

Project3 preemptive_scheduler 设计文档

中国科学院大学

孔静，崔鹏来

2016.10.29

1. 抢占调度实现设计流程

1.1 概述

这次试验相当于是上次实验的延续，我们要在上次实验的基础上实现时钟中断的中断处理还有进程睡眠时的调度。因为本次实验添加了中断这一不确定因素，所以选择合适的时间开闭中断是本次实验的一大难点。

（注：我们两个分别合作的方式是先讨论，接着分别写代码，然后遇到问题一起讨论互相问，最后代码综合一下交，报告一个人先写好另一个人改一下，最后统一看一下提交。鉴于这次要期中考试再加上是最后一次合作，所以就不综合了，反正两份都能跑，以下褐色部分由崔鹏来同学写的，对应代码是 part1，绿色部分由我即孔静写的，对应代码是 part2。这次是崔鹏来同学写报告，我上改，所以我把代码与他不同的地方加以说明，其他略去。）

1.2 队列和中断

在本次实验中，主要是使用了 `ready_queue` 和 `sleep_wait_queue` 这两个数据结构。队列的初始化在 `kernel` 中已经做好。Ready 队列中放入了要执行的所有任务，而 `sleep` 队列一开始是空的。我们这次要围绕这两个队列实现 `check_sleep`，`put_current_running`，`do_sleep` 的操作。而中断的操作则需要在 `entry.s` 中完成。

1.3 中断

中断是这次试验最艰难的部分了，当我们接收到时间中断时，我们的程序会自动跳转

到时间中断处理程序 `irq0_entry`。这一部分老师已经写好。接下来我们需要做的是进入临界区，修改 `time_elapsed`，判断当前程序的类型。如果当前任务是进程就进入核心区，运行 `put_current_running` 和 `scheduler`；如果当前程序是线程，就 `cheeck_sleep` 一下，然后继续进行。

这里需要注意几个问题：

1.不需要再增加对 `disable_count` 的修改，因为 `ENTER_CRITICAL` 里面已经有对 `disable_count` 的修改了。

2.如果是进程，不需要在中断处理函数里 `check_sleep`，因为 `scheduler` 函数一开始有 `check_sleep` 的操作。（我看到注释 `Do we want to wake up sleeping processes here as well?`，就在 `put_current_running` 写了 `check_sleep`，看了队友的报告才意识到，我写重复了，不过不想改了。）

3.尽量不使用带%的寄存器，我曾经使用了 `%edi` 然后遇到了各种问题。（利用栈 `pop` 和 `push`，还有题目中提供的 `scratch`，可以帮助我们存储一些数据。）

4.`Time_elapsed` 是 64 位数，不能直接用 `add` 直接加 1，否则会出 bug。（`addl` 和 `adcl` 处理多字节加法。）

1.4 `put_current_running`

我们这次实验要实现一个简单的时间片轮转，当进程被时间中断打断时，我们应该终止进程的运行，保存当前任务，并且完成将要执行任务的替换。`put_current_running` 所做的就是保存当前任务的工作。

所以 `put_current_running` 中主要是进行了如下操作：

```
enqueue(&ready_queue, (node_t *)current_running);
```

1.5 `do_sleep`

当一个任务 `do_sleep` 时我们要首先把他放进 `sleep_wait_queue` 队列里面 ,然后调用 `scheduler` ,完成任务的轮转。

1.6 check_sleep

在本次实验的三个函数里 , `check_sleep` 相对而言麻烦一点。在 `check_sleep` 中我们要遍历的 `sleep_queue` 中的所有任务 将超过任务运行时间的任务放置到 `ready` 队列里面。这里我们需要注意一个问题 , 就是如果 `ready` 队列为空 , 有没有任务可以被叫醒会发生什么。答案是会不断的 `check_sleep` , 这个工作是在 `scheduler` 里面做的。

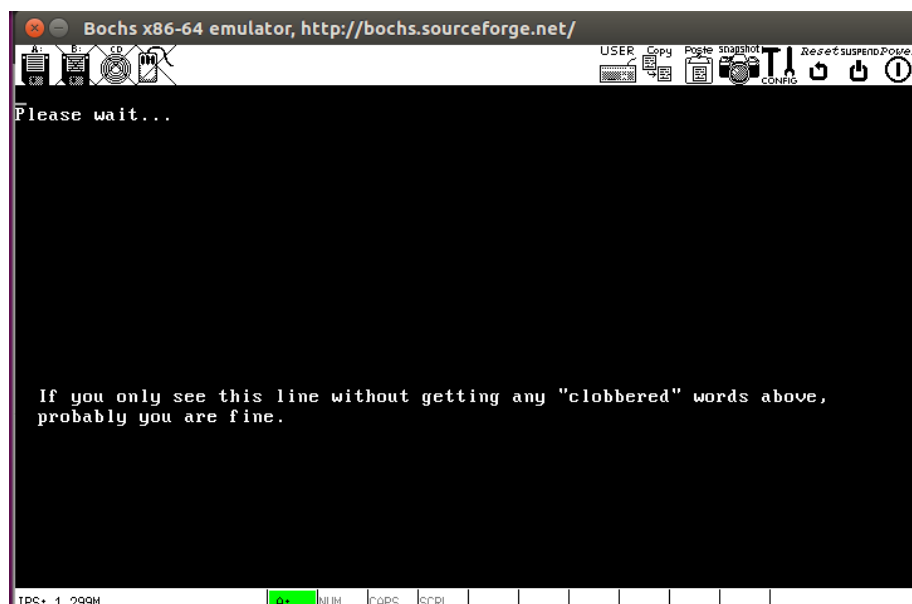
在这里还需要注意 `queue` 函数的使用方法 , 比如 `dequeue` 是要对你所选位置的下一个进行删减。

(看过队友代码 , 发现队友的 `check_sleep` 只要发现一个可以唤醒的任务就跳出循环结束了。我的代码是把所有符合条件的任务唤醒 , `sleep_queue` 队列是根据时间从小到大排序的 , 所以只要发现不符合条件的 , 就可以终止 `break` 跳出了。)

1.7 关于前三个 test

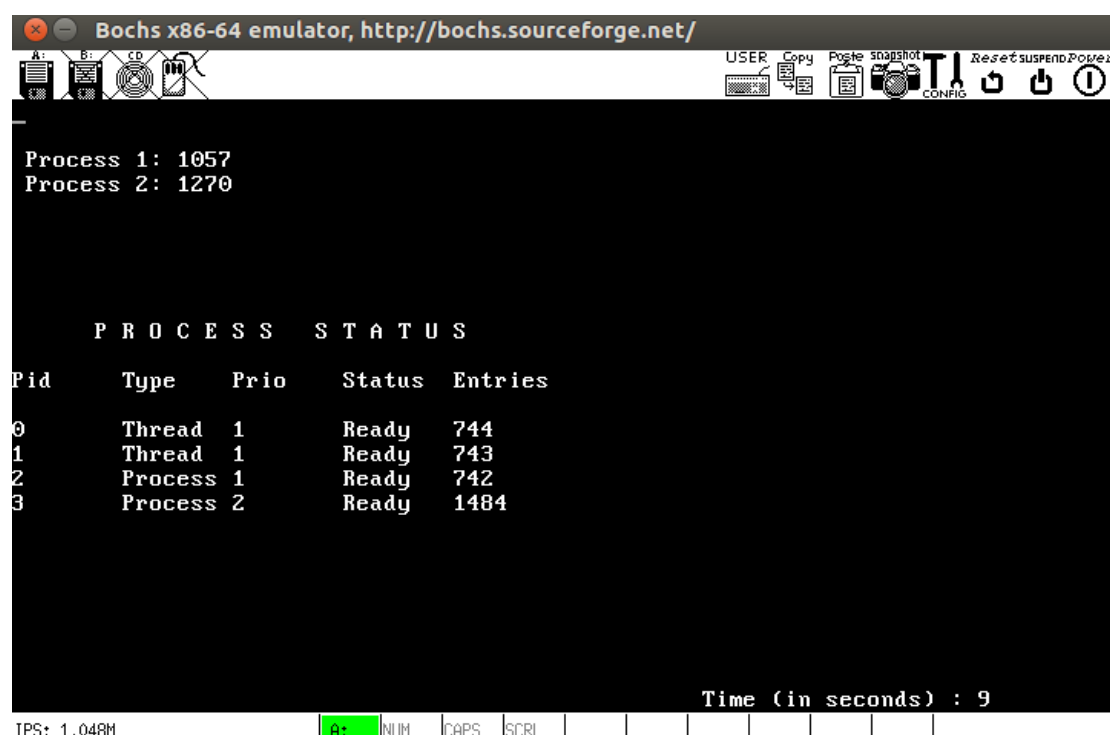
在这里只给出三次实验的结果 :

Test_regs:可以先看到 `please wait` , 经过一段时间会出现 `if you only` 这句话。



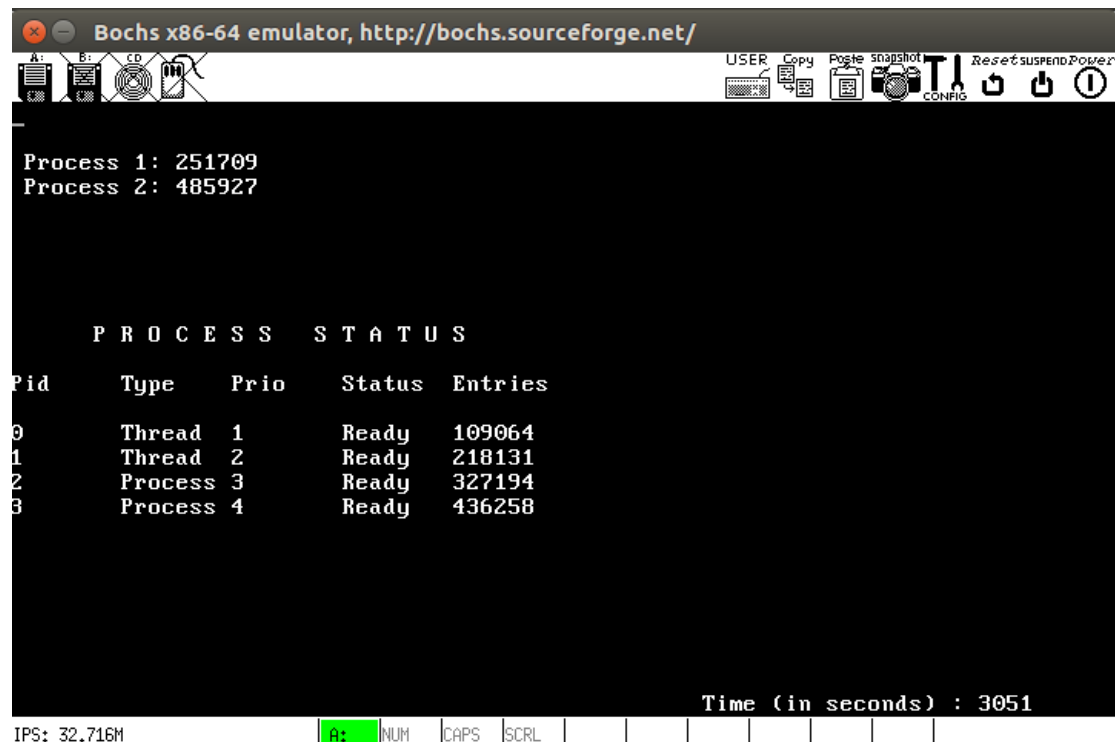
(测试结果与队友一样,不放图了。)

Test_preempt:可以看到 process1 和 2 不断上升,因为这里使用了基于优先级的调度,所以两个进程的时间差很多。一开始我对 time_elapsed 直接使用了 add,导致右下角的时间是一个很大的值。进程也只能显示出一个。



(我设置的是,不同的优先级拥有的时间片数量不一样,第 i 个优先级拥有 $2 \cdot i$ 个时间片,大家都用完了自己的时间片,才会重置这个时间片数据。最开始,是设置了 4 个优先级,并对 4 个优先级设置时间片,如果都是 0 的时候,就清空。在这个任务的时候,成功运行如下图,没有任何问题。但当后面任务飞机那边有段注释掉的代码,我增加到 64 个优先级,那边的程序改掉了自己的优先级,然后就卡住,因为更改优先级的过程中,导致某个优先级的时间片,没有用完,但是已经不存在这个优先级的任务了。类似的还有某任务带优先级剩余的时间片睡觉去了,导致卡住,所以我修改了重置的设置,即当搜了一圈之后,还

没有找到还有时间片可以跑的 `current_running` 的时候，就直接重置时间片，因为肯定是有程序带着剩余的时间片睡觉去了。）



Bochs x86-64 emulator, <http://bochs.sourceforge.net/>

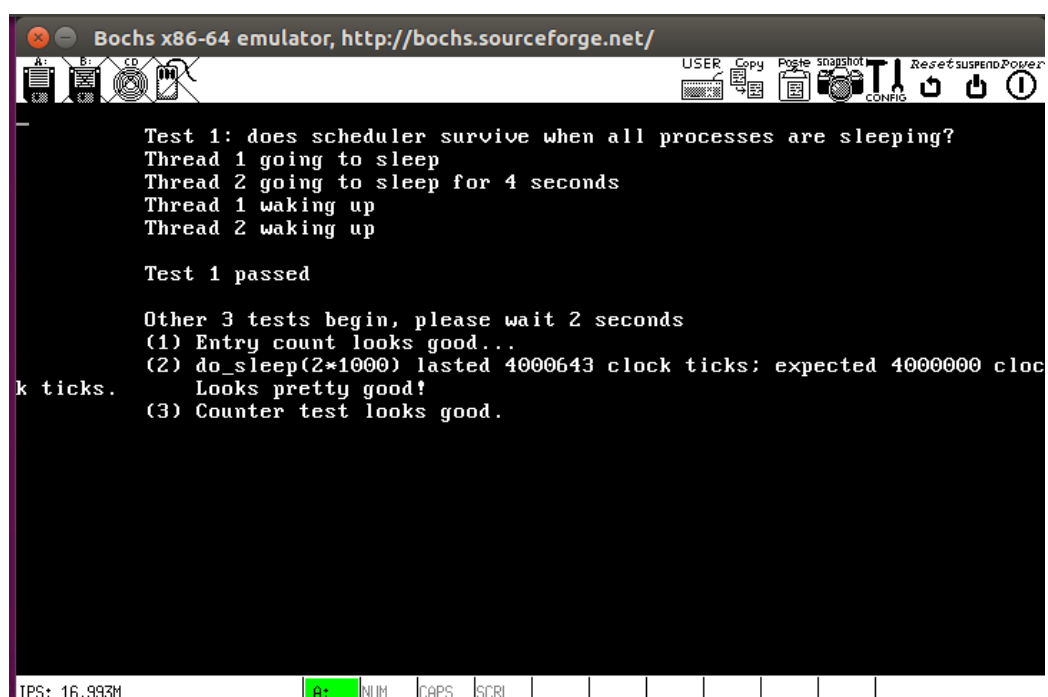
Process 1: 251709
Process 2: 485927

P R O C E S S S T A T U S				
Pid	Type	Prio	Status	Entries
0	Thread	1	Ready	109064
1	Thread	2	Ready	218131
2	Process	3	Ready	327194
3	Process	4	Ready	436258

Time (in seconds) : 3051

IPS: 32.716M

`Tset_blocksleep`:一开始的时候我以为 `dequeue` 是对当前的位置修改，结果做出来 `th2` 没办法唤醒。（测试结果与队友一样，不放图了。）



Bochs x86-64 emulator, <http://bochs.sourceforge.net/>

```

Test 1: does scheduler survive when all processes are sleeping?
Thread 1 going to sleep
Thread 2 going to sleep for 4 seconds
Thread 1 waking up
Thread 2 waking up

Test 1 passed

Other 3 tests begin, please wait 2 seconds
(1) Entry count looks good...
(2) do_sleep(2*1000) lasted 4000643 clock ticks; expected 4000000 clock ticks. Looks pretty good!
(3) Counter test looks good.
  
```

IPS: 16.993M

2.关于三种中断原语的实现

2.1 概述

我们这次需要实现条件变量，信号量，屏障这三种结构。条件队列是每一种条件对应一个队列，当条件满足的时候，我们就将当前任务阻塞进去，或将队列中的任务释放出来。信号量是用于标识一种资源的，以停车位为例，信号量就是指剩余车位的数量；当剩余车位的数量是零的时候，后来的车就只能在外面等（阻塞起来）；而当又有车位空出来的时候，车辆就可以进去了。屏障这个结构的功能是等任务积攒满了之后，一股脑的放到就绪队列里。之后我们讲一下三种结构的具体实现方法。

2.2 三种数据结构

conditionvariables	semaphore	barrier
<pre>typedef struct{ node_t wait_queue; } condition_t</pre>	<pre>typedef struct{ int amount; node_t wait_queue; } semaphore_t</pre>	<pre>typedef struct{ int max; int current; node_t wait_queue; } barrier_t</pre>

2.3 条件变量

条件变量要完成三个函数 `condition_init`，`condition_wait`，`condition_signal` 和 `condition_broadcast`。`Condition_init` 完成了队列的初始化，`condition_wait` 是将当前的

任务放到等待队列里，`condition_signal` 则是将阻塞队列的第一个任务提取出来，`condition_broadcast` 则组要将等待队列里所有的问题都提取出来。

需要注意的问题：

如果使用了 `lock_release` 和 `lock_acquire` 操作的话，要注意不要重复进入临界区，否则会发生 panic。

2.4 信号量

信号量需要实现 `semaphore_init` , `semaphore_up` , `semaphore_down` 这三个操作。
`semaphore_up` 是在有任务完成对资源占用时候做的，如果此时等待队列里有任务，我们就将阻塞队列里第一个被阻塞的任务取出来执行；如果阻塞队列里没有任务，我们就把标志剩余资源数的变量加一。`semaphore_down` 是在有任务要占用资源时候做的，如果此时标志剩余资源数的变量已经是零了，我们就将任务放进等待队列里；如果剩余资源数不是零，我们就可以直接把资源数减一。

2.5 屏障

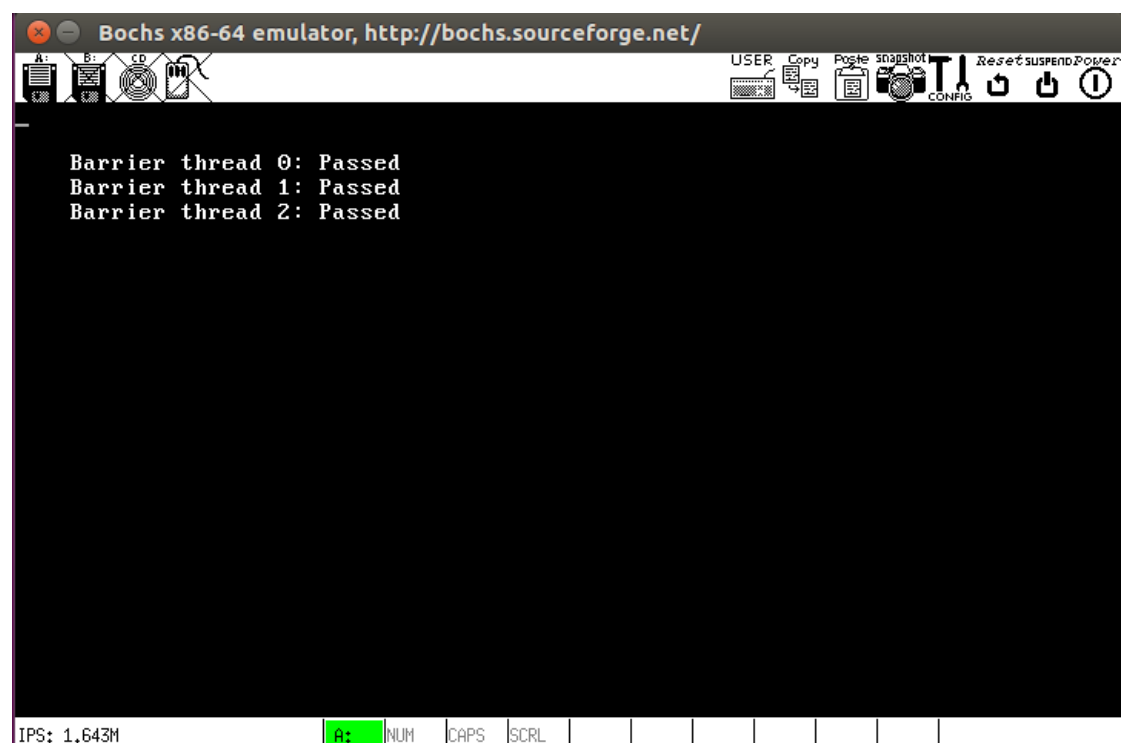
屏障需要实现 `barrier_init` 和 `barrier_wait` 两个操作。`Barrier_wait` 是如果等待队列里的任务数没有达到上限，就一直往里阻塞；如果达到上限就使用 `unblock_all` 把所有的等待队列里的任务都取到就绪队列里。

需要注意的问题：

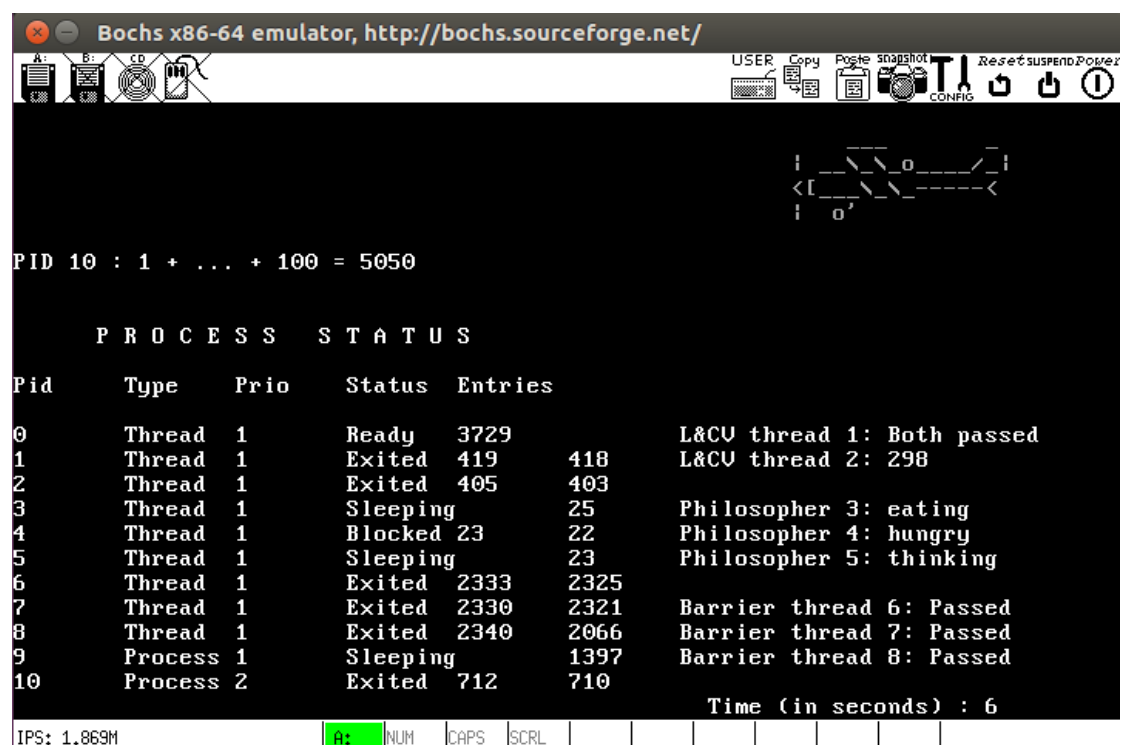
可以直接使用老师写好的 `unblock_one` , `unblock_all` , `block` 函数。如果是自己写的话，要注意 `block` 之后需要调用 `scheduler` 做调度。

2.6 测试结果

Test_barrier: 运行 1000 以后会显示成 pass。（测试结果与队友一样，不放图了。）



Test_all:如果所有的原语都设置正确，就可以看到小飞机飞过。



(我把被注释掉的代码恢复了一部分，所以有个自己调节优先级的东西。)

The screenshot shows a Bochs x86-64 emulator window. The title bar reads "Bochs x86-64 emulator, http://bochs.sourceforge.net/". The window contains a terminal-like interface with the following text:

```

PID 10 : 1 + ... + 100 = 5050
Process 9      priority 34

  P R O C E S S   S T A T U S

Pid      Type    Prio   Status  Entries
0         Thread  1      Ready   20291
1         Thread  1      Exited   419     418   L&CU thread 1: Both passed
2         Thread  1      Exited   405     403   L&CU thread 2: 298
3         Thread  1      Sleeping 115
4         Thread  1      Sleeping 115   Philosopher 3: eating
5         Thread  1      Sleeping 117   Philosopher 4: thinking
6         Thread  1      Exited  2341   2314   Philosopher 5: thinking
7         Thread  1      Exited  2339   2337   Barrier thread 6: Passed
8         Thread  1      Exited  2323   1811   Barrier thread 7: Passed
9         Process 35     Sleeping 7082   Barrier thread 8: Passed
10        Process 1      Exited  1227   1225

Time (in seconds) : 33
IPS: 40.020M

```

3.基于优先级的调度

因为这次实验只有 process1 和 process2 两个进程，所以我这次之设置了一和而两个优先级。具体的执行方法是在 scheduler 中进行设置。记录下进入 scheduler 的总次数，将总次数模十的余数作为标志，如果标志比 6 大就执行优先级为一的任务，否则就执行优先级为二的任务。

(我对不同优先级设置了不同的时间片数量，当存在的优先级把它们的时间片用完的时候，会重置这些时间片数量，在 scheduler.c 里面用 priority_time 数组记录，0 位为时间片记录，即当某个优先级的时间片开始被使用时，++；当某个优先级的时间片用完的时候，--。当时间片记录是 0 的时候重置。但带来一个问题，task2 的 test_all 里面注释掉的代码，有更改自己优先级的功能，可能有的优先级被使用了，++了，但是没有被用到 0，就睡觉去了，或者更改了优先级，该优先级不再存在，导致时间片纪录始终无法到 0，时间片数量无法被重置，但是又找不到能跑的任务，因为他们优先级的时间片都被用完了，所以

后来改成，找了一圈都没找到，那么就重置！发现这样省去了之前很多的操作，更加方便简单了。）

参考文献

[1] Xv6-public-master

