**西安电子科技大学**

**操作系统课程设计**

**(2021年度)**

**实**

**验**

**报**

**告**

**实验名称：** Add Priority

**班 级：** 1903015

**姓 名：** 吕思勤

**学 号：** 19030130448

**一、实验内容**

原始Pintos系统中对于线程的调度，没有考虑优先级问题，采用的是最为简单的FCFS策略。本实验要求为Pintos建立优先级调度机制，并确保任何时刻CPU上运行的都是最高优先级线程。

**二、分析与设计**

1. Pintos中线程优先级分为64级(PRI\_MIN (0) ~ PRI\_MAX (63))，默认优先级为31(PRI\_DEFAULT (31))。原始pintos代码已在thread.h中定义了线程的优先级属性，如下：因此你不需要再为线程定义优先级属性，直接使用即可。

2.线程在生成时，可通过init\_thread()函数（如下）得到一个优先级，我们称为初始优先级。struct thread{...int priority /\*Priority\*/...}在线程执行的过程中还可通过thread\_set\_priority()函数修改优先级(in thread.c)。原始代码已经为我们提供了thread\_set\_priority()函数的原型，但是这个函数完成的功能只是修改当前运行线程的优先级数值(如下)，而没有真正实现优先级调度的功能。

3. 导致线程优先级发生变化的原因有很多，这与具体的调度策略有关，这里我们暂时不去考虑，只知道线程优先级可能会因某些原因突然发生改变，并且在改变后系统要立刻做出响应，保证高优先级线程总是能够顺利运行就行。

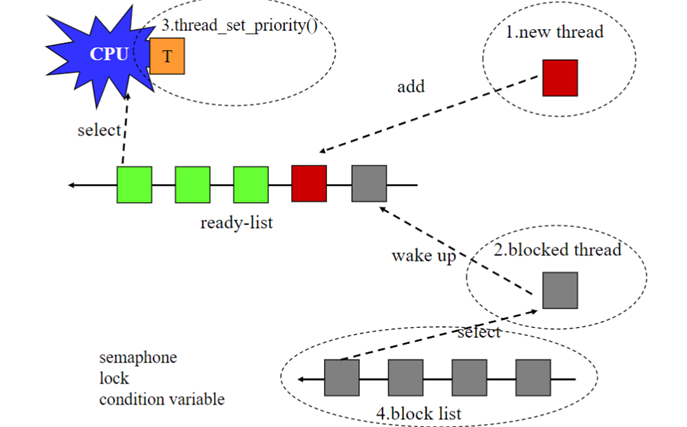
4.本实验中哪些情况下需要考虑优先级调度？

分以下几种：（1）当一个线程T正在CPU上运行，突然有一个更高优先级的线程H进入ready-list中，则Tstatic void init\_thread (struct thread \*, const char \*name, int priority);//in thread.c/\* Sets the current thread's priority to NEW\_PRIORITY. \*/voidthread\_set\_priority (int new\_priority) {thread\_current ()->priority = new\_priority;}/\* Returns the current thread's priority. \*/intthread\_get\_priority (void) {return thread\_current ()->priority;}

必须立刻让出CPU，让H运行。当然，突然出现一个更高优先级线程的原因可能如下：一是新线程生成；二是一个线程被唤醒进入ready-list；另一个原因见（3）。新线程的创建是由thread\_create()函数(in thread.c)完成的。可在线程的创建过程结束后，加入一个比较函数：如果这个新创建的线程比当前正在运行的线程优先级还高，则使正在运行的线程让出CPU，这一过程可以用thread\_yield()函数实现。但是源代码中给出的thread\_yield()函数，没有提供按优先级排序的功能，只是简单地把当前正在运行的线程“push back”到ready-list中。因此，可能需要在此加入优先级排序功能。源代码中已经为我们提供了一种链表排序方法——插入排序，请参考lib/kernel/list.c中的list\_insert\_ordered()函数，你可能会经常调用这个函数，原型如下：voidlist\_insert\_ordered (struct list \*list, struct list\_elem \*elem,list\_less\_func \*less, void \*aux)此外，thread\_unblock()函数的功能是把一个线程由blocked状态标记为ready状态，此时如果被唤醒线程的优先级较高，则当前运行线程也必须立刻放弃CPU，故thread\_unblock()函数也需进行同样的修改（有序化插入ready-list）。

（2）当一个线程T正在运行，如果他的优先级被突然降低，以至于比ready-list中的某些线程优先级还低，则此时T必须立刻让出CPU；线程优先级的改变是通过thread\_set\_priority()函数进行的，可能有以下两种情况：1）修改后该线程的优先级已经低于ready-list中某些线程的优先级，此时当前正在运行的线程应立刻让出CPU；2）修改后该线程的优先级高于ready\_list中所有线程的优先级，则该线程应继续运行。因此，我们需要据此修改thread\_set\_priority()函数。一点注意事项：在采用插入排序方式处理ready-list后，链表中的线程已按优先级有序排雷，位于队头的线程，优先级一定最高。因此，在上面两种情况中，只需把当前正在运行的线程的优先级和ready-list队头线程的优先级比较就可以了。

（3）如果有多个线程因为等待一把锁、一个信号量或者一个条件变量而被阻塞，当条件满足时，拥有最高优先级的线程应被优先唤醒进入ready-list。当其进入ready-list时，会引发（1）中情况发生。此处的任务是要保证在条件满足时，被唤醒的总是最高优先级线程。涉及到的函数有(in devices/synch.c)：sema\_down(), sema\_up( ), cond\_wait()。lsema\_down()函数：对一个信号量进行“P”操作，等待信号量的值变为正数并自动减1。可以看出该函数在检测到信号量值为0时会将当前线程阻塞，并放到该信号量的阻塞队列。为了保证该信号量>0时，被唤醒的是最高优先级线程，可将sema-waiter list也按优先级有序化处理。lsema\_up()函数(如下图)：实现了V操作，它将信号量的值加1，并唤醒一个之前因等待此信号量而被阻塞的线程。当然如果此时没有任何线程在等待该信号量，则没必要做唤醒操作，但是，如果同时有多个线程在等待该信号量，这时就应唤醒优先级最高的线程。同时，如果新唤醒的线程的优先级高于运行中线程的优先级，那么运行中线程应当立刻让出CPU。因此在此函数的末尾也需要进行一次优先级比较。cond\_wait()函数。当一个线程获得管程的锁时，如果条件变量（condition variable）不满足，这个线程无法运行。于是，为了满足其他线程的运行需要，线程需先将这个锁释放，等待条件变量满足以后再重新获取锁。但是一个条件变量可能正在同时被多个线程等待，所以你需要对这些线程也做好优先级排序。当条件变量满足运行要求后，优先级最高的线程应该优先获得锁。条件变量的定义为：struct condition {struct list waiters; /\* List of waiting threads. \*/};可见所谓条件变量实质上是一个list，里面记录着等待该条件变量的线程，我们应当使其按优先级有序排列。



**三、详细实现**

thread.c

thread\_creat()函数

一个新的线程创立，如果新的线程的优先级大于正在运行的线程的优先级，那么正在运行的线程让出CPU。

在return tid; 前添加：

if (priority > thread\_current ()->priority) thread\_yield();

thread\_yield()函数

当一个线程让出CPU之后，将此线程放入ready\_list中，插入的方法是按照优先级从大到小插入，而不是放在队尾。即

/\*list\_push\_back (&ready\_list, &cur->elem);\*/（原程序）

list\_insert\_ordered (&ready\_list, &cur->elem, pri\_more, NULL);（新的改法）

thread\_unblock()函数

将一个线程从block转换为ready态，此线程插入ready\_list仍然应按照优先级排序。即：/\*list\_push\_back (&ready\_list, &t->elem); \*/（原程序）list\_insert\_ordered(&ready\_list, &t->elem, pri\_more, NULL);（新的改法）

thread\_set\_priority()函数

如果一个线程（正在运行）被赋予新的优先级，那么此线程的优先级如果比ready\_list中优先级最高的线程的优先级低，那么此线程将让出CPU，添加代码：if(list\_entry(list\_begin(&ready\_list),struct thread,elem)->priority >=new\_priority) thread\_yield();

自定义函数：

bool

pri\_more(const struct list\_elem \*a,

const struct list\_elem \*b,void \*aux) //作用对象：线程

{

struct thread \*a\_thread,\*b\_thread;

a\_thread=list\_entry(a,struct thread, elem);

b\_thread=list\_entry(b,struct thread, elem);

return (a\_thread->priority > b\_thread->priority);

}

synch.c

sema\_up()函数

一个信号量可能会阻塞多个线程，应将优先级最高的线程从block态转换为ready态。

加入定义：

struct thread \*t=NULL;

修改if (!list\_empty (&sema->waiters)) 条件分支中：

/\* thread\_unblock (list\_entry (list\_pop\_front (&sema->waiters), struct thread, elem)); \*/（原程序）

{

t = list\_entry (list\_pop\_front (&sema->waiters), struct thread, elem); thread\_unblock (t);

}

在sema->value++; 后加入：

if(t != NULL && t->priority > thread\_current()->priority) thread\_yield();

sema\_down()函数

应有序化sema->waiters队列：

//list\_push\_back (&sema->waiters, &thread\_current ()->elem) (原程序) list\_insert\_ordered(&sema->waiters, &thread\_current ()->elem, pri\_more, NULL); (新方法)

cond\_wait()函数。在struct semaphore\_elem 中加入新属性：

int sema\_priority;

替换： /\* list\_push\_back (&cond->waiters, &waiter.elem); \*/ (原代码)

waiter.sema\_priority = thread\_current ()->priority;

list\_insert\_ordered (&cond->waiters, &waiter.elem, cond\_priority, NULL);

自定义函数：

bool

cond\_priority (const struct list\_elem \*lhs, const struct list\_elem \*rhs, void \*aux UNUSED) //功能与pri\_more类似，作用对象：semaphore\_elem

{

struct semaphore\_elem \*l, \*r;

l = list\_entry (lhs, struct semaphore\_elem, elem);

r = list\_entry (rhs, struct semaphore\_elem, elem);

return (l->sema\_priority > r->sema\_priority);

}

**四、实验结果**

成功完成了这些任务， alarm\_priority, priority-change, priority-fifo, priority-preempt, priority-sema, priority-condvar等6个检测通过。

一張含有 文字 的圖片

自動產生的描述

**五、心得体会**

通过本次实验，再次回顾了上个学期在操作系统理论课程中所学习到的线程优先级的基本概念，这次实验通过alarm clock实际例子，在上一个实验的基础上，修改部份代码，为原本的Pintos系统建立优先级调度机制，并确保任何时刻CPU上运行的都是最高优先级线程。实验过程中，我逐渐理解程序中一些关键函数的调用方式与线程优先级调度的工作原理，从中分析并加以修改，过程中也因为在函数声明的顺序错误导致make check时报错，但在不断地尝试与分析下，最终成功通过调试，完成了本次实验，在实验的过程中阅读代码让我对于操作系统的概念掌握的更加彻底，做完本次实验也让我获得满满的成就感。