所有其他参与线程也达到该点。应对连续的屏障调用，每一轮都应该确保所有线程同步到达屏障并正确处理。
* 理解 notxv6/barrier.c 中的实现，程序创建多个线程，每个线程执行一个循环。在每次循环迭代中，每个线程调用 barrier( ) ，而后在随机休眠一段时间。要让每个线程都阻塞在barrier( ) 处，只有当所有线程都调用了 barrier() 再继续执行。
* pthread\_cond\_wait(&cond, &mutex);这个函数让线程进入睡眠状态，等待条件变量 cond 满足，同时释放互斥锁 mutex 当条件变量满足时，线程被唤醒并重新获取互斥锁 mutex ，然后继续执行
* pthread\_cond\_broadcast(&cond);

这个函数唤醒所有正在等待条件变量 cond 的线程

* 对于barrier调用，所有线程的一次调用为一轮，当所有线程都到达屏障时需要将bstate.round++
* 实现 barrier() 函数，使其在所有线程调用 barrier() 之前都保持阻塞状态，并且增加bstate.round

static void barrier()

{

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

bstate.nthread++; // 增加已到达屏障的线程数

if (bstate.nthread == nthread) {

// 当所有线程都到达时（bstate.nthread == nthread）

bstate.round++; // 新增一轮

bstate.nthread = 0; // 重置线程数

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

} else {

// 当前线程进入睡眠状态等待

int current\_round = bstate.round;

while (current\_round == bstate.round) {

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

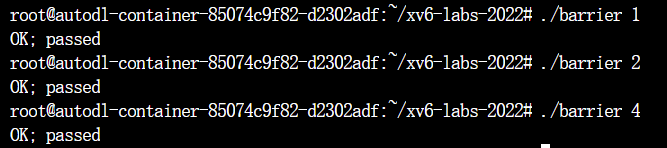
}

}

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

}

* 1. 实验结果



* 1. 遇到的问题
* 问题：

多个线程同时访问和修改共享变量可能导致竞态条件。我在增加已调用 barrier 函数的线程数之前，另一个线程可能检查并使用了旧的值，从而导致不正确的轮数计算或线程等待的条件不满足

* 解决方法：

使用互斥锁（ pthread\_mutex\_lock 和 pthread\_mutex\_unlock ）来保护共享变量的访问和修改。在修改共享变量之前，获取互斥锁以确保在关键时刻只有一个线程可以访问变量，从而避免竞态条件

* 1. 实验心得

屏障（Barrier）是一种同步机制，所有线程必须在屏障点等待，直到所有线程都到达该点，然后才可以继续执行。在多线程编程中，屏障非常适合用于需要同步多个线程的场景，如循环中的每一轮操作都需要所有线程完成后才能进入下一轮。通过这次实验，我掌握了如何在实际编程中实现屏障机制，以确保所有线程都同步到达某个状态点。

pthread\_cond\_wait 和 pthread\_cond\_broadcast 是本次实验中使用的两个关键函数。通过 pthread\_cond\_wait 函数，我理解了如何让线程在某个条件不满足时进入等待状态，并释放占用的互斥锁。另一方面，pthread\_cond\_broadcast 函数能够唤醒所有等待该条件的线程，使得它们能够继续执行。这些函数的使用让我掌握了如何通过条件变量实现线程间的同步。

在多线程编程中，竞态条件是一个常见且棘手的问题。当多个线程并发地访问或修改共享资源时，如果没有适当的同步机制，可能会导致数据不一致或程序崩溃。在实验中，通过使用互斥锁来保护共享变量 bstate.nthread，我确保了在关键时刻只有一个线程能够修改这个变量，从而避免了竞态条件的发生。

测试结果

