# 操 作 系 统

实验报告题目：Pintos实验1

**组 号: 36**

**所在学院: 软件学院**

需求分析

1. 睡眠和唤醒

既有的线程挂起机制为通过在timer\_sleep()中执行while()循环来实现。当前时间若不满足挂起的时间要求，则调用thread\_yield()函数继续循环；如果满足则直接压进就绪队列。因此实际上只存在两个态， Running 和 Ready,并没有真正意义上的睡眠与唤醒。查看源代码中 thread\_yield()函数的注释：

*/\* Yields the CPU. The current thread is not put to sleep and*

*may be scheduled again immediately at the scheduler's whim. \*/*

可以发现线程并没有真正进入睡眠，且该线程有可能又一次立即被调度，这样的结果就是产生了忙等待。

我们需要实现的目标为，在timer\_sleep()函数中使线程进入Block态。系统运行一段时间后，睡眠时间到，再对该线程进行唤醒，从Block态转入Ready态。

1. 优先级调度

优先级调度要求实现一个基于优先级的调度策略。包括解决优先级授予和其他授予时的相关问题。

优先级翻转描述如下：

线程 A,B,C 分别具有 1,2,3 的优先级(数字越大说明优先级越高),线程 A,B 目前在就绪队列中等待调度,线程 A 对一个互斥资源拥有线程锁。而此时,高优先级的线程 C 也想要访问这个互斥资源,线程C只 好在这个资源上等待,不能进入就绪队列。当调度器开始调度时,它只能从 A 和 B 中进行选择,根据优先 级调度原理,线程 B 将会首先运行。 这时就产生了一个问题,即本来线程C的优先级比线程B高,但是线程B却先运行了,从而产生了优先级翻转问题。

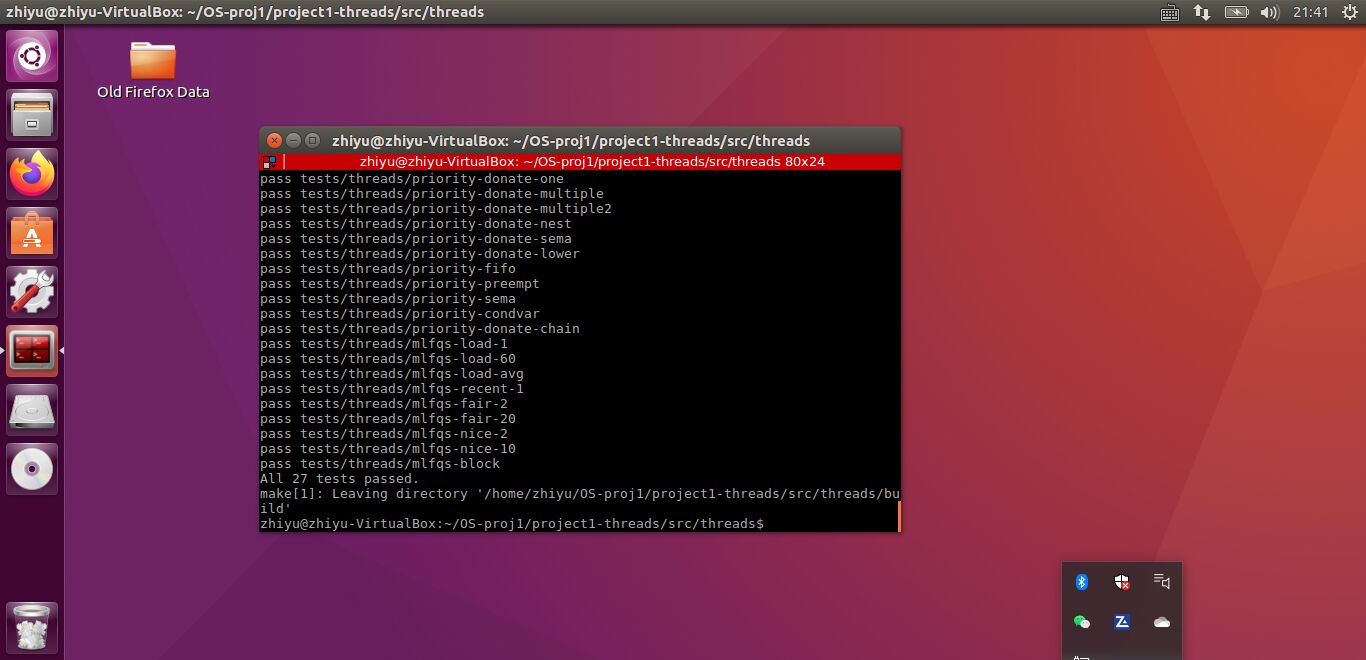
1. 高级调度

普通线程的调度策略是基于时间片轮转的优先级调度，总体来说，实现简单方便，体现了严格按优先级运行的严谨。

这种调度策略有一定的优点，不过，在某些情况下，这种策略存在一定弊端。这种调度策略极易造成线程的饥饿，当有高优先级且所需运行时间片较长的线程被创建，其余的线程只有在它行完之后才能重新获得 CPU，这种调度策略没有考虑到系统的吞吐量，无法减少线程的平均等待时间。这种弊端在大用户 量的交互系统中会被放大，造成用户长时间的等待会降低用户对于该系统的评价。

因此，我们需要一种新的调度策略，该策略在考虑线程优先级的基础之上，能够按照线程运行情况来动 态地考虑修改线程的优先级，于是根据执行情况反馈信息而对线程队列进行组织并调度线程的调度器多 级反馈队列调度器应运而生。

运行截图



+--------------------+

| CS 140 |

| PROJECT 1: THREADS |

| DESIGN DOCUMENT |

+--------------------+

---- GROUP ----

>> Fill in the names and email addresses of your group members.

王诗语 <18373123>

熊雪萤 <18373155>

李智宇 <18373091>

祖泊宁 <18373121>

---- PRELIMINARIES ----

>> If you have any preliminary comments on your submission, notes for the

>> TAs, or extra credit, please give them here.

>> Please cite any offline or online sources you consulted while

>> preparing your submission, other than the Pintos documentation, course

>> text, lecture notes, and course staff.

ALARM CLOCK

===========

---- DATA STRUCTURES ----

>> A1: Copy here the declaration of each new or changed `struct' or

>> `struct' member, global or static variable, `typedef', or

>> enumeration. Identify the purpose of each in 25 words or less.

1. 在threads中增加float.h，包含浮点运算，因为pintos没有浮点运算。
2. 修改synch.h中的struct lock，增加int max\_priority和struct list\_elem element两个属性来进行队列操作，前者用来记录阻塞队列中的最大优先级，后者用来找到链表中的lock结构体
3. 在float.c中增加static int load\_avg，记录当前的load average。
4. 在thread.h中修改struct thread，增加属性int64\_t ticks\_remain（记录剩余的睡眠ticks）、int old\_priority（记录之前的优先级）、struct lock\* waiting lock（线程阻塞的锁）、以及4.4bsd算法中要求的int nice和int recent\_cpu

---- ALGORITHMS ----

>> A2: Briefly describe what happens in a call to timer\_sleep(),

>> including the effects of the timer interrupt handler.

首先，更改线程的状态为blocked来阻塞线程，同时在线程结构体里加一个阻塞时间，每次时钟中断的时候递减。Handler在每个tick时都会对线程剩余的阻塞时间进行检测，当此线程的阻塞时间为0时通知该线程，将其唤醒。

>> A3: What steps are taken to minimize the amount of time spent in

>> the timer interrupt handler?

首先我们编写一个check\_remain()函数来检查一个阻塞线程剩余的阻塞时间，如果时间不为0，那么我们使它的阻塞时间减1，如果为0就将它转为就绪状态。然后通过thread\_foreach()来在每个tick时检查全部线程。

---- SYNCHRONIZATION ----

>> A4: How are race conditions avoided when multiple threads call

>> timer\_sleep() simultaneously?

Timer\_sleep()实现的方法是通过阻塞和唤醒机制，而不是通过yield和循环实现的busy waiting，这样我们可以在timer\_sleep()函数开始时关闭中断，来避免出现race condition的情况。

>> A5: How are race conditions avoided when a timer interrupt occurs

>> during a call to timer\_sleep()?

当一个线程调用timer\_sleep()时，enum intr\_level old\_level = intr\_disable ();这一句中，intr\_disable会将当前线程设置为关中断，同时返回当前状态让old\_level记录，这样可以禁止当前行为被中断。

---- RATIONALE ----

>> A6: Why did you choose this design? In what ways is it superior to

>> another design you considered?

这种设计是通过对线程的阻塞(timer\_sleep)与线程的唤醒(timer\_interrupt)来实现的，与通过循环来实现的的busy wait方法相比，这种设计不需要在等待中不断进行循环和检查，节省了CPU资源。

PRIORITY SCHEDULING

===================

---- DATA STRUCTURES ----

>> B1: Copy here the declaration of each new or changed `struct' or

>> `struct' member, global or static variable, `typedef', or

>> enumeration. Identify the purpose of each in 25 words or less.

答：

1. 修改synch.h文件，为struct lock增加属性：

//用来记录阻塞队列中的最大优先级，是用来捐赠给持有锁的线程的

int max\_priority;

//用来找到链表中的lock结构体

struct list\_elem element;

1. 修改thread.h文件，为struct thread增加属性：

//用来记录之前的优先级

int old\_priority;

//用来记录线程持有的锁

struct list locks;

//用来记录线程阻塞的锁

struct lock\* waiting\_lock;

>> B2: Explain the data structure used to track priority donation.

>> Use ASCII art to diagram a nested donation. (Alternately, submit a

>> .png file.)

（1）

我们为thread添加了属性old\_priority和waiting\_lock，为lock添加了max\_priority和element来帮助跟踪优先级捐赠。

每当一个线程获得一把锁时，锁会被插入到线程的struct list locks字段里。当一把锁被释放时，它就会从持有者的锁列表中删除。lock结构体中的element也是同样用处。

就一次捐赠而言，如果某个线程要获得一把锁，那么会先检查锁持有者的优先级，如果它低于需要锁的这个线程的优先级，捐赠就会发生。

在我们的实现中，线程的old\_priority将随着优先级改变（除了捐赠以外）,所以我们可以假设old\_priority已经保存当前的优先级。

然后将被捐赠线程的优先级设置为捐赠者线程的优先级。

锁的max\_priority被设置为捐赠者的优先级，以跟踪锁的waiter列表中的最高优先级。

捐赠者线程的waiting\_lock被设置为这个锁。

然后对于嵌套捐赠来说，需要检查被捐赠线程是否被另一个锁阻塞。如果是的话，会按上述程序再次进行另一次捐赠。现在的被捐赠线程是新的捐赠者，受捐者是当前线程所阻塞的锁的锁持有者。

嵌套的情况会一直检查下去，直到没有被捐赠线程被其他线程阻塞，或者到了嵌套层数的上限（通过全局变量来规定）。二者出现之一，嵌套捐赠都会停止。

（2）以如下情况为例：

A 线程，优先级31，持有锁lock\_1.

B 线程，优先级32，持有锁lock\_2, 要获取锁lock\_1

C 线程，优先级33，要获取锁lock\_2

第一阶段：开始状态

=========================

.---------------------------------------------------.

| 线程 A (开始状态) |

+-------------------+-------------------------------+

| 成员 | 值 |

+-------------------+-------------------------------+

| priority | 31 |

| old\_priority | 31 |

| locks | {lock\_1 (max\_priority = -1)} |

| waiting\_lock | NULL |

'-------------------+-------------------------------'

.---------------------------------------------------.

| 线程 B (开始状态) |

+-------------------+-------------------------------+

| 成员 | 值 |

+-------------------+-------------------------------+

| priority | 32 |

| old\_priority | 32 |

| locks | {lock\_2 (max\_priority = -1)} |

| waiting\_lock | NULL |

'-------------------+-------------------------------'

.---------------------------.

| 线程 C (开始状态) |

+-------------------+-------+

| 成员 | 值 |

+-------------------+-------+

| priority | 33 |

| old\_priority | 33 |

| locks | {} |

| waiting\_lock | NULL |

'-------------------+-------'

==================================================================

第二阶段：B 获取 lock\_1：

==========================

.---------------------------------------------------.

| 线程 A (B 获取 L1) |

+-------------------+-------------------------------+

| 成员 | 值 |

+-------------------+-------------------------------+

| priority | 31 |

| old\_priority | 32 |

| locks | {lock\_1 (max\_priority = 32)} |

| waiting\_lock | NULL |

'-------------------+-------------------------------'

.---------------------------------------------------.

| 线程 B (B 获取 L1) |

+-------------------+-------------------------------+

| 成员 | 值 |

+-------------------+-------------------------------+

| priority | 32 |

| old\_priority | 32 |

| locks | {lock\_2 (max\_priority = -1)} |

| waiting\_lock | &lock1 |

'-------------------+-------------------------------'

.---------------------------.

| 线程 C (B 获取 L1) |

+-------------------+-------+

| 成员 | 值 |

+-------------------+-------+

| priority | 33 |

| old\_priority | 33 |

| locks | {} |

| waiting\_lock | NULL |

'-------------------+-------'

==================================================================

第3阶段（1）：C 获取 lock\_2：

============================

.---------------------------------------------------.

| 线程 B (C 获取 L2, Step 1) |

+-------------------+-------------------------------+

| 成员 | 值 |

+-------------------+-------------------------------+

| priority | 32 |

| old\_priority | 33 |

| locks | {lock\_2 (max\_priority = 33)} |

| waiting\_lock | &lock1 |

'-------------------+-------------------------------'

.----------------------------------.

| 线程 C (C 获取 L2, Step 1) |

+----------------------+-----------+

| 成员 | 值 |

+----------------------+-----------+

| priority | 33 |

| old\_priority | 33 |

| locks | {} |

| waiting\_lock | &lock\_2 |

'----------------------+-----------'

.---------------------------------------------------.

| 线程 A (C 获取 L2, Step 1) |

+-------------------+-------------------------------+

| 成员 | 值 |

+-------------------+-------------------------------+

| priority | 31 |

| old\_priority | 32 |

| locks | {lock\_1 (max\_priority = 32)} |

| waiting\_lock | NULL |

'-------------------+-------------------------------'

==================================================================

第3阶段（2）：C 获取 lock\_2：

============================

.---------------------------------------------------.

| 线程 B (C 获取 L2, Step 2) |

+-------------------+-------------------------------+

| 成员 | 值 |

+-------------------+-------------------------------+

| priority | 32 |

| old\_priority | 33 |

| locks | {lock\_2 (max\_priority = 33)} |

| waiting\_lock | &lock1 |

'-------------------+-------------------------------'

.----------------------------------.

| 线程 C (C 获取 L2, Step 2) |

+----------------------+-----------+

| 成员 | 值 |

+----------------------+-----------+

| priority | 33 |

| old\_priority | 33 |

| locks | {} |

| waiting\_lock | &lock\_2 |

'----------------------+-----------'

.---------------------------------------------------.

| 线程 A (C 获取 L2, Step 2) |

+-------------------+-------------------------------+

| 成员 | 值 |

+-------------------+-------------------------------+

| priority | 31 |

| old\_priority | 33 |

| locks | {lock\_1 (max\_priority = 32)} |

| waiting\_lock | NULL |

'-------------------+-------------------------------'

==================================================================

第4阶段：A 释放 lock\_1：

==========================

.-------------------------------.

| 线程 A (A 释放 lock\_1)) |

+---------------------+---------+

| 成员 | 值 |

+---------------------+---------+

| priority | 31 |

| old\_priority | 31 |

| locks | {} |

| waiting\_lock | NULL |

'---------------------+---------'

.----------------------------------------------------.

| 线程 B (A 释放 lock\_1) |

+-------------------+--------------------------------+

| 成员 | 值 |

+-------------------+--------------------------------+

| priority | 32 |

| old\_priority | 33 |

| locks | {&lock\_2 (max\_priority = 33), |

| | &lock\_1 (max\_priority = 32)} |

| waiting\_lock | NULL |

'-------------------+--------------------------------'

.------------------------------.

| 线程 C (A 释放 lock\_1) |

+--------------------+---------+

| 成员 | 值 |

+--------------------+---------+

| priority | 33 |

| old\_priority | 33 |

| locks | {} |

| waiting\_lock | &lock\_2 |

'--------------------+---------'

==================================================================

第5阶段：B 释放 lock\_2：

==========================

.-------------------------------.

| 线程 A (B 释放 lock\_2)) |

+---------------------+---------+

| 成员 | 值 |

+---------------------+---------+

| priority | 31 |

| oldpriority | 31 |

| locks | {} |

| waiting\_lock | NULL |

'---------------------+---------'

.----------------------------------------------------.

| 线程 B (B 释放 lock\_2) |

+-------------------+--------------------------------+

| 成员 | 值 |

+-------------------+--------------------------------+

| priority | 32 |

| old\_priority | 32 |

| locks | {&lock\_1 (max\_priority = 32)} |

| waiting\_lock | NULL |

'-------------------+--------------------------------'

.----------------------------------------------------.

| 线程 C (B 释放 lock\_2) |

+-------------------+--------------------------------+

| 成员 | 值 |

+-------------------+--------------------------------+

| priority | 33 |

| old\_priority | 33 |

| locks | {&lock\_2 (max\_priority = 33)} |

| waiting\_lock | NULL |

'-------------------+--------------------------------'

---- ALGORITHMS ----

>> B3: How do you ensure that the highest priority thread waiting for

>> a lock, semaphore, or condition variable wakes up first?

答：将struct semaphore和struct condition中的属性waiters（正在等待的线程队列）改为按照优先级排序的队列。每次唤醒线程时，线程按照在该队列中的顺序依次放到ready\_list中，也就是优先级从最高到最低。这样保证了ready\_list中拥有最高优先级的线程是第一个等待被唤醒的。

>> B4: Describe the sequence of events when a call to lock\_acquire()

>> causes a priority donation. How is nested donation handled?

答：

第一步：屏蔽中断

第二步：捐赠

2.1 如果lock的holder为NULL

2.1.1 sema\_down：如果信号量值为0，把所有需要这把锁的线程放到该信号量的等待队列里，直到信号量值变为正值。

2.1.2 设置当前线程为这个锁的持有者

2.2 否则比较lock的holder(L)的优先级和当前线程(C)的优先级

2.2.1如果线程L的优先级大于线程C的优先级

2.2.1.1 执行sema\_down直到信号量值变为正值，也就是直到锁释放

2.2.1.2 设置当前线程为这个锁的持有者

2.2.2 否则

2.2.2.1 [捐赠]将线程L的优先级赋值给线程C的优先级

2.2.2.2 执行sema\_down，直到锁被释放

2.2.2.3 当前线程成为这个锁的持有者

第三步：把中断恢复到被禁用之前的状态

>> B5: Describe the sequence of events when lock\_release() is called

>> on a lock that a higher-priority thread is waiting for.

答：

①如果该锁不对应这个线程则报错；

②解锁，禁用中断，将锁的持有者设置为NULL

③设置锁原持有者的优先级。如果没有捐赠或者有捐赠但是原线程仅持有该锁，则把优先级还原；否则进行嵌套捐赠，把原线程的优先级设置为锁列表中线程的最高级.

④将信号量加一，thread\_yield()让列表中线程可以竞争，取就绪列表上的最高优先级线程。

---- SYNCHRONIZATION ----

>> B6: Describe a potential race in thread\_set\_priority() and explain

>> how your implementation avoids it. Can you use a lock to avoid

>> this race?

答：

潜在的竞争：更高级线程捐赠优先级和被捐赠线程本身改变优先级，可能会发生竞争。不同的改变顺序最后可能会导致不同的优先级。

避免算法：thread\_set\_priority()检查 lock 是否有更高优先级的线程在等待

，我们的代码通过禁用中断来防止该竞争。申请的锁被记下，当另一个线程申请这个锁时，检测它的优先级，判断是否进行捐赠，然后放入等待队列被阻塞。如果调用thread\_set\_priority()设置有限制是小于现优先级或者当前线程锁列表不为空，则不更新。

---- RATIONALE ----

>> B7: Why did you choose this design? In what ways is it superior to

>> another design you considered?

答：这样的设计没有过多改变struct thread结构和struct锁结构，不会造成代码复杂。struct thread结构里记录的old\_priority能使释放锁后的线程恢复原来的优先级，更加方便，节省了CPU资源。struct lock \*waiting lock记录了线程阻塞的锁。而使用max\_priority来记录阻塞队列中的最大优先级，使得捐赠行为发生时，改变线程的原优先级，有一定的效率提升。

ADVANCED SCHEDULER

==================

---- DATA STRUCTURES ----

>> C1: Copy here the declaration of each new or changed `struct' or

>> `struct' member, global or static variable, `typedef', or

>> enumeration. Identify the purpose of each in 25 words or less.

答：

- 由于pintos没有浮点运算，所以在threads包里增加float.h表示浮点运算。

- 修改thread.h中的struct thread，增加int old\_priority表示原优先级，struct list locks表示线程持有的锁，struct lock\* waiting\_lock表示线程阻塞的锁，以及根据4.4BSD scheduler设置的int nice和int recent\_cpu。

- 在thread.c中增加全局变量static load\_avg，记录当前的load average。

- 修改synch.h中的struct lock、增加int max\_priority和struct element两个属性来进行队列操作，前者用来记录阻塞队列中的最大优先级，后者用来找到链表中的lock结构体

---- ALGORITHMS ----

>> C2: Suppose threads A, B, and C have nice values 0, 1, and 2. Each

>> has a recent\_cpu value of 0. Fill in the table below showing the

>> scheduling decision and the priority and recent\_cpu values for each

>> thread after each given number of timer ticks:

timer recent\_cpu priority thread

ticks A B C A B C to run

----- -- -- -- -- -- -- ------

0 0 1 2 63 61 59 A

4 4 1 2 62 61 59 A

8 8 1 2 61 61 59 B

12 8 5 2 61 61 59 A

16 12 5 2 60 60 59 B

20 12 9 2 60 59 59 A

24 16 9 2 59 59 59 C

28 16 9 6 59 59 58 B

32 16 13 2 59 58 58 A

36 20 13 6 58 58 58 C

>> C3: Did any ambiguities in the scheduler specification make values

>> in the table uncertain? If so, what rule did you use to resolve

>> them? Does this match the behavior of your scheduler?

recent\_cpu是含糊的，当计算recent\_cpu的时候，需要考虑CPU每4个tick中花在计算上的时间，例如load\_avg，每个线程的recent\_cpu，all\_list中所有线程的优先级，和ready\_list。当CPU进行上述计算时，当前线程需要为之让路，无法运行。因此，每4个tick里面，真正加在recent\_cpu上的是少于4个的。但是我们无法确定这些计算花了多长时间。

因此，每4个tick，我们在recnt\_cpu上加4。

>> C4: How is the way you divided the cost of scheduling between code

>> inside and outside interrupt context likely to affect performance?

如果CPU在计算recent\_cpu、load\_avg和优先级上花费了太多的时间，这会占用抢占前线程的大部分时间。这样，这个线程就不能像预期的那样获得足够的运行时间，从而运行得更久。这样就会占用更多的CPU时间，并提高了它的load\_avg和recent\_cpu，从而降低了它的优先级。这可能会影响调度决策的制定。因此，如果在中断上下文内进行调度的成本上升，会降低性能。

---- RATIONALE ----

>> C5: Briefly critique your design, pointing out advantages and

>> disadvantages in your design choices. If you were to have extra

>> time to work on this part of the project, how might you choose to

>> refine or improve your design?

我们使用了ready\_list。每次开始的时候ready\_list中都会进行优先级降序排列，即每当我们向ready\_list插入一个线程时，都会按顺序插入。这部分的时间复杂度为O(n)。每隔四个tick，就需要计算all\_list中所有线程的优先级。在这之后，我们需要对ready\_list进行排序，这将花费O(nlgn)的时间。由于我们每4个tick都需要执行此任务，因此它将使线程运行的tick比预期的要短。如果n变大，线程切换可能会经常发生。

如果有更多的时间，我们会采用更多的队列，将他们放入一个索引等于其优先级值的数组中。当线程第一次被插入时，它只需要根据线程的优先级索引队列。这将只需要O(1)时间。运行速度会更快。

>> C6: The assignment explains arithmetic for fixed-point math in

>> detail, but it leaves it open to you to implement it. Why did you

>> decide to implement it the way you did? If you created an

>> abstraction layer for fixed-point math, that is, an abstract data

>> type and/or a set of functions or macros to manipulate fixed-point

>> numbers, why did you do so? If not, why not?

recent\_cpu和load\_avg是实数，但是pintos禁用了浮点数。因此使用定点数代替浮点数来表示recent\_cpu和load\_avg。

在float.h中#define CONVERT\_M\_TO\_FIXED\_POINT(n)将浮点数转化为定点数。

在thread.c中可以直接调用这个宏进行转换，提高运行速度，更加方便。

SURVEY QUESTIONS

================

Answering these questions is optional, but it will help us improve the

course in future quarters. Feel free to tell us anything you

want--these questions are just to spur your thoughts. You may also

choose to respond anonymously in the course evaluations at the end of

the quarter.

>> In your opinion, was this assignment, or any one of the three problems

>> in it, too easy or too hard? Did it take too long or too little time?

>> Did you find that working on a particular part of the assignment gave

>> you greater insight into some aspect of OS design?

>> Is there some particular fact or hint we should give students in

>> future quarters to help them solve the problems? Conversely, did you

>> find any of our guidance to be misleading?

>> Do you have any suggestions for the TAs to more effectively assist

>> students, either for future quarters or the remaining projects?

>> Any other comments?