

# Project2 non-preemptive+kernel 设计文档

中国科学院大学

孔静，崔鹏来

2016.10.10

## 1. 多 task 启动与 task switch 设计流程

### 1.1 概述

我们不妨先粗略地看看我们这次任务都需要什么东西。首先，我们需要有一个队列，来保存可以运行的任务。但是光有队列是不够的，因为操作系统还是不知道我们要先执行哪个任务，所以说我们还得做一个 scheduler 来告诉操作系统下一个执行的任务应该是什么。当然，队列里的内容也不应该是一成不变的，所以我们还得做一些可以对队列进行修改的调度，就是 `do_yield` 和 `do_exit`。

### 1.2 队列

可以看到任务是围绕着队列进行的，那么队列是什么东西呢？首先队列里应该有一些标志，用来显示队列的首尾位置和是否为空这些基本情况。然后，队列里应该有这对于任务的描述，也就是 `pcb`。所谓 `pcb` 相当于就是一个任务的档案，拿到这个档案，任务执行到哪里，任务接下来要做什么，操作系统都会一目了然。`pcb` 中最重要的东西是一个叫 `context` 的结构。这个结构里记录了 `edi`, `esp`, `ebx` 等比较重要的寄存器和栈指针，有了这个东西我们就可以记录任务运行到哪里。这样就算任务的执行被中断了，我们也可以按图索骥，找到原来运行的位置，继续执行。

### 1.3 PCB

那么，你可能会问，这个 pcb 这么厉害，那它要怎么保存呢？其实说起来还是很简单的，我们只需要把 edi, esp, ebx 这些值从寄存器中推进栈里，然后把指向 context 的指针记录下来即可。在这里需要注意一点就是 eip 这个东西是不用保存下来的，因为我们在 scheduler 中做了一个 call scheduler 的操作，这个操作会自动将 eip 保存到 ebp 上面的位置。

## 1.4 运行流程

了解了这些内容之后就让我们以 task1 为例，看看这个调度器究竟是怎么运行的。

首先，我们跑到 kernel.c 中，完成队列的初始化，也就是将队列的几个标志填好，并把要执行的几个任务的 pcb 填到队列里。然后我们就做了一个 scheduler\_entry，这个 scheduler\_entry 到位之后，马上把当前运行程序的 eip 保存下来，然后调用 scheduler。Scheduler 会选取队列首位的任务（FIFO）作为我们马上要执行的 current\_running。之后，继续 scheduler\_entry，把要执行的函数的 edi, esp.....信息从栈提取到寄存器中，之后正好 return 到 eip，开始执行任务。一个任务运行到一定阶段，可能会做一个 do\_yield 或者 do\_exit 操作。这个 do\_yield 可以保存 pcb，并且通过 scheduler\_entry 操纵 scheduler，实现任务的轮转。这里需要注意 do\_yield 和 do\_exit 的区别。Do\_yield 操作是将当前运行的操作转移到队尾，而 do\_exit 操作是直接当前运行的操作从队列中踢出去。这样我们的任务就会不断地轮转，直到所有的任务都通过 do\_exit 跳出队列为止。

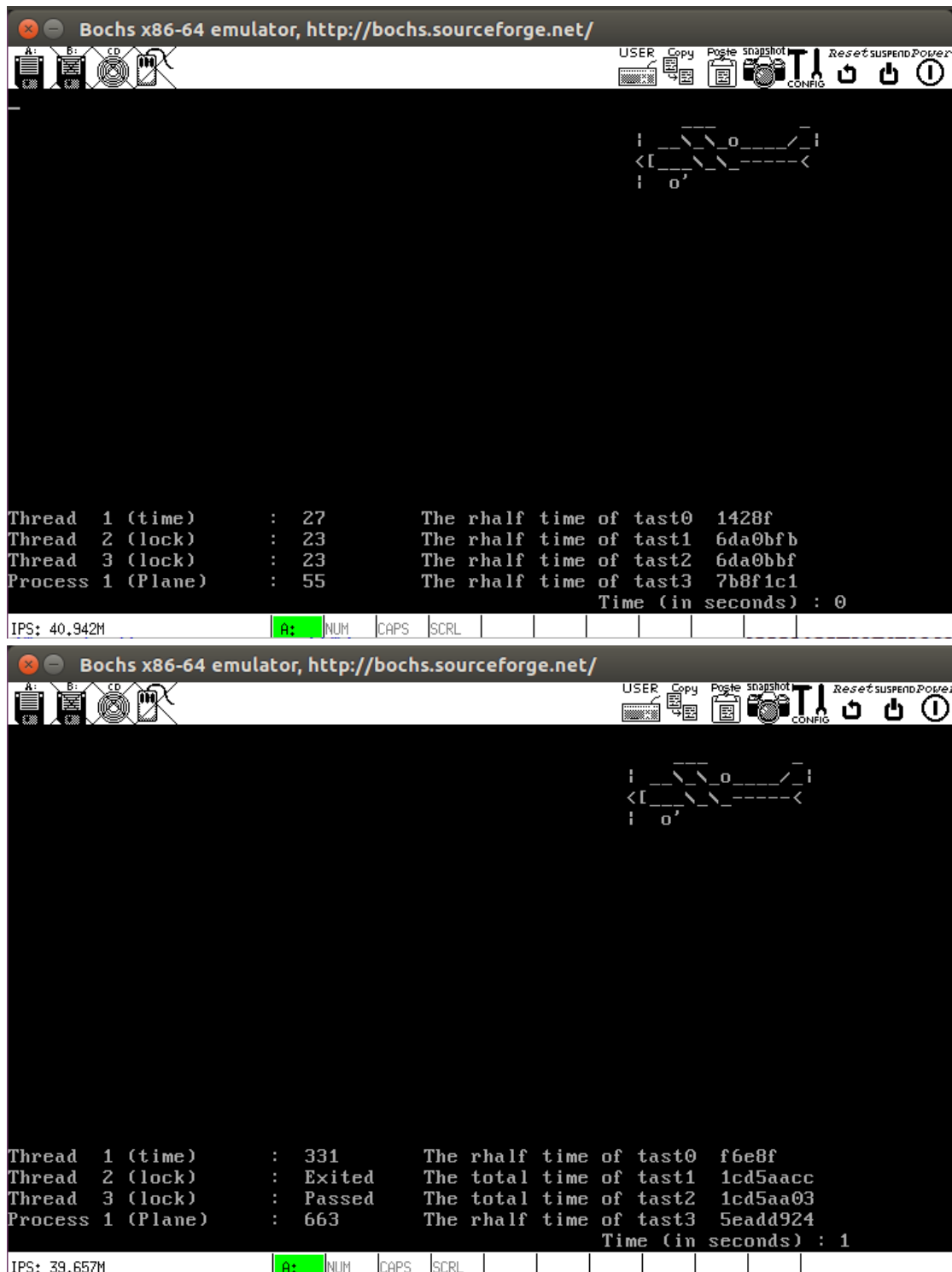
## 1.5 注意事项

这个任务中，遇到的主要问题集中在 save pcb 这里，我们在进行栈操作的时候，一定要将 pcb 中的 edi, esp, ebx.....寄存器对应好。

## 1.6 实验结果

实验完成后，可以看到小飞机飞过。

设置 type=1 ( 自定义变量 type=0 为 fair scheduler ) ,task 队列 1 ,SPIN=TRUE ,  
开始跑，结果如图。( 右下角 rhalf time 表示正在跑，右侧的时间是上次 exit 之前累计跑  
的时间；total time 表示已经跑完，总共跑了多久。)



若设置 `type=0`，即使用公平调度器，由于比较时间选择执行的程序流程耗时比 FIFO 多，所以可以看出飞机跑的相对较慢。与之前的 FIFO 调度器执行结果比较，可以看出不计算 context switch 花销的运行时间是一样的（如线程 2 的总运行时间 `1cd5aa78`，线程 3 的总运行时间 `1cd5aa8f`，但左边显示的线程时间大大超过右侧，说明公平调度器耗时较长，当然也因为我写的不好 orz。）

The screenshot shows a Bochs x86-64 emulator window. The main display area shows a simple flight simulation with a plane and some terrain. Below the display is a terminal window with the following output:

```
Thread 1 (time)      : 200993      The rhalf time of tast  27dc811b
Thread 2 (lock)      : Exited      The total time of tast1 1cd5aa78
Thread 3 (lock)      : Passed      The total time of tast2 1cd5aa8f
Process 1 (Plane)    : 279          The rhalf time of tast3 280b991b
                                     Time (in seconds) : 1
```

At the bottom of the terminal, there is a status bar showing "IPS: 49.384M" and some keyboard status indicators like "NUM", "CAPS", and "SCRL".

## 2. Context switch 开销测量设计流程

### 2.1 概述

这个任务比较简单，我们只需要在 `process` 和 `th` 中用 `get_timer` 函数把时间记录下来，然后使用 `util` 的函数里定义的 `print_str` 函数和 `print_int` 函数把测得的时间输出到屏幕中就可以。

### 2.2 时间记录的位置

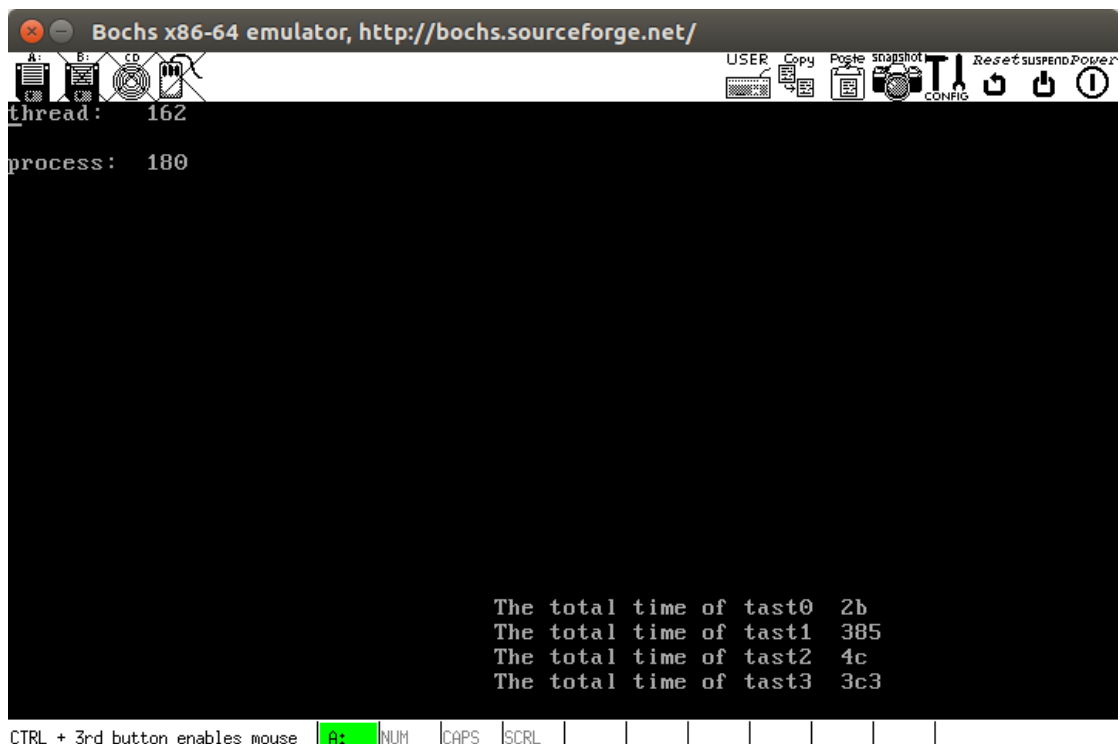
因为在这个试验中我们一共有两个线程和一个运行了两次次的进程。所以线程设置

记录时间的位置可以定在第一个线程 `do_yield` 之前和第二个线程开始运行的时候。而对于进程，我们需要进行一个判断，第一次进入进程的时候，我们在 `do_yield` 之前记录时间；然后如果是第二次进入进程，那么我们就直接进入时间。

## 2.3 实验结果

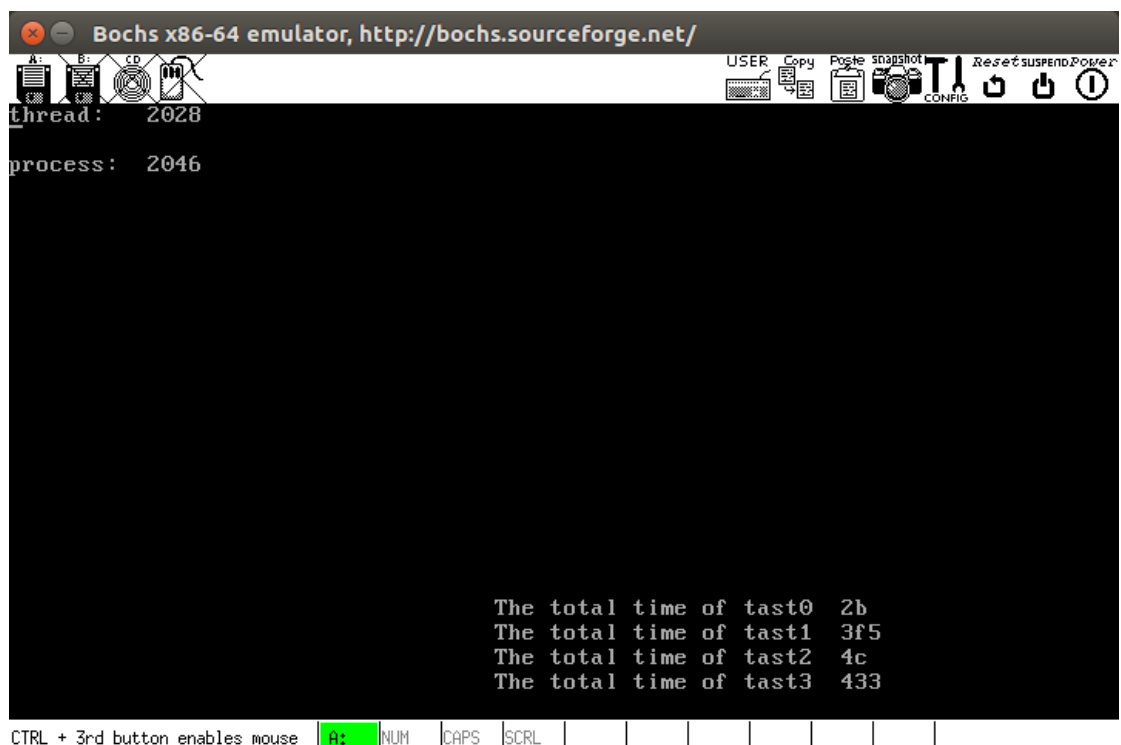
完成后可以在屏幕左上角看到函数运行的时间。

设置 `type=1`，task 队列 2。（最开始的版本，我记得 thread 是 147，process 是 165，这里因为后面的实验加了其他代码，context switch 耗时增加。）

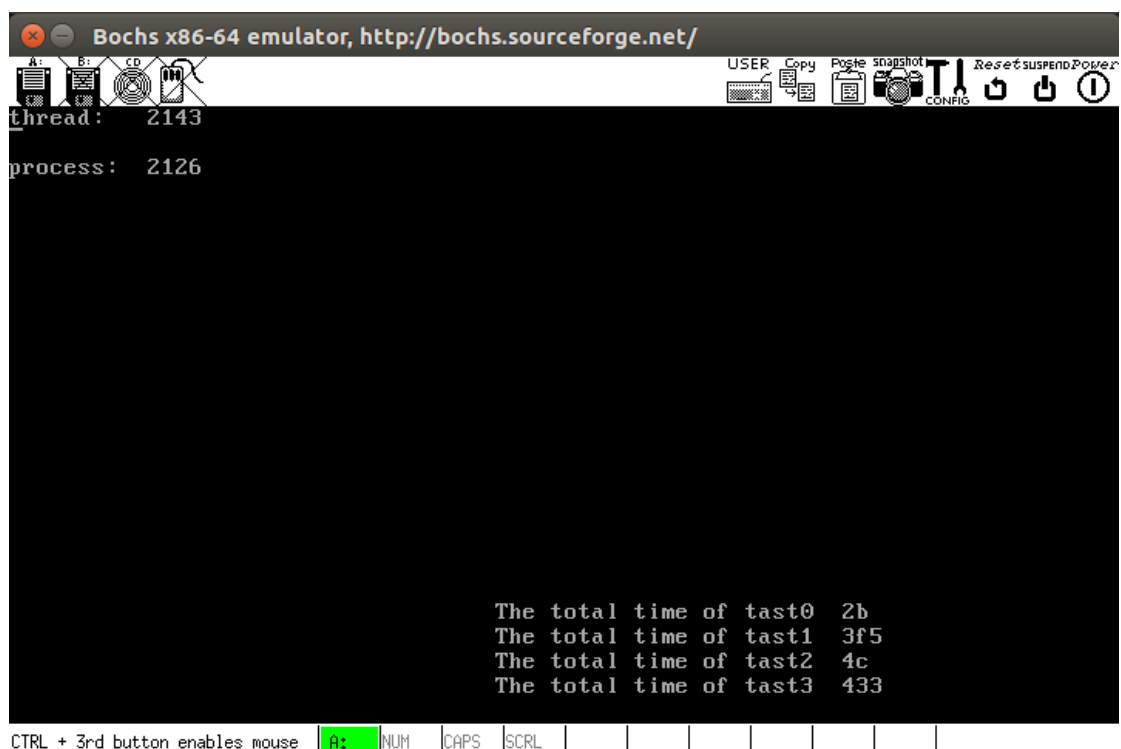


```
void do_yield(void)
{
    save_pcb();
    //print_str(current_running->line, 35, strings1);
    //print_int(current_running->line, 57, current_running->pid);
    //print_u64(current_running->line, 60, current_running->runtime);
}
```

将 task4 涉及的输出，即上图注释掉的代码加入，context switch 时间明显增加。



再把 `type=0` , 改为 `fair scheduler` , 时间又增加了, 不过因为这里运行的 `pcb` 数量少, 在比较哪个程序运行时间少上耗时少, 所以没有明显增加。



### 3. 线程间互斥锁设计流程

#### 3.1 概述

我们的操作系统一般是有好几个进程和线程在同时跑的。如果这些进程/线程如果可以随意的互相干扰,那就一定会产生错误。比如如果我们两个进程同时对一个共享的资源进行修改,那我们就会难以预测修改之后的结果是什么。这样显然是不行的。所以我们要设计原子操作,使用锁的结构将不应该被打断的操作保护起来,从而使进程可以文件地运行。

### 3.2 具体操作

在我们的这个任务中,就是设计了一个简单的互斥锁。在 th4 中,我们设计了几个简单的线程操作,这些线程主要是进行了 lock\_acquire 和 lock\_release 这两个操作。Lock\_acquire 操作会将锁的状态标记成 locked 如果我们的锁此时已经是 locked 的了,那么我们就将当前运行的任务扔进 block 队列里面。而 lock\_release 操作则可以将 block 队列中的第一个任务取到 ready 队列的第一个来运行。在这个实现的过程中,因为我们并没有定义将任务放在队列第一位的操作,所以我们需要自己写。具体方法是将 head 减一,然后将 pcb 的地址放在数组中第 head 位里。

### 3.3 注意事项

在这里我们需要注意一个问题就是,我们在将 block\_queue 中第一个任务取到 ready 队列的时候需要先判断一下 block 队列是否为空。如果我们不进行判断,就会导致程序提前结束。还有就是 block 队列和 ready 队列一定要赋好初值,如果不进行处理,那么 block 和 ready 就是指向同一地址的,这样相当于我们只有一个 ready 队列。

### 3.4 执行结果

设置 SPIN=FALSE, type=1, task 队列 3。(结果如图,和指导书一致。)

The screenshot shows the Bochs x86-64 emulator window. The title bar reads "Bochs x86-64 emulator, http://bochs.sourceforge.net/". The window contains a black terminal area with white text showing the execution of four threads (thread 1, 2, 3, 4) and a lock. The logs show thread 1 initializing and acquiring the lock, then yielding to thread 2. Thread 3 then yields to thread 4, which acquires the lock. Thread 1 then releases the lock and exits. Thread 4 releases the lock and exits. Thread 2 then acquires the lock, releases it, and exits. Thread 3 then acquires the lock, releases it, and exits. Finally, thread 2 exits. At the bottom right of the terminal, timing statistics are displayed:

```

The total time of tast0  b25e0a2d
The rhalf time of tast1  4758ce76
The total time of tast2  8eb1a347
The total time of tast3  6b053cf4

```

Below the terminal area, the status bar shows "IPS: 64,939M" and a row of keyboard shortcuts: "A: NUM CAPS SCRL" followed by several empty boxes.

但最开始，和指导书有一点出入，因为一开始写 unblock 的时候，把 blocked 队列 push 进了 ready 队列，即放在了末尾。但貌似 blocked 队列应该优先级更高，后改为放到队列最前。（下图注释掉部分，就是一开始写的直接 push 进 ready 队列。）

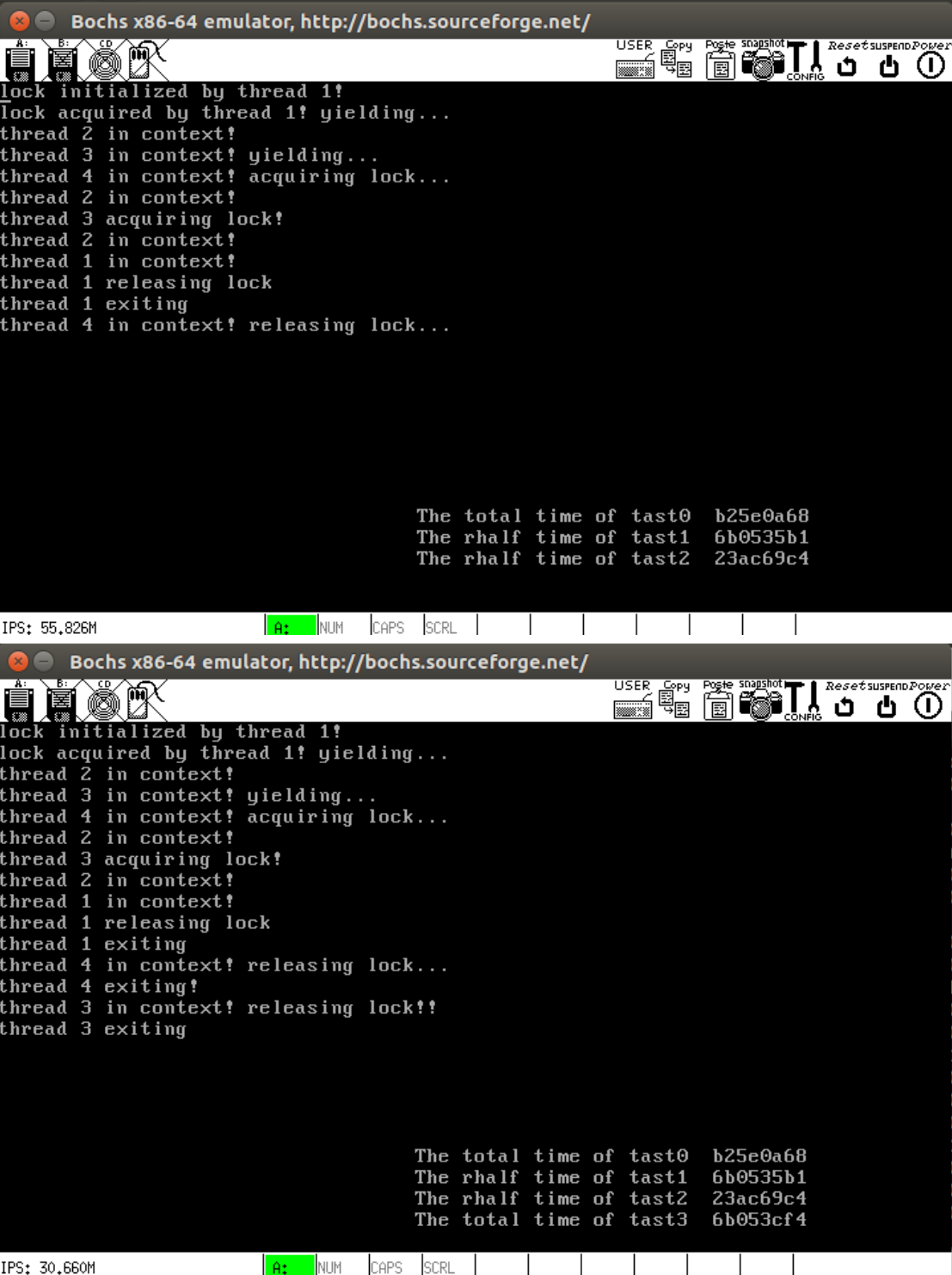
```

void unblock(void)
{
    pcb_t *Pcb;
    int i;
    int block_count = queue_size(blocked_queue);
    ready_queue->head = (ready_queue->head - block_count + ready_queue->
    i = ready_queue->head;
    while(block_count > 0)
    {
        Pcb = queue_pop(blocked_queue);
        ready_queue->pcbs[i] = Pcb;
        if(ready_queue->isEmpty == TRUE)
            ready_queue->isEmpty = FALSE;
        i++;
        i %= ready_queue->capacity;
        block_count--;
    }
    /*while(block_count > 0)
    {
        Pcb = queue_pop(blocked_queue);
        Pcb->state = READY;
        queue_push(ready_queue, Pcb);
        block_count--;
    }*/
}

```

OK，再把 type=0，即 fair scheduler 跑一次。因为调度器改变了，跑的顺序也变了。





The image displays two screenshots of the Bochs x86-64 emulator window, showing the execution of a program with multiple threads and a lock. The emulator's title bar indicates the URL <http://bochs.sourceforge.net/>. The interface includes a toolbar with icons for USER, Copy, Paste, snapshot, CONFIG, Reset, suspend, and Power.

**Top Screenshot:**

```
lock initialized by thread 1!
lock acquired by thread 1! yielding...
thread 2 in context!
thread 3 in context! yielding...
thread 4 in context! acquiring lock...
thread 2 in context!
thread 3 acquiring lock!
thread 2 in context!
thread 1 in context!
thread 1 releasing lock
thread 1 exiting
thread 4 in context! releasing lock...
```

The total time of tast0 b25e0a68  
The rhalf time of tast1 6b0535b1  
The rhalf time of tast2 23ac69c4

IPS: 55,826M

**Bottom Screenshot:**

```
lock initialized by thread 1!
lock acquired by thread 1! yielding...
thread 2 in context!
thread 3 in context! yielding...
thread 4 in context! acquiring lock...
thread 2 in context!
thread 3 acquiring lock!
thread 2 in context!
thread 1 in context!
thread 1 releasing lock
thread 1 exiting
thread 4 in context! releasing lock...
thread 4 exiting!
thread 3 in context! releasing lock!!
thread 3 exiting
```

The total time of tast0 b25e0a68  
The rhalf time of tast1 6b0535b1  
The rhalf time of tast2 23ac69c4  
The total time of tast3 6b053cf4

IPS: 30,660M

```

lock initialized by thread 1!
lock acquired by thread 1! yielding...
thread 2 in context!
thread 3 in context! yielding...
thread 4 in context! acquiring lock...
thread 2 in context!
thread 3 acquiring lock!
thread 2 in context!
thread 1 in context!
thread 1 releasing lock
thread 1 exiting
thread 4 in context! releasing lock...
thread 4 exiting!
thread 3 in context! releasing lock!!
thread 3 exiting
thread 2 in context!
thread 2 in context!

The total time of tast0 b25e0a68
The rhalf time of tast1 8eb19ceb
The total time of tast2 8eb1a35c
The total time of tast3 6b053cf4

```

## 4. 公平调度器设计流程多 task 启动与 task switch 设计流程

### 4.1 概述

我们之前都是使用一个简单的 FIFO 算法实现任务的调度。这种算法实现起来非常简单，但是也有两个致命缺陷。其一：如果我们先运行的任务执行的时间过长，那么后面的任务就不得不进行长时间的等待。其二：如果我们运行一个 IO 密集型的操作，那么 cpu 会把很长的时间浪费到等待上，这样 cpu 的利用率会降低很多。

### 4.2 时间记录方法

为了记录函数运行的时间，我们在 pcb 中增加了一个 runtime 常量，用于记录在整个的运行过程中一个进程/线程运行的总时间。具体的操作方法是使用一个 rdtsc 函数，这个 rdtsc 函数会将系统运行的总的周期数记录在寄存器 eax 内。然后我们将这个 rdtsc 函数分别设置在恢复寄存器之后和将寄存器内容保存进栈之前的位置。在恢复寄存器之后，我们把 runtime 减去系统运行时间；在寄存器内容保存进栈之前，我们把 runtime 加上此时系统运行的时间。这样，我们就可以得到一个任务运行的总时间。

再解释一下这个逻辑，即，进入程序时， $\text{runtime} - \text{time\_in}$ ，保存寄存器出程序时， $\text{runtime} + \text{time\_out}$ 。整体来看就是， $\text{runtime} - \text{time\_in} + \text{time\_out} = \text{runtime} + (\text{time\_out} - \text{time\_in})$  就是加上了运行时间。考虑机器溢出进位，减法的时候，肯定不够减，会借位，但同理加法的时候肯定会溢出，把之前借来的位还回去，即  $\text{runtime} - \text{time\_in} + 0x1\ 0000\ 0000 + \text{time\_out} - 0x1\ 0000\ 0000$ 。不过我们并没有考虑当运行时间超过了 64 位寄存器所能记录的情况。

### 4.3 实现细节

我们的公平调度器需要选取运行时间最短的任务作为接下来要执行的任务。所以，我们首先得在 pcb 中添加一个 runtime，用于记录我们这个任务到目前为止的总的运行时间。然后每次在 scheduler 中进行调度的时候，我们都要先比较一下 ready\_queue 中各个 pcb 的 runtime 并选出其中 runtime 最小的 pcb，将之放到 current\_running 中。到这为止，操作还没有做完，因为我们所选取的这个 pcb 有可能是位于 ready 队列中央的，这有可能会造成 ready 队列从中间断掉。所以我们还需要进行一个操作，将选中的 pcb 之前的 pcb 向后顺延一位，再次形成一个连续的 ready\_queue。

### 4.4 遇到的问题

这次实验的过程中遇到了一个问题就是，runtime 最后输出的值出现了负数。原因是我们记录时间时，加上系统时间的操作是用汇编语言写在 save\_pcb 里的。而我们在做 exit 的时候不用做 save\_pcb。也就是说在 do\_exit 的时候，我们有在恢复寄存器内容之后，减去系统时间，却没有加回来，这就造成了出现负数的情况。

### 4.5 实验结果

跑的结果，在之前的任务中均有展示，没有自己写测试用例（觉得应该没错=。= 也许迷之自信了）。

## 5. 关键函数

### **\_Start :**

对 pcb 和队列进行一些初始化，并最终跳到 scheduler\_entry，开始调度

### **Scheduler :**

会在 ready\_queue 中取出运行时间最短的任务加载到 current\_running 上

### **Scheduler\_entry :**

会保存当前的 eip 并调用 scheduler，并把函数要用的参数传到通用寄存器

### **Save\_pcb :**

会将通用寄存器中的值按顺序压进栈，并将其地址记录到 context 指针中

### **Do\_yield :**

将正在运行的任务 push 到队尾，然后调用 scheduler\_entry

### **Do\_exit :**

将当前运行的任务标记为 exit，并调用 scheduler\_entry

### **Lock\_acquire:**

如果当前没锁，就获得锁。否则将当前操作放到 block 队列中

### **Lock\_release :**

如果此时 block 队列里有任务，就将之取到 ready 队列队首

### **Block :**

把当前进行的操作放进 block 队列里

### **Unblock :**

如果 block 队列为空就什么都不做，如果非空就将任务取到 ready 队列头部

## 6. 参考文献

[1] Xv6-public-master