

# 中山大学本科生实验报告

## (2018 学年春季学期)

课程名称: Operating System

任课教师: 饶洋辉

年级+班级	16 级 + 2 班	专业 (方向)	信息与计算科学
学号	16343065	姓名	桑娜

### 1. 实验目的

- 了解 priority-fifo、priority-change、priority-preempt 三个 test 的测试机制;
- 通过修改相关函数通过上述 3 个 test;
- 理解信号量 semaphore 和锁 lock 的概念;

### 2. 实验过程

#### (一) Test 源文件分析

##### priority-change

##### 测试目的:

如果一个正在运行的线程降低自己的优先级, 以至于它不再是系统中优先级最高的线程, 那么它就应当立即让出 CPU。

##### 过程分析:

在**主线程** `test_priority_change` 中,

```
msg ("Creating a high-priority thread 2.");  
thread_create ("thread 2", PRI_DEFAULT + 1, changing_thread, NULL);
```

- 打印消息 “正在创建一个高优先级的线程 thread 2。” ;
- 创建了 1 个名为 “thread 2”, 优先级为默认优先级 (32) + 1, 也就是 33 的线程, 高于主线程的 32。

因此执行**子线程** `changing_thread` :

```
msg ("Thread 2 now lowering priority.");  
thread_set_priority (PRI_DEFAULT - 1);
```

- 打印消息 “Thread 2 正在降低优先级。” ;
- 通过**函数** `thread_set_priority` 将 thread2 自己的优先级设为默认优先级 (32) - 1, 也就是 31。
- 分析 1:

优先级为 32 的主线程, 创建了优先级为 33 的子线程 thread 2 后, thread 2 应当立即抢占 CPU; thread 2 抢占到 CPU 后, 降低自己的优先级为 31, 低于主线程的 32, 应当立即让出 CPU。

即 thread 被放回 ready\_list 中, 而**主线程** `test_priority_change` 得以继续执行:

```
msg ("Thread 2 should have just lowered its priority.");  
thread_set_priority (PRI_DEFAULT - 2);
```

- 打印消息 “thread 2 应该已经降低了它的优先级” ;
- 将主线程自己的优先级设为默认优先级 (32) - 2, 也就是 30。

- 分析 2:

正在占用 CPU 的主线程，将自己的优先级降低为 30，低于了 thread 2 的 31，应当立即让出 CPU；即主线程被放回 ready\_list，而子线程 **changing\_thread** 得以继续执行：

```
msg ("Thread 2 exiting.");
```

➤ 打印消息 “thread 2 正在退出。”

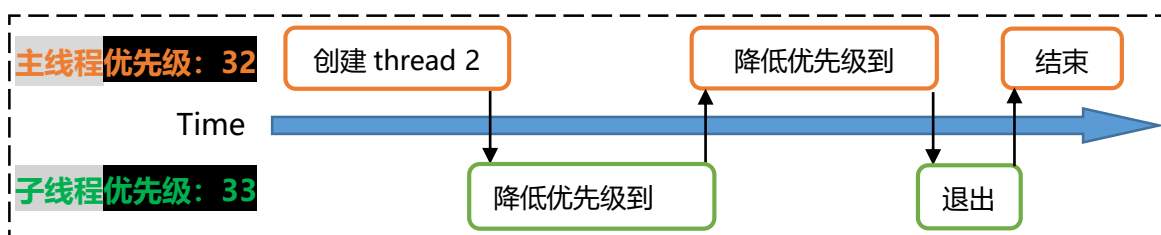
这时候子线程 **changing\_thread** 已经结束了，主线程 **test\_priority\_change** 获得 CPU，继续

```
msg ("Thread 2 should have just exited.");
```

执行：

➤ 打印消息 “thread 2 应该刚刚已经退出了。”

整个过程如下：



这个测试的关键在于：

- (1) 主线程创建了更高优先级的子线程后，应当让出 CPU，也就是需要修改函数 **thread\_create**；
- (2) 主/子线程更改自己的优先级后，如果比就绪队列中的最高优先级低，也要让出 CPU，即需要修改函数 **thread\_set\_priority**。

### 结果分析：

(1) 修改之前，失败样例

```
Executing 'priority-change':
(priority-change) begin
(priority-change) Creating a high-priority thread 2.
(priority-change) Thread 2 should have just lowered its priority.
(priority-change) Thread 2 should have just exited.
(priority-change) end
Execution of 'priority-change' complete.
(priority-change) Thread 2 now lowering priority.
(priority-change) Thread 2 exiting.
sangna@16343065sang: ~/pintos/src/threads/build$
```

根据打印出来的信息，一直执行主线程，直到 end 之后才执行子线程，这是不对的。

(2) 修改之后，成功样例

```
Executing 'priority-change':
(priority-change) begin
(priority-change) Creating a high-priority thread 2.
(priority-change) Thread 2 now lowering priority.
(priority-change) Thread 2 should have just lowered its priority.
(priority-change) Thread 2 exiting.
(priority-change) Thread 2 should have just exited.
(priority-change) end
Execution of 'priority-change' complete.
```

可以看到线程是按照上面的流程图运行的。

## priority-fifo

### 测试目的:

创建几个拥有相同优先级的线程，确认它们一直以相同的轮转顺序运行。如果在某次迭代中，顺序不同于以前，那么就有问题。

### 过程分析:

- 定义 **结构体 simple\_thread\_data**
- 定义线程的数量 THREAD\_CNT，和迭代次数 ITER\_CNT，均为 16。

```
struct simple_thread_data
{
    int id;                /* 休眠线程 ID. */
    int iterations;        /* 目前已迭代次数. */
    struct lock *lock;     /* 保护输出buffer的锁 */
    int **op;              /* 处于输出buffer中的位置 */
};
```

```
#define THREAD_CNT 16
#define ITER_CNT 16
```

- 在**主线程 test\_priority\_fifo**中，

```
struct simple_thread_data data[THREAD_CNT];
struct lock lock;
```

- 定义了 simple\_thread\_data 数组 data，大小为 16；
- 定义了一个 lock 型的 **结构体 lock**。

lock 结构体定义在 `src/threads/synch.h` 中，它是为了控制线程能否获取到某一资源(在这里，这个资源是输出 buffer，也就是 output 所指向的用来记录线程执行顺序的区域)。

```
struct lock
{
    struct thread *holder;
    struct semaphore semaphore;
};
```

holder 是某一时刻拥有这个 lock 的线程，在 lock 被 holder 拥有的时间内，其他线程无法访问指定的资源。

**结构体 semaphore** 定义在同一个文件中：

```
struct semaphore
{
    unsigned value;        /* Current value. */
    struct list waiters;   /* List of waiting threads. */
};
```

使用 lock 时，value 的值为 1，waiters 是等待拥有 lock 的 list。

```
output = op = malloc (sizeof *output * THREAD_CNT * ITER_CNT * 2);
ASSERT (output != NULL);
lock_init (&lock);
```

- 初始化指向 int 型的指针 output，op，它们所指向的区域是 2\*线程数\*迭代次数 倍的 \*output 的大小。断言内存分配成功了；
- 调用 **函数 lock\_init**

lock\_init 定义在 synch.c 中：先确认 lock 指针是指向一个实体的，然后把 lock 的拥有者设为

```
void
lock_init (struct lock *lock)
{
    ASSERT (lock != NULL);

    lock->holder = NULL;
    sema_init (&lock->semaphore, 1);
}
```

```
void
sema_init (struct semaphore *sema, unsigned value)
{
    ASSERT (sema != NULL);

    sema->value = value;
    list_init (&sema->waiters);
}
```

NULL。

用 1 初始化 lock 的 semaphore 成员的 value，初始化 semaphore 的等待队列。

- For 循环 16 次，创建了 16 个子线程 simple\_thread\_func，它们的优先级均为 33。

```
thread_create (name, PRI_DEFAULT + 1, simple_thread_func, d);
```

这里的 “d” 是这样的：

```
char name[16];
struct simple_thread_data *d = data + i;
snprintf (name, sizeof name, "%d", i);
d->id = i;
d->iterations = 0;
d->lock = &lock;
d->op = &op;
```

- i 从 0 循环到 15，d 指向 data[i]，也就是之前定义好的 simple\_thread\_data 数组 data 中的第 i 个元素；
- 第 i 个子线程的名字为 “i” ；
- 初始化 d 所指向的 simple\_thread\_data 的数据成员；
- 在这里，d 作为参数传给 simple\_thread\_func。

- 来看看子线程 simple\_thread\_func：

```
struct simple_thread_data *data = data_;
int i;
```

- 定义了指向 simple\_thread\_data 结构体类型的指针 data，用传入的参数 data\_ (也就是上面所说的 d) 初始化它。

```
for (i = 0; i < ITER_CNT; i++)
{
    lock_acquire (data->lock);
    *(*data->op)++ = data->id;
    lock_release (data->lock);
    thread_yield ();
}
```

- For 循环 i 从 0 到 15，共迭代 16 次；
- 子线程通过 lock\_acquire 获得 lock 的拥有权；
- 把 data 所指向的结构体的 id，也就是线程 i 的 id 写入输出 buffer 中，并且把 op 往后移；
- 子线程通过 lock\_acquire 放弃 lock 的拥有权；
- 子线程礼让出 CPU，切换到就绪状态，进入 ready\_list 中。

在 synch.c 中查看 lock\_acquire:

```
lock_acquire (struct lock *lock)
{
    ASSERT (lock != NULL);
    ASSERT (!intr_context ());
    ASSERT (!lock_held_by_current_thread (lock));

    sema_down (&lock->semaphore);
    lock->holder = thread_current ();
}
```

断言 lock 没有被正在运行的线程拥有, 因为 pintos 中的 lock 不是“递归的”, 一个正在拥有某一 lock 的线程不可以再去获得这个 lock 的拥有权。

再在同一文件中查看 sema\_down.

```
old_level = intr_disable ();
while (sema->value == 0)
{
    list_push_back (&sema->waiters, &thread_current ()->elem);
    thread_block ();
}
sema->value--;
intr_set_level (old_level);
```

在关闭中断功能后, 因为 sema 的 value 初始值是 1, 所以直接让 value 减 1, 这就变为了 0。修改正在运行的线程(假设为 A)为 lock 的拥有者。这个时候, 如果获得 CPU 的线程变为了 B, 并且它执行了 lock\_acquire, 因为 value 为 0, 那么线程 B 的 elem 就会到 sema 的等待队列中排队, 同时线程会被阻塞, schedule 函数调度下一个线程。

```
lock_release (struct lock *lock)
{
    ASSERT (lock != NULL);
    ASSERT (lock_held_by_current_thread (lock));

    lock->holder = NULL;
    sema_up (&lock->semaphore);
}
```

同一文件中, 查看 lock\_release

首先要确认正在占用 CPU 的线程是 lock 的拥有者, 然后把 lock 的拥有者设为 NULL, 最后调用 sema\_up。

再在同一文件中查看 sema\_up。

```
old_level = intr_disable ();
if (!list_empty (&sema->waiters))
    thread_unblock (list_entry (list_pop_front (&sema->waiters),
                                              struct thread, elem));
sema->value++;
intr_set_level (old_level);
```

在关闭中断功能后, 将排在 sema 的等待队列中的第 1 个 elem 所代表的线程 unblock, 也就是把它状态设为 READY, 并放入 ready\_list 中。最后要把 sema 的 value 加 1。

● 回到 **主线程 test\_priority\_fifo**

```
thread_set_priority (PRI_DEFAULT + 2);
```

在创建 16 条子线程之前，它先将自己的优先级设为了 34，高于子线程的 33，所以 16 条子线程虽然被创建了，但是只是放到了 ready\_list 里面，并没有执行。因而，主线程可以连续地不被打扰创建 16 条子线程。

```
thread_set_priority (PRI_DEFAULT);
```

而在创建子线程的 for 循环结束后，主线程将自己的优先级设为了 32，这个时候位于 ready\_list 中的 16 个子线程可以抢占 CPU 了。直到它们全部执行完毕，主线程才可以获得 CPU 继续执行。

```
ASSERT (lock.holder == NULL);

cnt = 0;
for (; output < op; output++)
{
    struct simple_thread_data *d;

    ASSERT (*output >= 0 && *output < THREAD_CNT);
    d = data + *output;
    if (cnt % THREAD_CNT == 0)
        printf("(priority-fifo) iteration:");
    printf(" %d", d->id);
    if (++cnt % THREAD_CNT == 0)
        printf("\n");
    d->iterations++;
}
```

- 断言 lock 没有拥有者；
- 用指针 output 遍历整个输出 buffer，获取到存储的 id 值 \*output；
- 结构体指针 d 指向 data[id]，也就是说 d 按执行顺序指向相应的线程；
- cnt 用来计数；
- 在每次循环的末尾，d 所指向的线程结构体的迭代次数加 1；
- 正确的情况下，这个代码的输出格式应该为：

*(priority-fifo) iteration: 0 1 2 ... .. 13 14 15*

*(priority-fifo) iteration: 0 1 2 ... .. 13 14 15*

*(priority-fifo) iteration: 0 1 2 ... .. 13 14 15*

... ..


共 16 行。

并且每个线程结构体的 iteration 应该为 16。

整个过程如下：

TIME	主线程	0	1	2	3
	创建 4 个子线程	READY 0	READY 1	READY 2	READY 3
	★ 优先级降为 32	READY 0	READY 1	READY 2	READY 3
	READY 3	拥有 lock	READY 0	READY 1	READY 2
	READY 3	写 id	READY 0	READY 1	READY 2
	READY 3	READY 2	block	READY 0	READY 1
	READY 2	READY 1	waiter 0	block	READY 0



TIME 	READY 1	READY 0	waiter 0	waiter 1	block
	READY 1	释放 lock	READY 0	waiter 0	waiter 1
	READY 1	让出 CPU	READY 0	waiter 0	waiter 1
	READY 1	READY 0	拥有 lock	waiter 0	waiter 1
	READY 1	READY 0	写 id	waiter 0	waiter 1
	READY 1	block	READY 0	waiter 0	waiter 1
	READY 1	waiter 1	释放 lock	READY 0	waiter 0
	READY 1	waiter 1	让出 CPU	READY 0	waiter 0
	READY 1	waiter 1	READY 0	拥有 lock	waiter 0
	READY 1	waiter 1	READY 0	写 id	waiter 0
	READY 1	waiter 1	block	READY 0	
	READY 1	waiter 0	waiter 1	释放 lock	READY 0
	READY 1	waiter 0	waiter 1	让出 CPU	READY 0
	READY 1	waiter 0	waiter 1	READY 0	拥有 lock
	READY 1	waiter 0	waiter 1	READY 0	写 id
	READY 1	waiter 0	waiter 1	block	READY 0
	READY 1	READY 0	waiter 0	waiter 1	释放 lock
	READY 1	READY 0	waiter 0	waiter 1	让出 CPU
	READY 1	拥有 lock	waiter 0	waiter 1	READY 0
	READY 1	写 id	waiter 0	waiter 1	READY 0
	READY 1	READY 0	waiter 0	waiter 1	block
	...	...	...	...	...
	...	释放 lock	...	...	...
			释放 lock	...	...
	...			释放 lock	...
	...				释放 lock
	打印信息				

为了简化，这里只以 4 个子线程为例进行说明，16 个子线程的情况是类似的；同时，假设时间片(4 ticks)可以让一个线程执行完 lock\_acquire 和写 id 两条语句，其他情形也是类似的。

READY i 表示处于就绪状态的第 i 个线程，waiter i 表示处于 sema 的等待队列中的第 i 个线程，block 表示阻塞当前线程。时间是从上往下流逝的，每一行代表 1 个 tick，每一列代表 1 个线程。浅绿色的格子表示线程占用 CPU 时的操作，可以看到每一行，只有 1 个格子是浅绿色的。即每个 tick 只有 1 个线程可以使用 CPU。

可以看到，某个线程写完 id 后，由于达到了时间片的限制，要进行线程的切换。但是因为 lock 的存在，后面凡是想要获得 lock 的线程都被以此阻塞掉了，并且进入了 sema 的等待队列。这个时候，lock 的拥有者再度占有 CPU，完成后续操作。也就是释放 lock，并进行礼让。释放 lock 的语句，排在 sema 等待队列中的第 1 个线程进入就绪队列，在前面的线程礼让后，可以获得 CPU。如此继续下去，直到所有线程完成了所有次数的迭代。

观察表中每个线程“写 id”的操作，可以看到它们确实是每次迭代按照相同的次序进行的。

这个测试的关键就在于，创建完所有子线程后，主线程将自己的优先级降低，能否让优先级更高的子线程立刻获得 CPU。也就是需要修改函数 `thread_set_priority`。

### 结果分析：

(1) 修改之前，失败样例

```
Unexpected interrupt 0x0e (#PF Page-Fault Exception)
Unexpected interrupt 0x0e (#PF Page-Fault Exception)
Unexpected interrupt 0x0e (#PF Page-Fault Exception)
Unexpected interrupt 0x0e (#PF Page-Fault Exception)
Unexpected interrupt 0x0e (#PF Page-Fault Exception)
Unexpected interrupt 0x0e (#PF Page-Fault Exception)
Unexpected interrupt 0x0e (#PF Page-Fault Exception)
Unexpected interrupt 0x0e (#PF Page-Fault Exception)
Unexpected interrupt 0x0e (#PF Page-Fault Exception)
Unexpected interrupt 0x0e (#PF Page-Fault Exception)
Unexpected interrupt 0x0e (#PF Page-Fault Exception)
Unexpected interrupt 0x0e (#PF Page-Fault Exception)
ads/build$
```

因为此时还没有实现优先级抢占，所以主线程在将自己的优先级设为 31 后，还将继续使用 CPU，打印输出 buffer 的结果；由于子线程还没有机会执行，就造成了错误。

(2) 修改之后，成功样例

```
Executing 'priority-fifo':
(priority-fifo) begin
(priority-fifo) 16 threads will iterate 16 times in the same order each time.
(priority-fifo) If the order varies then there is a bug.
(priority-fifo) iteration: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
(priority-fifo) iteration: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
(priority-fifo) iteration: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
(priority-fifo) iteration: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
(priority-fifo) iteration: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
(priority-fifo) iteration: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
(priority-fifo) iteration: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
(priority-fifo) iteration: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
(priority-fifo) iteration: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
(priority-fifo) iteration: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
(priority-fifo) iteration: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
(priority-fifo) iteration: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
(priority-fifo) iteration: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
(priority-fifo) iteration: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
(priority-fifo) iteration: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
(priority-fifo) end
Execution of 'priority-fifo' complete.
sangna@16343065sang:~/pintos/src/threads/build$
```

可以看到在 16 次迭代中，每一次 16 个线程都是按相同的次序占用 CPU 的。

## priority-preempt

### 测试目的：

确保高优先级的线程的确会抢占 CPU。

### 过程分析：



- 在**主线程** `test_priority_preempt` 中:

```
ASSERT (thread_get_priority () == PRI_DEFAULT);
thread_create ("high-priority", PRI_DEFAULT + 1, simple_thread_func, NULL);
```

- 断言主线程的优先级为 32;
- 创建 1 个名为 "high-priority" , 优先级为 33 的子线程, 执行 `simple_thread_func`.

- **子线程** `simple_thread_func`:

```
int i;
for (i = 0; i < 5; i++)
{
    msg ("Thread %s iteration %d", thread_name (), i);
    thread_yield ();
}
msg ("Thread %s done!", thread_name ());
```

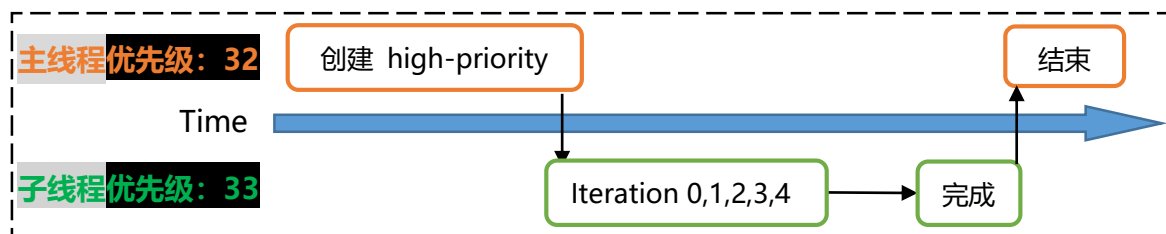
- For 循环 `i` 从 0 到 4 共 5 次: 打印消息 "线程 high-priority 迭代 `i`" , 礼让 CPU。因为子线程的优先级是高于主线程的, 所以这里的礼让是没有意义的。依然是子线程继续执行。
  - 打印消息 "线程 high-priority 完成! "
- 这个时候, 子线程已经完成了它的任务。主线程获得 CPU。

- **主线程** `test_priority_preempt`

```
msg ("The high-priority thread should have already completed.");
```

打印消息如上。

整个过程如下:



这个测试的关键在于:

主线程创建了更高优先级的子线程后, 应当让出 CPU, 也就是需要修改函数 `thread_create`。

## 结果分析:

(1) 修改之前, 失败样例

```
Executing 'priority-preempt':
(priority-preempt) begin
(priority-preempt) The high-priority thread should have already completed.
(priority-preempt) end
Execution of 'priority-preempt' complete.
(priority-preempt) Thread high-priority iteration 0
(priority-preempt) Thread high-priority iteration 1
(priority-preempt) Thread high-priority iteration 2
(priority-preempt) Thread high-priority iteration 3
(priority-preempt) Thread high-priority iteration 4
(priority-preempt) Thread high-priority done!
sangna@16343065sang:~/pintos/src/threads/build$
```

根据打印出来的信息，一直执行主线程，直到 end 之后才执行子线程，这是不对的。

(2) 修改之后，成功样例

```
Executing 'priority-preempt':  
(priority-preempt) begin  
(priority-preempt) Thread high-priority iteration 0  
(priority-preempt) Thread high-priority iteration 1  
(priority-preempt) Thread high-priority iteration 2  
(priority-preempt) Thread high-priority iteration 3  
(priority-preempt) Thread high-priority iteration 4  
(priority-preempt) Thread high-priority done!  
(priority-preempt) The high-priority thread should have already completed.  
(priority-preempt) end  
Execution of 'priority-preempt' complete.
```

可以看到线程是按照上面的流程图运行的。

## (二) 实验思路与代码分析

### ● 问题分析

综合上面对 3 个 test 的分析，总结如下：

- priority-fifo: 需要修改函数 `thread_set_priority`
- priority-preempt: 需要修改函数 `thread_create`
- priority-change: 需要修改函数 `thread_set_priority` 和函数 `thread_create`

这两个函数分别对应于发生优先级抢占的 2 种情形：

#### (1) `thread_set_priority`

正在占用 CPU 的线程重新设定自己的优先级，并且新的优先级低于在就绪队列中的优先级最高的线程。

#### (2) `thread_create`

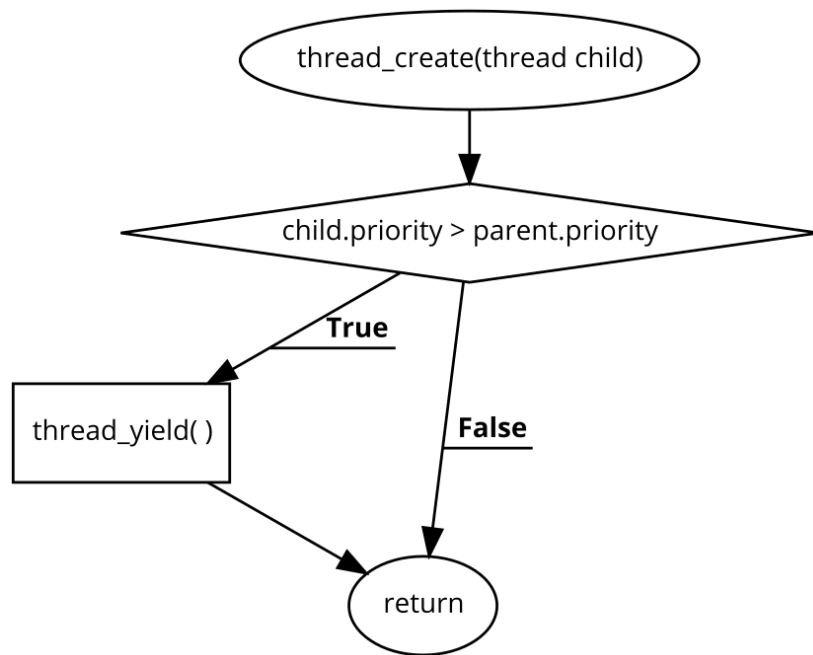
正在占用 CPU 的主线程创建了子线程，并且子线程的优先级高于主线程；

### ● 解决办法

#### ✧ 修改 `thread_create`

这个函数的实现方法我只想到了 1 种，就是在创建 1 个子线程 t，并将其 unblock 后，判断新创建的子线程的优先级是否高于创建它的主线程的优先级。

如果子线程的优先级高，就抢占 CPU，否则主线程继续占用 CPU。



代码可以有如下几种写法:

#### Version 0

```

if(t->priority > thread_get_priority())
{
    thread_yield();
} //lab2-add version 0
  
```

#### Version 1

```

thread_yield();
if(priority > thread_current()->priority)
{
    thread_yield();
} //lab2-add version 1
  
```

如果再排列组合一下, 可以再有两种写法, 不过我认为没有太大意义。本质上还是同一种思路想法。

#### Version 2

这种想法比较独特, 考虑到当前占用 CPU 的主线程是目前优先级最高的, 而 `thread_yield` 函数就是将当前占用 CPU 的线程重新有序地插入就绪队列并调度下一线程, 相当于队就绪队列重新排序, 所以可以不用判断条件, 直接调用 `thread_yield`。

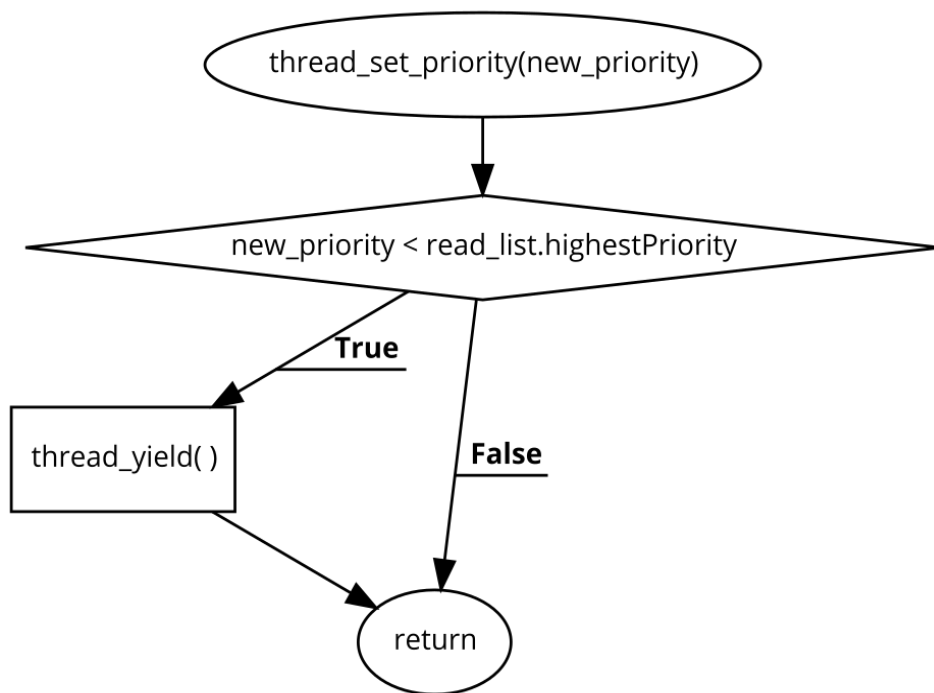
如果子线程的优先级高, 显然通过 `thread_yield` 可以获得 CPU;

如果子线程的优先级低, 那么主线程通过 `thread_yield` 中的 `schedule` 调度的还是它自己。

```

/* Add to run queue. */
thread_unblock (t);
thread_yield(); //lab2-add version 2
return tid;
  
```

✧ 修改 `thread_set_priority`



这个函数需要判断当前线程新设定的优先级，是否低于就绪队列中的最高优先级。如果新设定的优先级低于就绪队列中的最高优先级，高优先级的线程就抢占 CPU，否则继续执行当前线程。

不同的实现方法区别主要在于，如何获得就绪队列中优先级最高的线程代表 `elem`。

### Version 0

```

if(new_priority < list_entry(list_max(&ready_list,less,NULL),struct thread,elem)->priority)
{
    thread_yield();
} //lab2-add version 0
  
```

这里用到的 `list_max` 定义在 `list.c` 中

```

struct list_elem *
list_max (struct list *list, list_less_func *less, void *aux)
{
    struct list_elem *max = list_begin (list);
    if (max != list_end (list))
    {
        struct list_elem *e;

        for (e = list_next (max); e != list_end (list); e = list_next (e))
            if (less (max, e, aux))
                max = e;
    }
    return max;
}
  
```

可以看到它返回的是通过 `less` 函数比较后，指向 `list` 中某一特征“最大”的 `elem` 的指针。按照这一想法，只需要在定义比较函数 `less` 即可。

在 thread.c 中定义比较函数 less 如下:

```
//lab2-add: to compare threads' priority
static bool
less(const struct list_elem *a, const struct list_elem *b, void *aux UNUSED)
{
    struct thread *ta = list_entry(a, struct thread, elem);
    struct thread *tb = list_entry(b, struct thread, elem);
    return ta->priority < tb->priority;
}
```

如果 elem a 所代表的线程的优先级低于 elem b 所代表的线程的优先级, 返回 true, 否则返回 false。这样就可以得到就绪队列中最高的优先级是多少。

### Version 1

因为线程在插入就绪队列的时候是按照优先级排队的, 所以其实没有必要通过 list\_max 找到最高的优先级。因为很明显, 要么就绪队列为空, 要么就绪队列中的第一个 elem 所代表的线程就是优先级最高的线程。

为此在 list.c 中找到了可以返回队列第一个元素的函数:

```
struct list_elem *
list_front(struct list *list)
{
    ASSERT (!list_empty (list));
    return list->head.next;
}

71 struct list_elem *
72 list_begin (struct list *list)
73 {
74     ASSERT (list != NULL);
75     return list->head.next;
76 }
```

可以看到 list\_front 和 list\_begin 都可以返回 list 的第一个元素, 区别在于, list\_front 断言 list 不是空的, 即是有元素 elem 的; 而 list\_begin 则是断言指针 list 是指向某个实体的, 不是空指针。修改后的代码如下:

### Version 2

如果当前线程新的优先级低于就绪队列中的最高优先级, 显然通过 thread\_yield 可以让最高优先级的线程获得 CPU;

如果当前线程新的优先级高, 那么当前线程通过 thread\_yield 中的 schedule 调度的还是它自己;

```
// lab2-add version 1 ↓
if (!list_empty(&ready_list) &&
    new_priority < list_entry(list_begin(&ready_list), struct thread, elem)->priority)
{
    thread_yield();
}
```

所以可以不用判断条件, 直接调用 thread\_yield。

```
// lab2-add version 3 ↓
thread_yield();
```



### 3. 实验结果

```
16 of 27 tests failed.  
.../tests/Makefile:26: recipe for target 'check' failed  
make: *** [check] Error 1  
sangna@16343065sang:~/pintos/src/threads/build$
```

**结果分析：**之前是 19/27 test failed，也就是通过了 8 个测试，在本次实验后，又多通过了 3 个 test，所以是 16/27 tests failed。具体已在实验报告其他部分描述。

### 4. 回答问题

- (1) 如果没有考虑修改thread\_create函数的情况，test能通过吗？如果不能，会出现什么结果（请截图），解释为什么会出现这个结果。
- 答：如果没有考虑thread\_create的修改，priority-fifo测试可以通过，而priority-change和priority-preempt不可以通过。make check结果如下：



```
FAIL tests/threads/priority-change  
pass tests/threads/priority-fifo  
FAIL tests/threads/priority-preempt  
18 of 27 tests failed.
```

结合报告中关于test的分析，出现这个结果的原因是：

1. Priority-fifo不是通过thread\_create创建优先级更高的子线程来进行测试，而是创建完16个优先级低于自己的子线程后，通过thread\_set\_priority降低自己的优先级，使得子线程抢占CPU。所以在未修改thread\_create的时候，可以通过该测试。
2. Priority-change和priority-preempt都有通过thread\_create来创建优先级更高的子线程的部分，如果没有修改thread\_create，那么子线程在被创建后并不能立即获得CPU，而必须等待主线程经过一定的时间片以后，强行被插入回就绪队列，子线程才可以被调度。

- (2) 用自己的话阐述Pintos中的semaphore 和lock 的区别和联系

区别：① semaphore结构体中的数据成员为1个值value，表示某种资源的数量，1个等待队列waiter，里面存放了等待使用某种资源的线程；lock结构体中的数据成员为1个指向线程的指针holder，表示lock的拥有者，1个semaphore类型的结构体semaphore。

② 只要semaphore的value大于1，资源就可以被获取，同时value减1；如果value为0，

后续想要获得这个资源的线程就会被阻塞，排到waiter队列中；直到原本占用资源的某线程结束了它的操作，value+1；一旦value大于0，排在waiter队列中的第一个线程就被unlock，进入就绪状态，获得当前资源；lock中的semaphore的value值只能是1，也就是某种资源的数量只有1个。

③ 在操作上，lock是有拥有者的，并且它的操作包括lock\_acquire和lock\_release，在这两个操作中分别使用了semaphore的操作sema\_down和sema\_up。但是比sema\_down和sema\_up多了一个设定holder和释放holder的操作。

联系：①semaphore结构体是lock结构体的一个成员，在lock中，semaphore中的value被初始化为1，代表某种资源的数量为1；其实也就是保护某个线程间共享的资源在被1个线程使用时，其他线程无法使用。

②lock可以看成是value值为1，并且有holder的semaphore，主要是为了解决共享资源的保护问题；而semaphore还可以解决线程同步的问题，即安排线程按照某种次序执行各自的语句。

### ● (3) 考虑优先级抢占调度后，重新分析 alarm-priority

#### alarm-priority

##### 测试目的：

创建 10 个线程命名为 “priority p(i)”，(i=0,1,...,9)。其中，p(i)为线程 priority p(i) 的优先级，且  $p(i) = 30 - (i + 5) \% 10$ 。

##### 过程分析：

先在 alarm\_priority.c 中查看 test\_alarm\_priority 是如何进行测试的。

##### ● 主线程 test\_alarm\_priority

```
wake_time = timer_ticks () + 5 * TIMER_FREQ;
// 设置线程被唤醒时刻为当前时间 + 5 * 100
sema_init (&wait_sema, 0);
// 初始化semaphore结构体wait_sema
```

➤ 首先设置了线程应该被唤醒的时间 wake\_time;

```
for (i = 0; i < 10; i++)
{
    int priority = PRI_DEFAULT - (i + 5) % 10 - 1;
    /*      i = 0, 1, 2, 3, 4, 5, 6, 7, 8, 9
    priority = 25,24,23,22,21,30,29,28,27,26 */
    char name[16];
    snprintf (name, sizeof name, "priority %d", priority);
    thread_create (name, priority, alarm_priority_thread, NULL);
}
```

➤ 创建 10 个子线程，命名为 priority j(j=21,22,23,...,30)，它们的优先级和名字一致，执行 alarm\_priority\_thread;

➤ 但是，由于子线程的最高优先级仅为 30，低于主线程的 32，所以子线程们仅仅是被创建好，按照优先级的顺序在就绪队列里排好，并没有获得 CPU 执行；

```
thread_set_priority (PRI_MIN);
```

- 在执行创建 10 个线程的 for 循环之后，主线程将自己的优先级设为最低(0)，这样之前创建好的优先级最高的子线程就可以抢占 CPU 了。

- **子线程 alarm\_priority\_thread**

```
/* 忙等待直到当前的时刻改变 */
int64_t start_time = timer_ticks ();
while (timer_elapsed (start_time) == 0)
    continue;

/* 现在在一个timer tick的最开端，可以调用timer_sleep
而不用担心检查时间和系统中断之间的竞争 */
timer_sleep (wake_time - timer_ticks ());
```

- 将当前线程 block，调度下一个线程，直到 wake\_time 将其唤醒，插入就绪队列；
- 如此进行 10 次，直到 10 个子线程均进入阻塞状态；
- 由于休眠的时间足够长，主线程成为了就绪队列中的唯一线程，获得 CPU；

- **主线程 test\_alarm\_priority**

```
for (i = 0; i < 10; i++)
    sema_down (&wait_sema);
```

再回顾一下 sema\_up 和 sema\_down。

- ◆ sema\_up：若 sema 的等待队列不是空的，就 unblock 位于其等待队列第一位的 elem 所代表的线程；无论是否为空，都将 value 加 1。
- ◆ sema\_down：若 value 为 0，则调用 sema\_down 的线程将被 block，它的 elem 会被放到 sema 等待队列的末尾；当 value 不为 0 时，value 就减 1。

由于 wait\_sema 的 value 初始值为 0，所以调用了 sema\_down 的主线程就被阻塞掉了，此时 CPU 应该执行 idel\_thread 直到休眠的子线程按优先级顺序被执行。

- **子线程 alarm\_priority\_thread**

```
msg ("Thread %s woke up.", thread_name ());

sema_up (&wait_sema);
```

- 打印消息 "Thread pi 醒来了" ；
- 调用 sema\_up，在第 1 个子线程调用 sema\_up 的时候，由于 wait\_sema 不为空（当中有主线程在等待），所以主线程就被 unblock，进入就绪队列；
- 又由于主线程的优先级为 32，高于所有的子线程，所以，主线程获得 CPU；

- **主线程 test\_alarm\_priority**

```
for (i = 0; i < 10; i++)
    sema_down (&wait_sema);
```

- 主线程 sema\_down，由于这个时候 value 还是 0，所以主线程仍然满足 while 忙等待的条件，主线程又把自己 block 了；刚刚的 unblock 它的子线程继续运行。

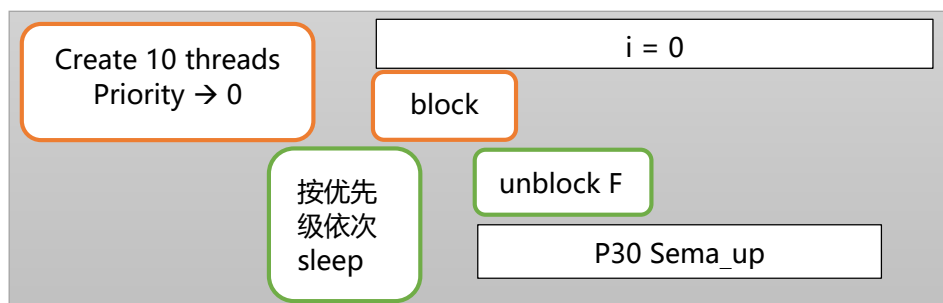
- **子线程 alarm\_priority\_thread**

- 还是在 sema\_up 中，将 value+1 变为 1；

- 子线程结束;
- 优先级次之的子线程获得 CPU;

如此重复 10 次, 直到所有的子线程被唤醒, 主线程才会结束。

整个过程大概是这样的:



所有子线程休眠后, 因为 value 值为 0, 主线程在  $i=0$ , 调用 `sema_down`, 将自己 `block`; 之后 CPU 一直运行 `idle_thread`, 直到优先级最高的子线程获得了 CPU。

**结果分析:** 线程按照正确的顺序执行。

```

Executing 'alarm-priority':
(alarm-priority) begin
(alarm-priority) Thread priority 30 woke up.
(alarm-priority) Thread priority 29 woke up.
(alarm-priority) Thread priority 28 woke up.
(alarm-priority) Thread priority 27 woke up.
(alarm-priority) Thread priority 26 woke up.
(alarm-priority) Thread priority 25 woke up.
(alarm-priority) Thread priority 24 woke up.
(alarm-priority) Thread priority 23 woke up.
(alarm-priority) Thread priority 22 woke up.
(alarm-priority) Thread priority 21 woke up.
(alarm-priority) end
Execution of 'alarm-priority' complete.
sangna@16343065sang: ~/pintos/src/threads/build$
  
```

分析完 `alarm-priority` 我不禁好奇, 这里是不是还有 1 种抢占的情景没有考虑? 某个线程正在运行, 它通过 `unblock` 将另外 1 个优先级高于它的线程插入了就绪队列, 但是并没有重新调度。因此, 高优先级的线程可能没有立即获得 CPU。我想应该在 `thread_unblock` 中做出相应的修改来解决这个问题。

## 5. 实验感想

本次实验内容比较简单, 耗时较少, 但是应该多想几种解决办法。由于我个人能力有限, 没有想出很多。另外有一些内容, 比如信号量、锁, 由于我在上一次实验中分析得比较多, 这次就轻松了一点。但是我还是有一些问题没有想明白, 特别是 `alarm-priority` 的 test 分析。我本来按照自己的理解画了  $i$  从 0 到 2 的过程, 但是按照这种循环规律画到  $i=9$  的时候发现不大对头。我想还是我理解的有偏差, 因为在子线程使用 `sema_up` 的时候, `unblock` 了主线程, 尽管主线程的优先级更高, 但是 `unblock` 中并没有抢占 CPU 的操作, 所以和我之前理解的情形就不一样了。因此,  $i=0$  之后的调度顺序具体究竟是什么样的, 我还是没有彻底弄明白。另外, 我发现自己比较擅长形象思维, 有些过程画出时间线, 就比较容易弄清楚整个流

程是什么样的。希望经过几次实验，我可以解决自己的疑惑，对整个架构有更正确、深入的理解。由于实验课内容和理论课内容有一定的不一致，所以还是需要自己提前学习一些基本知识，否则不是很容易弄懂实验内容。