## 一、实验内容

现代操作系统，并发是基本的特征。当有多个线程在系统中运行时，一般是通过排队方式采用RR策略（时间片轮转法）轮流占用CPU的。其中最简单、看似最公平的调度策略是FCFS（先来先服务），但是这种方法不能够区分线程执行的轻重缓急，看似公平实则不公平，容易造成系统性能的低下。因此，操作系统设计者通常会赋予线程优先级，并按照优先级高度进行调度，以兼顾公平和效率。

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

### Pintos 中线程优先级的定义

按照老师的提示，阅读 thread.h ，发现 Pintos 中已经有了关于线程优先级的定义，在 thread 结构体中，有一个 int 类型的 priotity 字段，因此，可以直接根据该字段的属性值，实现 Pintos 的线程优先级调度。

观察 next\_thread\_to\_run() 这个函数体，可以发现，CPU 将要运行的下一个线程，永远是 ready\_list 中头部的进程

static struct thread \*  
next\_thread\_to\_run (void)   
{  
 if (list\_empty (&ready\_list))  
 return idle\_thread;  
 else  
 return list\_entry (list\_pop\_front (&ready\_list), struct thread, elem);  
}

因此，只需要保证 ready\_list 是按照优先级有序的，就可以实现 Pintos 按照优先级对进程进行调度。而保证 ready\_list 的有序，需要修改所有对 ready\_list 进行了修改操作的函数，保证这些函数在修改 ready\_list 时，维护 ready\_list 的有序性。

### 需要修改的调度

按照提示，我们需要修改的位置有以下这些：

* 线程的创建：init\_thread() 、 thread\_create()
* 进程出让cpu ：thread\_yield()
* **线程被唤醒进入 ready\_list**:
  + thread\_unblock()
* 线程的优先级被降低 ：thread\_set\_priority()

不单单是线程的执行需要按优先级调度，资源的释放与分配同样应该按照优先级进行调度，因此，还需要修改：

* sema\_down() 线程进入信号量阻塞队列
* sema\_up() 线程获取信号量后需要与当前正在运行的程序优先级进行比较

除信号量外，Pintos 中还有等待条件变量的队列，这个队列也同样需要按优先级有序化。

* cond\_wait() 线程进入条件等待队列

## 三、具体实现

### ready\_list 队列有序化

* thread\_init 函数
* 阅读源代码，可以发现，Pintos 中 ready\_list 就绪队列是靠 Pintos 的 list 数据结构实现的，list 中已经有元素有序插入队列的方法，函数原型为
* void list\_insert\_ordered (struct list \*, struct list\_elem \*,  
   list\_less\_func \*, void \*aux);
* 其中，第三个参数是列表元素实现比较的函数指针，若要实现列表按照进程优先级排序，我们首先需要自己实现这个比较函数
* bool thread\_priority\_cmp(const struct list\_elem \*a, const struct list\_elem \*b, void \*aux){  
   return list\_entry(a,struct thread, allelem)->priority > list\_entry(b,struct thread, allelem)->priority;  
  }
* 其中，list\_entry 函数从列表中取出进程元素，通过 priority 字段的比较，将 list 从大到小排列。
* 修改 init\_thread 函数，在其中按照 list\_insert\_ordered 方法，将创建的进程有序插入。
* list\_insert\_ordered(&all\_list, &t->allelem, (list\_less\_func \*) &thread\_priority\_cmp, NULL);
* thread\_create()
* 进程创建完成后，还需要对当前正在运行进程的优先级和新创建的进程进行比较，如果行创建的线程优先级高，当前运行的线程就需要释放 CPU 资源。需要在 thread\_create 底部加入如下代码
* if(thread\_current()->priority < priority){  
   thread\_yield();  
   }
* thread\_yield:
* 原始 thread\_yield 也仅仅是把当前运行的线程置入就绪队列尾部，因此也需要修改：
* void  
  thread\_yield (void)   
  {  
   struct thread \*cur = thread\_current ();  
   enum intr\_level old\_level;  
     
   ASSERT (!intr\_context ());  
    
   old\_level = intr\_disable ();  
   if (cur != idle\_thread)   
   list\_insert\_ordered(&ready\_list, &cur->elem, (list\_less\_func \*) &thread\_priority\_cmp, NULL);  
   cur->status = THREAD\_READY;  
   schedule ();  
   intr\_set\_level (old\_level);  
  }
* thread\_unblock
* 同样替换掉 thread\_unblock 函数中原始的插入函数
* void  
  thread\_unblock (struct thread \*t)   
  {  
   enum intr\_level old\_level;  
    
   ASSERT (is\_thread (t));  
    
   old\_level = intr\_disable ();  
   ASSERT (t->status == THREAD\_BLOCKED);  
   list\_insert\_ordered(&all\_list, &t->allelem, (list\_less\_func \*) &thread\_priority\_cmp, NULL);  
   t->status = THREAD\_READY;  
   intr\_set\_level (old\_level);  
  }
* thread\_set\_priority()
* 与创建进程类似，重新设定优先级之后，需要将当前运行的线程优先级和目前优先级最高的线程优先级比较 （队首元素），判断是否需要释放 CPU
* void  
  thread\_set\_priority (int new\_priority)   
  {  
   struct thread \*cur = thread\_current ();  
   thread\_current ()->priority = new\_priority;  
   if (cur->priority < list\_entry(list\_front(&ready\_list), struct thread, elem)->priority){  
   thread\_yield();  
   }  
  }

### 信号量队列有序化

与就绪队列不同，首先需要为信号量队列引入优先级

/\* One semaphore in a list. \*/  
struct semaphore\_elem   
 {  
 struct list\_elem elem; /\* List element. \*/  
 struct semaphore semaphore; /\* This semaphore. \*/  
 int sema\_priority; // 引入优先级  
 };

同样，要实现优先级的比较函数，为之后的队列有序插入做准备

bool cond\_priority\_cmp(const struct list\_elem \*a, const struct list\_elem \*b, void \*aux){  
 return list\_entry(a,struct semaphore\_elem, elem)->sema\_priority > list\_entry(b,struct semaphore\_elem, elem)->sema\_priority;  
}

* sema\_down
* 当信号量sema 值等于 0 时，将进行 P 操作的线程有序置入 sema 的等待队列
* while (sema->value == 0)   
   {  
   list\_insert\_ordered(&sema->waiters, &thread\_current()->elem, (list\_less\_func \*) &thread\_priority\_cmp, NULL);  
   //list\_push\_back (&sema->waiters, &thread\_current ()->elem);  
   thread\_block ();  
   }
* sema\_up
* 需要在 thread\_unblock 之后，将就绪的进程和当前进程进行比较，如果当前进程的优先级低，应该调用thread\_yield() 函数释放 CPU
* void  
  sema\_up (struct semaphore \*sema)   
  {  
   enum intr\_level old\_level;  
    
   ASSERT (sema != NULL);  
    
   old\_level = intr\_disable ();  
   if (!list\_empty (&sema->waiters))   
   {  
   struct thread \*to\_unblock = list\_entry (list\_pop\_front (&sema->waiters),  
   struct thread, elem);  
   thread\_unblock (to\_unblock);  
   if (thread\_current()->priority < to\_unblock->priority){  
   thread\_yield();  
   }  
   }  
     
   sema->value++;  
   intr\_set\_level (old\_level);  
  }

### cond 序列有序化

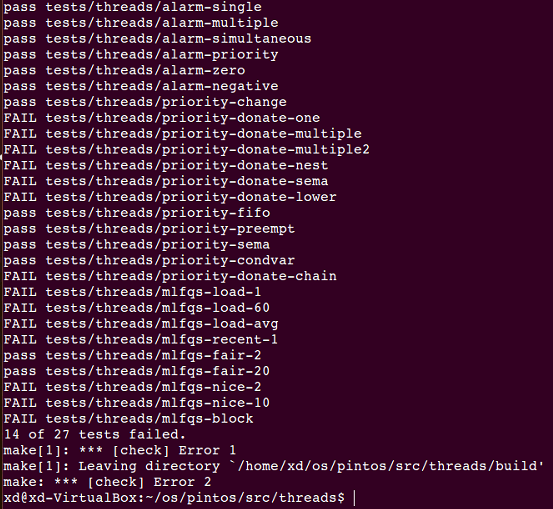
* cond\_wait() 函数，使用 list\_insert\_ordered 替代 list\_push\_back, 实现序列有序化
* list\_insert\_ordered(&cond->waiters, &waiter.elem, (list\_less\_func \*) cond\_priority\_cmp, NULL);

## 四、实验结果

完成修改后，进行 make check 测试发现，在执行信号量相关的测试时，程序处于僵死状态，仔细检查，是sema\_up 函数中 thread\_yield 调用的位置有误，应该在开中断之后调用thread\_yield， 故需要修改成如下：

void
  
 sema\_up (struct semaphore \*sema)
  
 {
  
 enum intr\_level old\_level;
  
   
 ASSERT (sema != NULL);
  
   
 old\_level = intr\_disable ();
  
 struct thread \*to\_unblock;
  
 bool should\_yield = false;
  
 if (!list\_empty (&sema->waiters))
  
 {
  
 to\_unblock = list\_entry (list\_pop\_front (&sema->waiters),
  
 struct thread, elem);
  
 thread\_unblock (to\_unblock);
  
 should\_yield = thread\_current()->priority < to\_unblock->priority;
  
 }
  
 sema->value++;
  
 intr\_set\_level (old\_level);
  
 if (should\_yield){
  
 thread\_yield();
  
 }
  
 }

完成上述修改后，对 Pintos 源码进行重新 make check, 可以看到，通过了和线程优先级相关的以下测试：



## 实验心得：

本次实验的核心内容实质上很简单，由于 Pintos 已经为我们实现了 list 数据结构，我们只需要通过调用list\_insert\_ordered 函数，维护队列的有序性，这样，从队首取出的线程即为当前优先级最高的线程。但是，因为进程队列在多处均有改动，这些函数彼此之间也紧密联系，还需要考虑进程新建等特殊情况，实际上修改这些函数也不像看上去那么简单，最终的工作量较大，需要对 Pintos 线程概念，以及对 Pintos 线程调度过程的细致理解，再加上耐心，才能实现本次的实验。