操作系统实验PINTOS

PROJECT 1:THREADS

设计文档

1. 小组成员以及贡献

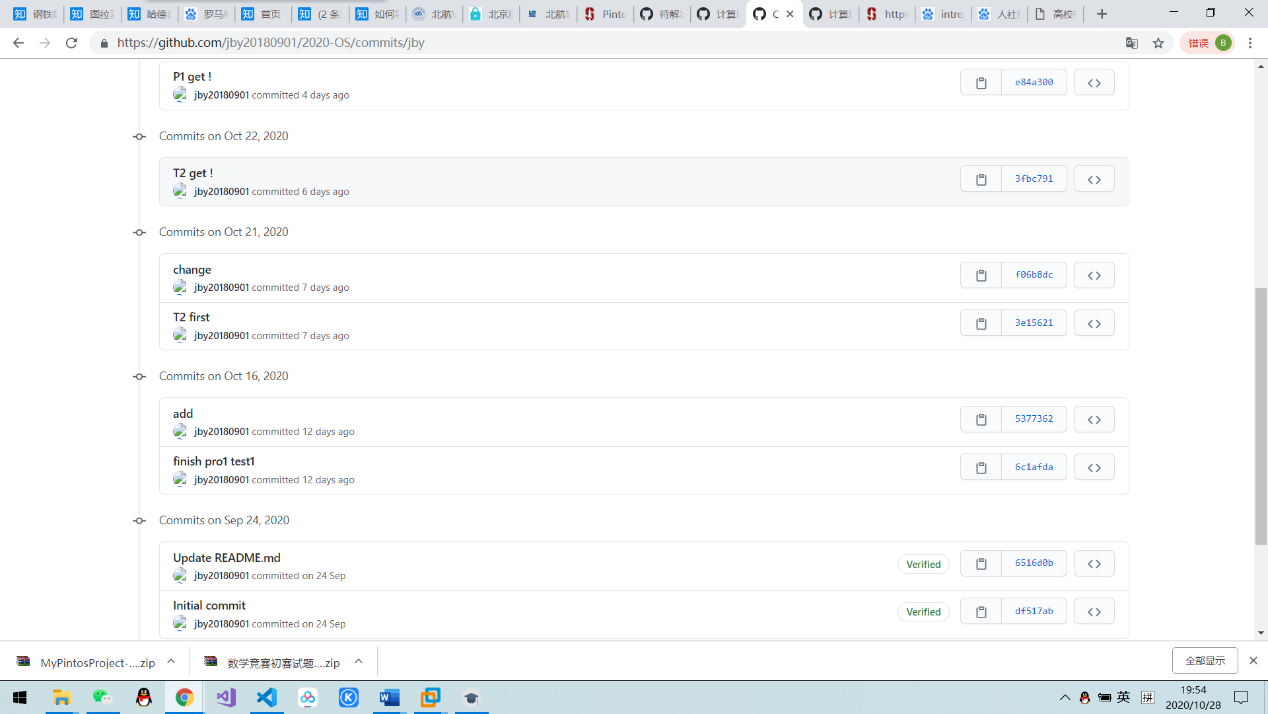
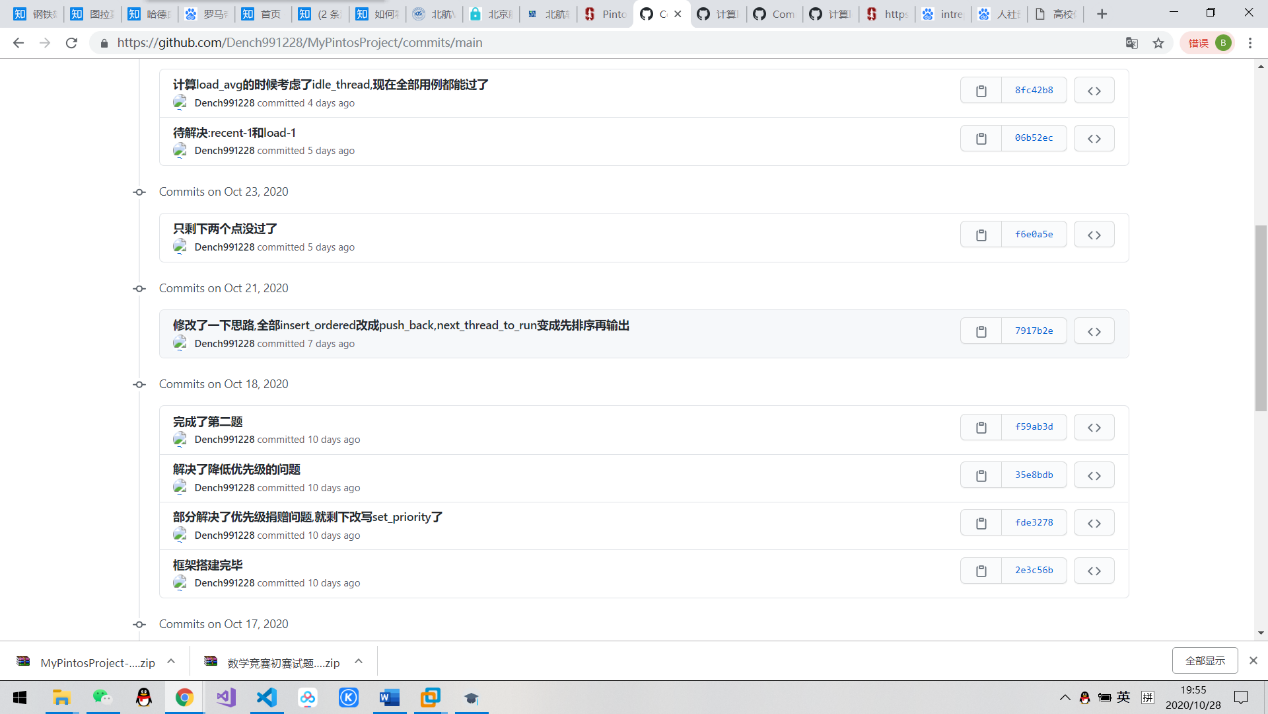
18373403 姜宝洋 25%

18373570 邓涵之 25%

18373133 方洁 25%

18373744 姜昊 25%

总共100%



1. 实验设计与相关问题

2.1 重构timer\_sleep()

2.1.1 数据结构

A1 在此复制每个新的或更改的struct或的声明struct成员，全局或静态变量，“ typedef”或枚举。用25个字以内的单词来确定每个单词的目的。

在结构体`struct thread`添加了成员：

bool isSleep;//线程是否处于睡眠状态，睡眠则为1，反之为0

/\*the time when thread start to sleep\*/

int64\_t start\_tick;//记录线程开始睡眠的时间

/\*the time thread going to sleep\*/

int64\_t sleep\_tick;//线程将要睡眠多长时间

struct thread

{

/\* Owned by thread.c. \*/

tid\_t tid; /\* Thread identifier. \*/

enum thread\_status status; /\* Thread state. \*/

char name[16]; /\* Name (for debugging purposes). \*/

uint8\_t \*stack; /\* Saved stack pointer. \*/

int priority; /\* Priority. \*/

struct list\_elem allelem; /\* List element for all threads list. \*/

/\* Shared between thread.c and synch.c. \*/

struct list\_elem elem; /\* List element. \*/

#ifdef USERPROG

/\* Owned by userprog/process.c. \*/

uint32\_t \*pagedir; /\* Page directory. \*/

#endif

/\*is the thread sleeping\*/

bool isSleep;//线程是否处于睡眠状态，睡眠则为1，反之为0

/\*the time when thread start to sleep\*/

int64\_t start\_tick;//记录线程开始睡眠的时间

/\*the time thread going to sleep\*/

int64\_t sleep\_tick;//线程将要睡眠多长时间

/\* Owned by thread.c. \*/

unsigned magic; /\* Detects stack overflow. \*/

};

2.1.2 算法

A2 简要描述timer\_sleep函数里面发生了什么，包括时间中断处理程序对它的影响

在使用`timer\_sleep()`函数时，首先获取当前系统的时间，其次获取当前正在运行的线程的信息，然后将设置 线程结构体内的睡眠状态判断变量为真，表示线程将进入睡眠状态，并设置睡眠起始时间为刚刚获得的当前系统时间，睡眠时长为传给函数的参数`ticks`，最后执行一个原子性操作`thread\_block();`，使当前运行的线程进入阻塞状态。

对于`timer interrupt handler`也就是`timer\_interrupt ()`函数，增加了一句`thread\_foreach(checkSleep,NULL);`，用以遍历所有线程，检查睡眠时间，将结束睡眠时间的线程放进就绪队列里。

A3 我们可以采取什么措施，使得时间中断造成的开销最小。

在`timer.c`中增加函数`checkSleep(struct thread\* t, void\* aux)`来检查每个进程中的是否已经到达了唤醒时间。在时间中断函数运行时，遍历查询队列里每个线程的状态，如果线程已到唤醒时间，则唤醒该线程，加入到等待队列中，此时中断函数只需要遍历一次就可以更改所有线程的状态，节约了大量的时间。

void checkSleep(struct thread\* t, void\* aux){

if(t->status==THREAD\_BLOCKED&&t->isSleep){//正在睡眠并且已经阻塞

if(timer\_elapsed(t->start\_tick)>=t->sleep\_tick){//放入到正在准备的队列

//list\_push\_back(&ready\_list, &t->allelem);

t->isSleep=false;

thread\_unblock(t);

}

}

}

2.1.3 同步

A4 当多个线程同时调用timer\_sleep（）时，如何避免竞争条件？

只要是处于一下两端代码之间的操作，均为原子性操作，无法被中断。不论调用函数对线程队列去遍历、插入还是删除，我们的操作内容均放在以下两段代码之间，保证了操作的独立性。

asm volatile ("cli" : : : "memory");

···

asm volatile("sti":::"memory");

A5 在调用timer\_sleep（）期间发生计时器中断时，如何避免竞争条件？

调用`timer\_sleep()`函数时，我们将主要操作放在了

asm volatile ("cli" : : : "memory");

···

asm volatile("sti":::"memory");

代码段之间，屏蔽了时钟中断对函数调用的影响，在调用`timer\_sleep()`时，时钟是无法被中断的。

2.1.4 基本原理

A6 你为什么选择了这个设计？它在哪一方面是优秀的呢？

基于我们的设计，将`timer\_sleep()`函数原本的忙等状态修改为设置睡眠状态以及睡眠时间，减少了不必要的在就绪队列和运行队列转换所消耗的时间，更加节省系统资源。

2.2 优先级调度

* + 1. 数据结构

B1 在此复制每个新的或更改的struct或的声明struct成员，全局或静态变量，“ typedef”或枚举。用25个字以内的单词来确定每个单词的目的。

（1）、thread结构体

/\*the lock the thread is waiting for\*/

   struct lock\* lock\_wait;

   /\*the list of locks that the thread is holding\*/

   struct list locks;

   /\*the original priority\*/

   int base\_priority;

（2）、lock结构体

struct list\_elem elem;

B2 解释用于跟踪优先级捐赠的数据结构。 使用ASCII艺术作品来绘制嵌套捐赠。

|Thread 1|low| //首先我们有一个低优先级的线程拥有一个锁lock1

|Thread 2|mid|->|Thread 1|low|//然后一种中优先级的线程被lock1阻塞

|Thread 2|mid|->|Thread 1|mid|//那么thread2就将优先级donate给thread1

|Thread 3|hig|->|Thread 2|mid|->|Thread 1|mid|//现在一个更高优先级的线程想要lock

|Thread 3|hig|->|Thread 2|hig|->|Thread 1|mid|//于是donate优先级

|Thread 3|hig|->|Thread 2|hig|->|Thread 1|hig|//新的高优先级的线程会继续donate给第一个线程

* + 1. 算法

B3 如何确保等待锁，信号灯或条件变量的优先级最高的线程首先唤醒？

每当一个锁、信号量被释放时，我们就会对所有等待的线程进行检查，并唤醒所有线程中优先级最高的线程。然后将这个线程从等待列表中删除。对于条件变量，我们一样是寻找有最高优先级线程的信号量。

B4 描述对lock\_acquire()的调用导致优先级捐赠时的事件顺序。嵌套捐赠如何处理？

如下处理：

/\*update the lock we're waiting for\*/

    thread\_current()->lock\_wait = lock;

    struct lock \*l = lock;

    struct thread \*receiver = l->holder;

    while(l!=NULL&&receiver!=NULL&&receiver->priority<thread\_current()->priority){

      donatePriority(receiver, thread\_current()->priority);

      l = receiver->lock\_wait;

      if(l==NULL){// just sort the ready\_list because the donated wasn't waiting for a lock

        thread\_yield();

      }

      receiver = l->holder;

    }

当当前调用的线程需要的锁有另一个线程持有，且这个线程是低优先级线程，当前线程马上进行优先级捐赠。当被捐赠的线程的优先级仍然高于其等待锁的持有者的优先级，那么就将继续优先级捐赠，直到递归道德线程不等待任何锁，或者比等待锁的持有者的优先级高。

B5 描述在较高优先级线程正在等待的锁上调用lock\_release()时的事件顺序。

enum intr\_level old\_level = intr\_disable();

  release\_lock(lock);

  //TODO: restore the priority before donation

  if(!thread\_mlfqs)restorePriority();

  lock->holder = NULL;

  intr\_set\_level(old\_level);

/\*remove the lock from acquired locks\*/

void release\_lock(struct lock \*l){

  list\_remove(&l->elem);

}

/\*return to the priority whose waiter has the highest priority if it is bigger than base\_priority\*/

void restorePriority(){

  if(list\_empty(&thread\_current()->locks)){

    //printf("No locks at hand!\n");

    thread\_current()->priority = thread\_current()->base\_priority;

  }

  else{

    int priority\_lower\_bound = 0;

    /\*run through all the locks holden by the current thread\*/

    for(struct list\_elem \*temp\_lock\_elem = list\_begin(&thread\_current()->locks);temp\_lock\_elem!=list\_end(&thread\_current()->locks); temp\_lock\_elem = list\_next(temp\_lock\_elem)){

      struct lock\* cur\_lock = list\_entry(temp\_lock\_elem, struct lock, elem);

      /\*find the waiter with highest priority\*/

      for(struct list\_elem \*temp\_thread\_elem = list\_begin(&cur\_lock->semaphore.waiters);temp\_thread\_elem!= list\_end(&cur\_lock->semaphore.waiters); temp\_thread\_elem = list\_next(temp\_thread\_elem)){

        struct thread\* cur\_thread = list\_entry(temp\_thread\_elem, struct thread, elem);

        priority\_lower\_bound = priority\_lower\_bound>cur\_thread->priority?priority\_lower\_bound:cur\_thread->priority;

      }

    }

    if(priority\_lower\_bound>thread\_current()->base\_priority){

      thread\_current()->priority = priority\_lower\_bound;

    }

    else{

      thread\_current()->priority = thread\_current()->base\_priority;

    }

  }

}

首先，我们要明确，释放锁的线程并不一定返回原来的优先级，当他的持有的其他锁（不包括释放的锁）的中存在某个等待锁的线程的优先级，比释放锁的该线程的优先级高时，我们的这个线程的优先级就需要恢复到，它所持有的锁中（不包括释放的锁），等待这些锁的线程的最高优先级。这样，我们就可以避免多次的优先级捐赠。

* + 1. 同步

B6 在thread\_set\_priority()中描述潜在的竞争，并说明您的实现如何避免这种情况。您可以使用锁来避免这场竞争吗？

/\* Sets the current thread's priority to NEW\_PRIORITY. \*/

void

thread\_set\_priority (int new\_priority)

{

  if(list\_empty(&thread\_current()->locks)){

    thread\_current()->priority = new\_priority;

  }

  else{

    if(new\_priority>thread\_current()){

      thread\_current()->priority = new\_priority;

    }

  }

  thread\_current()->base\_priority = new\_priority;

  if(list\_entry(list\_max(&ready\_list, comparePriorityElem,NULL),struct thread, elem)->priority>new\_priority){

    thread\_yield();

  }

}

当新优先级被赋予，线程中存储的原优先级，立即改变为新的优先级。而正在使用的优先级并不一定要改变。

当这个新优先级大于当前优先级时，改变当前优先级。

当这个线程持有的锁没有阻塞其他线程，改变。

当这个线程持有的锁阻塞其他线程且新优先级小于当前优先级，那么当前优先级变为MAX（新优先级，被阻塞的线程的优先级的最大值）

* + 1. 基本原理

B7 你为什么选择了这个设计？它在哪一方面是优秀的呢？

这种设计很稳定。相比其他的设计，我们的设计确保了捐赠立即发生，不会对之后的事件产生影响。

2.3 高级调度程序

2.3.1数据结构

C1：在此复制每个新的或更改的struct或的声明struct成员，全局或静态变量，“ typedef”或枚举。用25个字以内的单词来确定每个单词的目的。

（1）、浮点数计算

typedef int myfloat;

/\*偏移量\*/

#define shift 16

/\*把整数转换为定点数\*/

#define convert\_float(n) ((myfloat)(n<<shift))

/\*把定点数转换为整数\*/

#define convert\_integer(x) (x>0?(int)((x+(1<<shift)/2)>>shift):(int)((x-(1<<shift)/2)>>shift))

/\*定点数相乘\*/

#define mult(x,y) (((int64\_t)x\*(int64\_t)y)>>shift)

/\*定点数相除\*/

#define div(x,y) ((int64\_t)x<<shift)/y

（2）、thread结构体

/\*the niceness of a thread\*/

    int nice;

/\*recent cpu of a thread\*/

   myfloat recent\_cpu;

（3）、全局变量

myfloat load\_avg;

2.3.2 算法

C2：假设线程A，B和C的nice值分别为0、1和2。每个的recent\_cpu值均为0。填写下表，显示调度决策以及每个进程的priority和recent\_cpu值

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Timer | Recent\_cpu | | | Priority | | | Thread |
| Ticks | A | B | C | A | B | C | To run |
| 0 | 0 | 0 | 0 | 63 | 61 | 59 | A |
| 4 | 4 | 0 | 0 | 62 | 61 | 59 | A |
| 8 | 8 | 0 | 0 | 61 | 61 | 59 | A |
| 12 | 12 | 0 | 0 | 60 | 61 | 59 | B |
| 16 | 12 | 4 | 0 | 60 | 60 | 59 | B |
| 20 | 12 | 8 | 0 | 60 | 59 | 59 | A |
| 24 | 16 | 8 | 0 | 59 | 59 | 59 | A |
| 28 | 20 | 8 | 0 | 58 | 59 | 59 | B |
| 32 | 20 | 12 | 0 | 58 | 58 | 59 | C |
| 36 | 20 | 12 | 4 | 58 | 58 | 58 | A |

C3：那些在任何模糊调度程序规范使表中的值不确定？如果是这样，您使用什么规则来解决这些问题？这是否与您的调度程序的行为相符？

同优先级线程运行顺序问题。当出现一下两种情况：当前线程优先级和等待队列中的线程优先级相同时、等待队列中最高优先级线程有多个时，我们应该让那个线程先运行？

实际上我们并不能确定他们的运行关系，因为在给线程优先级排序的时候，我们使用了系统自带的sort函数，而sort函数是不稳定的，我们并不能确定同优先级线程在快排的时候，是否被改变了顺序。

C4：您如何分摊那些可能会影响性能的，处于中断内部与中断外部的上下文切换的时间花费？

为了避免“关中断”操作中占用的时间过长，影响系统时间中断的正常运行，我们尽量将代码写在关中断之外，只有那些需要被原子地运行的操作，才会写入到关中断代码块中，以减少关中断时间过长对时钟中断的影响。

2.3.3 基本原理

C5：简要评析您的设计，指出您选择的设计中的优点和缺点。如果您有更多的时间来处理项目的这一部分，您将如何选择改进或改进您的设计？

设计：

（1）、题目中要求：每一个tick，增加当前线程的recent\_cpu；每4个ticks，更新所有线程的优先级；每秒钟（100个ticks）更新所有线程的优先级、recent\_cpu，并且更新load\_avg的值。

因为这个要求与时间中断密切相关，所以我们将更新优先级、recent\_cpu、load\_avg的函数写在timer\_interrupt函数里面，每一个时间中断tick就进行一次操作，进行操作的时候屏蔽中断，达到原子操作的效果。

（2）、我们在调取等待队列的大小值时，没有用list\_size函数（因为出现了一些问题）我们利用thread\_foreach函数使每个运行中和等待中的线程修改同一个全局变量，达到计数的目的。

改进：在维护各个优先队列的时候，我们发现pintos自带的关于链表的操作函数有一定的缺陷，而且还可能被我们之前修改的代码影响（比如本组在做该部分实验时，list\_size函数出现了一些问题）如果再有一些时间，我们可以对pintos自带的list函数进行修改。

C6：作业详细说明了定点数学的算术，但它使您可以实施它。 您为什么决定以自己的方式实施它？ 如果您为定点数学创建了一个抽象层，即一个抽象数据类型和/或一组函数或宏来操纵定点数，为什么要这么做？ 如果没有，为什么不呢？

实现浮点数的添加代码如下：

typedef int myfloat;

/\*偏移量\*/

#define shift 16

/\*把整数转换为定点数\*/

#define convert\_float(n) ((myfloat)(n<<shift))

/\*把定点数转换为整数\*/

#define convert\_integer(x) (x>0?(int)((x+(1<<shift)/2)>>shift):(int)((x-(1<<shift)/2)>>shift))

/\*定点数相乘\*/

#define mult(x,y) (((int64\_t)x\*(int64\_t)y)>>shift)

/\*定点数相除\*/

#define div(x,y) ((int64\_t)x<<shift)/y

由于在内核中直接实现浮点数的相加需要大量的开销，对整个内核的性能造成了巨大的影响，所以我们还是用整型变量的计算来模拟浮点数的计算。

我们将整型变量向左偏移16个二进制位，来模拟浮点数，留出的16位二进制位就是表示浮点数小数的位置。

3、实验结果

