laiy

Pintos-斯坦福大学操作系统Project详解-Project1

前言：

本实验来自斯坦福大学cs140课程，只限于教学用途，以下是他们对于Pintos系统的介绍：

> **Pintos is a simple operating system framework for the 80x86 architecture. It supports kernel threads, loading and running user programs, and a file system, but it implements all of these in a very simple way. In the Pintos projects, you and your project team will strengthen its support in all three of these areas. You will also add a virtual memory implementation.**

Pintos实验主要分成四部分，如下所示：

- 实验一：Thread
- 实验二：User Programs
- 实验三：Virtual Memory
- 实验四：File System

实验原理：

通过 bochs 加载 pintos 操作系统， 该操作系统会根据 pintos 的实现打印运行结果，通过比较标准输出文档和实际输出，来判断 pintos 实现是否符合要求。

环境配置：

> **参考： http://www.stanford.edu/class/cs140/projects/pintos/pintos_12.html#SEC166**

实验实现代码地址：

> **https://github.com/laiy/Pintos/tree/master/src**

实验一 THREAD：

我们试验一的最终任务就是在threads/中跑make check的时候， 27个test全pass。

Mission1：

重新实现timer_sleep函数（2.2.2）

（注意， 博主以下用了包括代码在内大概7000字的说明从每一个底层细节解析了这个函数的执行， 虽然很长但是让我们对pintos这个操作系统的各种机制和实现有更深刻的理解， 如果嫌长请直接跳到函数重新实现）

timer_sleep函数在devices/timer.c。系统现在是使用busy wait实现的，即线程不停地循环，直到时间片耗尽。更改timer_sleep的实现方式。

我们先来看一下devices目录下timer.c中的timer_sleep实现：

```
/* Sleeps for approximately TICKS timer ticks.  Interrupts must
   be turned on. */
void
timer_sleep (int64_t ticks)
{
  int64_t start = timer_ticks ();
  ASSERT (intr_get_level () == INTR_ON);
  while (timer_elapsed (start) < ticks)
    thread_yield();
}
```

让我们一行一行解析：

第6行： 调用了timer_ticks函数， 让我们来看看这个函数做了什么。

```
/* Returns the number of timer ticks since the OS booted. */
int64_t
timer_ticks (void)
{
  enum intr_level old_level = intr_disable ();
  int64_t t = ticks;
  intr_set_level (old_level);
  return t;
}
```

然后我们注意到这里有个intr_level的东西通过intr_disable返回了一个东西，没关系，我们继续往下找。

```
/* Interrupts on or off? */
enum intr_level
  {
    INTR_OFF,             /* Interrupts disabled. */
    INTR_ON               /* Interrupts enabled. */
  };
```

```
/* Disables interrupts and returns the previous interrupt status. */
enum intr_level
intr_disable (void)
{
  enum intr_level old_level = intr_get_level ();

  /* Disable interrupts by clearing the interrupt flag.
     See [IA32-v2b] "CLI" and [IA32-v3a] 5.8.1 "Masking Maskable
     Hardware Interrupts". */
  asm volatile ("cli" : : : "memory");

  return old_level;
}
```

这里很明显，intr_level代表能否被中断，而intr_disable做了两件事情：1. 调用intr_get_level() 2. 直接执行汇编代码，调用汇编指令来保证这个线程不能被中断。

注意： 这个asm volatile是在C语言中内嵌了汇编语言，调用了CLI指令，CLI指令不是command line interface, 而是clear interrupt, 作用是将标志寄存器的IF（interrupt flag）位置为0, IF=0时将不响应可屏蔽中断。

好，让我们继续来看intr_get_level又做了什么鬼。

```
/* Returns the current interrupt status. */
enum intr_level
intr_get_level (void)
{
  uint32_t flags;

  /* Push the flags register on the processor stack, then pop the
     value off the stack into `flags'.  See [IA32-v2b] "PUSHF"
     and "POP" and [IA32-v3a] 5.8.1 "Masking Maskable Hardware
     Interrupts". */
  asm volatile ("pushfl; popl %0" : "=g" (flags));

  return flags & FLAG_IF ? INTR_ON : INTR_OFF;
}
```

这里就是intr_disable函数调用的最深的地方了!

这个函数一样是调用了汇编指令，把标志寄存器的东西放到处理器栈上，然后把值pop到flags（代表标志寄存器IF位）上，通过判断flags来返回当前终端状态(intr_level)。

好， 到这里。 函数嵌套了这么多层， 我们整理一下逻辑：

1. intr_get_level返回了intr_level的值

2. intr_disable获取了当前的中断状态， 然后将当前中断状态改为不能被中断， 然后返回执行之前的中断状态。

有以上结论我们可以知道： timer_ticks第五行做了这么一件事情： 禁止当前行为被中断， 保存禁止被中断前的中断状态（用old_level储存）。

让我们再来看timer_ticks剩下的做了什么， 剩下的就是用t获取了一个全局变量ticks, 然后返回， 其中调用了set_level函数。

```
/* Enables or disables interrupts as specified by LEVEL and
   returns the previous interrupt status. */
enum intr_level
intr_set_level (enum intr_level level)
{
  return level == INTR_ON ? intr_enable () : intr_disable ();
}
```

有了之前的基础，这个函数就很容易看了， 如果之前是允许中断的（INTR_ON）则enable否则就disable.

而intr_enable正如你们所想，实现和之前基本一致：

```
/* Enables interrupts and returns the previous interrupt status. */
enum intr_level
intr_enable (void)
{
  enum intr_level old_level = intr_get_level ();
  ASSERT (!intr_context ());

  /* Enable interrupts by setting the interrupt flag.

     See [IA32-v2b] "STI" and [IA32-v3a] 5.8.1 "Masking Maskable
     Hardware Interrupts". */
  asm volatile ("sti");

  return old_level;
}
```

说明一下， sti指令就是cli指令的反面，将IF位置为1。

然后有个ASSERT断言了intr_context函数返回结果的false。

再来看intr_context

```
```

```
/* Returns true during processing of an external interrupt
   and false at all other times. */
bool
intr_context (void)
{
  return in_external_intr;
}
```

这里直接返回了是否外中断的标志in_external_intr， 就是说ASSERT断言这个中断不是外中断（IO等， 也称为硬中断）而是操作系统正常线程切换流程里的内中断（也称为软中断）。

好的， 至此， 我们总结一下：

这么多分析其实分析出了pintos操作系统如何利用中断机制来确保一个原子性的操作的。

我们来看， 我们已经分析完了timer_ticks这个函数， 它其实就是获取ticks的当前值返回而已， 而第5行和第7行做的其实只是确保这个过程是不能被中断的而已。

那么我们来达成一个共识， 被以下两个语句包裹的内容目的是为了保证这个过程不被中断。

```
enum intr_level old_level = intr_disable ();
...
intr_set_level (old_level);
```

好的， 那么ticks又是什么？ 来看ticks定义。

```
/* Number of timer ticks since OS booted. */
static int64_t ticks;
```

从pintos被启动开始， ticks就一直在计时， 代表着操作系统执行单位时间的前进计量。

好， 现在回过来看timer_sleep这个函数， start获取了起始时间， 然后断言必须可以被中断， 不然会一直死循环下去， 然后就是一个循环

```
while (timer_elapsed (start) < ticks)
  thread_yield();
```

注意这个ticks是函数的形参不是全局变量， 然后看一下这两个函数：

```
/* Returns the number of timer ticks elapsed since THEN, which
   should be a value once returned by timer_ticks(). */
int64_t
timer_elapsed (int64_t then)
{
  return timer_ticks () - then;
}
```

很明显timer_elapsed返回了当前时间距离then的时间间隔， 那么这个循环实质就是在ticks的时间内不断执行thread_yield。

那么我们最后来看thread_yield是什么就可以了：

```
/* Yields the CPU.  The current thread is not put to sleep and
   may be scheduled again immediately at the scheduler's whim. */
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_push_back (&ready_list, &cur->elem);
  cur->status = THREAD_READY;
  schedule ();
  intr_set_level (old_level);
}
```

第6行thread_current函数做的事情已经可以顾名思义了， 不过具有钻研精神和强迫症的你还是要确定它的具体实现：

```
/* Returns the running thread.
   This is running_thread() plus a couple of sanity checks.
   See the big comment at the top of thread.h for details. */
struct thread *
thread_current (void)
{
  struct thread *t = running_thread ();

  /* Make sure T is really a thread.
     If either of these assertions fire, then your thread may
     have overflowed its stack.  Each thread has less than 4 kB
     of stack, so a few big automatic arrays or moderate
     recursion can cause stack overflow. */
  ASSERT (is_thread (t));
  ASSERT (t->status == THREAD_RUNNING);

  return t;
}
```

```
/* Returns the running thread. */
struct thread *
running_thread (void)
{
  uint32_t *esp;

  /* Copy the CPU's stack pointer into `esp', and then round that
     down to the start of a page.  Because `struct thread' is
     always at the beginning of a page and the stack pointer is
     somewhere in the middle, this locates the curent thread. */
  asm ("mov %%esp, %0" : "=g" (esp));
  return pg_round_down (esp);
}
```

```
/* Returns true if T appears to point to a valid thread. */
static bool
is_thread (struct thread *t)
{
  return t != NULL && t->magic == THREAD_MAGIC;
}
```

先来看thread_current调用的running_thread, 把CPU栈的指针复制到esp中， 然后调用pg_round_down

```
/* Round down to nearest page boundary. */
static inline void *pg_round_down (const void *va) {
  return (void *) ((uintptr_t) va & ~PGMASK);
}
```

好，这里又涉及到这个操作系统是怎么设计页面的了：

```
/* Page offset (bits 0:12). */
#define PGSHIFT 0                          /* Index of first offset bit. */
#define PGBITS  12                         /* Number of offset bits. */
#define PGSIZE  (1 << PGBITS)              /* Bytes in a page. */
#define PGMASK  BITMASK(PGSHIFT, PGBITS)   /* Page offset bits (0:12). */
```

```
/* Functions and macros for working with virtual addresses.

   See pte.h for functions and macros specifically for x86
   hardware page tables. */

#define BITMASK(SHIFT, CNT) (((1ul << (CNT)) - 1) << (SHIFT))
```

一个页面12位， PGMASK调用BITMASK其实就是一个页面全部位都是1的这么个MASK， 注意1ul的意思是unsigned long的1。

然后来看pg_round_down， 对PGMASK取反的结果就是一个页面大小全部为0的这么个数， 然后和传过来的指针做与操作的结果就是清0指针的靠右12位。

这里有什么效果呢？ 我们知道一个页面12位， 而struct thread是在一个页面的最开始的， 所以对任何一个页面的指针做pg_round_down的结果就是返回到这个页面最开始线程结构体的位置。

好， 我们现在分析出了pg_round_down其实就是返回了这个页面线程的最开始指针， 那么running_thread的结果返回当前线程起始指针。

再来看thread_current里最后的两个断言， 一个断言t指针是一个线程， 一个断言这个线程处于THREAD_RUNNING状态。

然后is_thread用的t->magic其实是用于检测时候有栈溢出的这么个元素。

```
1 /* Owned by thread.c. */
2 unsigned magic;                    /* Detects stack overflow. */
```

好， 现在thread_current分析完了， 这个就是返回当前线程起始指针位置。

我们继续看thread_yield， 然后剩下的很多东西其实我们已经分析过了， 在分析的过程其实是对这个操作系统工作过程的剖析， 很多地方都是相通的。

第9断言这是个软中断， 第11和16包裹起来的就是我们之前分析的线程机制保证的一个原子性操作。

然后我们来看12-15做了什么：

```
1 if (cur != idle_thread)
2   list_push_back (&ready_list, &cur->elem);
3 cur->status = THREAD_READY;
4 schedule ();
```

如何当前线程不是空闲的线程就调用list_push_back把当前线程的元素扔到就绪队列里面， 并把线程改成THREAD_READY状态。

关于队列list的相关操作mission2会涉及到， 这里先不作解释， 顾名思义即可。

然后再调用schedule:

```
 1 /* Schedules a new process.  At entry, interrupts must be off and
 2    the running process's state must have been changed from
 3    running to some other state.  This function finds another
 4    thread to run and switches to it.
 5
 6    It's not safe to call printf() until thread_schedule_tail()
 7    has completed. */
 8 static void
 9 schedule (void)
10 {
11   struct thread *cur = running_thread ();
12   struct thread *next = next_thread_to_run ();
13   struct thread *prev = NULL;
14
15   ASSERT (intr_get_level () == INTR_OFF);
16   ASSERT (cur->status != THREAD_RUNNING);
17   ASSERT (is_thread (next));
18
19   if (cur != next)
20     prev = switch_threads (cur, next);
21   thread_schedule_tail (prev);
22 }
```

首先获取当前线程cur和调用next_thread_to_run获取下一个要run的线程：

```
 1 /* Chooses and returns the next thread to be scheduled.  Should
 2    return a thread from the run queue, unless the run queue is
 3    empty.  (If the running thread can continue running, then it
 4    will be in the run queue.)  If the run queue is empty, return
 5    idle_thread. */
 6 static struct thread *
 7 next_thread_to_run (void)
 8 {
 9   if (list_empty (&ready_list))
10     return idle_thread;
11   else
12     return list_entry (list_pop_front (&ready_list), struct thread, elem);
13 }
```

如果就绪队列空闲直接返回一个空闲线程指针， 否则拿就绪队列第一个线程出来返回。

然后3个断言之前讲过就不多说了， 确保不能被中断， 当前线程是RUNNING_THREAD等。

如果当前线程和下一个要跑的线程不是同一个的话调用switch_threads返回给prev。

```
/* Switches from CUR, which must be the running thread, to NEXT,
   which must also be running switch_threads(), returning CUR in
   NEXT's context. */
struct thread *switch_threads (struct thread *cur, struct thread *next);
```

注意， 这个函数实现是用汇编语言实现的在threads/switch.S里：

```
#### struct thread *switch_threads (struct thread *cur, struct thread *next);
####
#### Switches from CUR, which must be the running thread, to NEXT,
#### which must also be running switch_threads(), returning CUR in
#### NEXT's context.
####
#### This function works by assuming that the thread we're switching
#### into is also running switch_threads().  Thus, all it has to do is
#### preserve a few registers on the stack, then switch stacks and
#### restore the registers.  As part of switching stacks we record the
#### current stack pointer in CUR's thread structure.

.globl switch_threads
.func switch_threads
switch_threads:
    # Save caller's register state.
    #
    # Note that the SVR4 ABI allows us to destroy %eax, %ecx, %edx,
    # but requires us to preserve %ebx, %ebp, %esi, %edi.  See
    # [SysV-ABI-386] pages 3-11 and 3-12 for details.
    #
    # This stack frame must match the one set up by thread_create()
    # in size.
    pushl %ebx
    pushl %ebp
    pushl %esi
    pushl %edi

    # Get offsetof (struct thread, stack).
.globl thread_stack_ofs
    mov thread_stack_ofs, %edx

    # Save current stack pointer to old thread's stack, if any.
    movl SWITCH_CUR(%esp), %eax
    movl %esp, (%eax,%edx,1)

    # Restore stack pointer from new thread's stack.
    movl SWITCH_NEXT(%esp), %ecx
    movl (%ecx,%edx,1), %esp

    # Restore caller's register state.
    popl %edi
    popl %esi
    popl %ebp
    popl %ebx
        ret
.endfunc
```

分析一下这个汇编代码： 先4个寄存器压栈保存寄存器状态（保护作用）， 这4个寄存器是switch_threads_frame的成员：

```
/* switch_thread()'s stack frame. */
struct switch_threads_frame
  {
    uint32_t edi;               /*  0: Saved %edi. */
    uint32_t esi;               /*  4: Saved %esi. */
    uint32_t ebp;               /*  8: Saved %ebp. */
    uint32_t ebx;               /* 12: Saved %ebx. */
```

```
8     void (*eip) (void);         /* 16: Return address. */
9     struct thread *cur;         /* 20: switch_threads()'s CUR argument. */
10    struct thread *next;        /* 24: switch_threads()'s NEXT argument. */
11  };
```

然后全局变量thread_stack_ofs记录线程和栈之间的间隙， 我们都知道线程切换有个保存现场的过程，

来看34,35行， 先把当前的线程指针放到eax中， 并把线程指针保存在相对基地址偏移量为edx的地址中。

38,39： 切换到下一个线程的线程栈指针， 保存在ecx中， 再把这个线程相对基地址偏移量edx地址（上一次保存现场的时候存放的）放到esp当中继续执行。

这里ecx, eax起容器的作用， edx指向当前现场保存的地址偏移量。

简单来说就是保存当前线程状态， 恢复新线程之前保存的线程状态。

然后再把4个寄存器拿出来， 这个是硬件设计要求的， 必须保护switch_threads_frame里面的寄存器才可以destroy掉eax, edx, ecx。

然后注意到现在eax(函数返回值是eax)就是被切换的线程栈指针。

我们由此得到一个结论， schedule先把当前线程丢到就绪队列，然后把线程切换如果下一个线程和当前线程不一样的话。

然后再看shedule最后一行的函数thread_schedule_tail做了什么鬼， 这里参数prev是NULL或者在下一个线程的上下文中的当前线程指针。

```
1 /* Completes a thread switch by activating the new thread's page
2    tables, and, if the previous thread is dying, destroying it.
3
4    At this function's invocation, we just switched from thread
5    PREV, the new thread is already running, and interrupts are
6    still disabled.  This function is normally invoked by
7    thread_schedule() as its final action before returning, but
8    the first time a thread is scheduled it is called by
9    switch_entry() (see switch.S).
10
11   It's not safe to call printf() until the thread switch is
12   complete.  In practice that means that printf()s should be
13   added at the end of the function.
14
15   After this function and its caller returns, the thread switch
16   is complete. */
17 void
18 thread_schedule_tail (struct thread *prev)
19 {
20   struct thread *cur = running_thread ();
21
22   ASSERT (intr_get_level () == INTR_OFF);
23
24   /* Mark us as running. */
25   cur->status = THREAD_RUNNING;
26
27   /* Start new time slice. */
28   thread_ticks = 0;
29
30 #ifdef USERPROG
31   /* Activate the new address space. */
32   process_activate ();
33 #endif
34
35   /* If the thread we switched from is dying, destroy its struct
36      thread.  This must happen late so that thread_exit() doesn't
37      pull out the rug under itself.  (We don't free
38      initial_thread because its memory was not obtained via
39      palloc().) */
40   if (prev != NULL && prev->status == THREAD_DYING && prev != initial_thread)
41     {
42       ASSERT (prev != cur);
43       palloc_free_page (prev);
44     }
45 }
```

先是获得当前线程cur, 注意此时是已经切换过的线程了（或者还是之前run的线程， 因为ready队列为空）。

然后把线程状态改成THREAD_RUNNING， 然后thread_ticks清零开始新的线程切换时间片。

然后调用process_activate触发新的地址空间。

```
/* Sets up the CPU for running user code in the current
   thread.
   This function is called on every context switch. */
void
process_activate (void)
{
  struct thread *t = thread_current ();

  /* Activate thread's page tables. */
  pagedir_activate (t->pagedir);

  /* Set thread's kernel stack for use in processing
     interrupts. */
  tss_update ();
}
```

这里先是拿到当前线程， 调用pagedir_activate:

```
/* Loads page directory PD into the CPU's page directory base
   register. */
void
pagedir_activate (uint32_t *pd)
{
  if (pd == NULL)
    pd = init_page_dir;

  /* Store the physical address of the page directory into CR3
     aka PDBR (page directory base register).  This activates our
     new page tables immediately.  See [IA32-v2a] "MOV--Move
     to/from Control Registers" and [IA32-v3a] 3.7.5 "Base
     Address of the Page Directory". */
  asm volatile ("movl %0, %%cr3" : : "r" (vtop (pd)) : "memory");
}
```

这个汇编指令将当前线程的页目录指针存储到CR3（页目录表物理内存基地址寄存器）中，也就是说这个函数更新了现在的页目录表。

最后来看tss_update：

```
/* Sets the ring 0 stack pointer in the TSS to point to the end
   of the thread stack. */
void
tss_update (void)
{
  ASSERT (tss != NULL);
  tss->esp0 = (uint8_t *) thread_current () + PGSIZE;
}
```

首先要弄清楚tss是什么， tss是task state segment， 叫任务状态段， 任务（进程）切换时的任务现场信息。

这里其实是把TSS的一个栈指针指向了当前线程栈的尾部， 也就是更新了任务现场的信息和状态。

好， 到现在process_activate分析完了， 总结一下： 其实就是做了2件事情： 1.更新页目录表 2.更新任务现场信息（TSS）

我们现在继续来看thread_schedule_tail， 最后是这4行：

```
  /* If the thread we switched from is dying, destroy its struct
     thread.  This must happen late so that thread_exit() doesn't
     pull out the rug under itself.  (We don't free
     initial_thread because its memory was not obtained via
     palloc().) */
  if (prev != NULL && prev->status == THREAD_DYING && prev != initial_thread)
```

```
7     {
8       ASSERT (prev != cur);
9       palloc_free_page (prev);
10    }
```

这里是如果我们切换的线程状态是THREAD_DYING（代表欲要销毁的线程）的话， 调用palloc_free_page：

```
/* Frees the page at PAGE. */
void
palloc_free_page (void *page)
{
  palloc_free_multiple (page, 1);
}
```

```
/* Frees the PAGE_CNT pages starting at PAGES. */
void
palloc_free_multiple (void *pages, size_t page_cnt)
{
  struct pool *pool;
  size_t page_idx;

  ASSERT (pg_ofs (pages) == 0);
  if (pages == NULL || page_cnt == 0)
    return;

  if (page_from_pool (&kernel_pool, pages))
    pool = &kernel_pool;
  else if (page_from_pool (&user_pool, pages))
    pool = &user_pool;
  else
    NOT_REACHED ();

  page_idx = pg_no (pages) - pg_no (pool->base);

#ifndef NDEBUG
  memset (pages, 0xcc, PGSIZE * page_cnt);
#endif

  ASSERT (bitmap_all (pool->used_map, page_idx, page_cnt));
  bitmap_set_multiple (pool->used_map, page_idx, page_cnt, false);
}
```

这里创建了一个pool的结构体：

```
/* A memory pool. */
struct pool
  {
    struct lock lock;                   /* Mutual exclusion. */
    struct bitmap *used_map;            /* Bitmap of free pages. */
    uint8_t *base;                      /* Base of pool. */
  };
```

首先palloc实现的是一个页分配器， 这里pool的角色就是记忆分配的内容。 这里结构体用位图记录空的页， 关键是这里又有一个操作系统很重要的知识概念出现了，就是lock：

```
/* Lock. */
struct lock
  {
    struct thread *holder;      /* Thread holding lock (for debugging). */
    struct semaphore semaphore; /* Binary semaphore controlling access. */
  };
```

然后锁其实是由二值信号量实现的：

```
/* A counting semaphore. */
struct semaphore
```

```
3   {
4     unsigned value;             /* Current value. */
5     struct list waiters;        /* List of waiting threads. */
6   };
```

具体信号量方法实现在threads/synch.c中， 这里不作更多讲解了， 毕竟函数分析还没涉及到这里。

继续看palloc_free_multiple， 第8行其实就是截取后12位， 即获得当前页偏差量， 断言为0就是说页指针应该指向线程结构体

```
1 /* Offset within a page. */
2 static inline unsigned pg_ofs (const void *va) {
3   return (uintptr_t) va & PGMASK;
4 }
```

然后分析12-17行， 这里要弄清楚一点是系统memory分成2个池， 一个是kernel pool, 一个是user pool， user pool是提供给用户页的， 别的都是kernel pool。

然后看下这里调用的page_from_pool函数：

```
1 /* Returns true if PAGE was allocated from POOL,
2    false otherwise. */
3 static bool
4 page_from_pool (const struct pool *pool, void *page)
5 {
6   size_t page_no = pg_no (page);
7   size_t start_page = pg_no (pool->base);
8   size_t end_page = start_page + bitmap_size (pool->used_map);
9
10   return page_no >= start_page && page_no < end_page;
11 }
```

pg_no是获取虚拟页数的， 方法其实就是直接指针右移12位就行了：

```
1 /* Virtual page number. */
2 static inline uintptr_t pg_no (const void *va) {
3   return (uintptr_t) va >> PGBITS;
4 }
```

然后这里获取当前池中的的起始页和结束页位置， 然后判断页面时候在这个池的Number范围之类来判断时候属于某个池。

再看NOT_REACHED函数,这个函数博主找了半天， 最后用全文件搜索才找着在哪，在lib/debug.h中：

```
1 /* This is outside the header guard so that debug.h may be
2    included multiple times with different settings of NDEBUG. */
3 #undef ASSERT
4 #undef NOT_REACHED
5
6 #ifndef NDEBUG
7 #define ASSERT(CONDITION)                                       \
8         if (CONDITION) { } else {                               \
9                 PANIC ("assertion `%s' failed.", #CONDITION);   \
10         }
11 #define NOT_REACHED() PANIC ("executed an unreachable statement");
12 #else
13 #define ASSERT(CONDITION) ((void) 0)
14 #define NOT_REACHED() for (;;)
15 #endif /* lib/debug.h */
```

```
1 /* GCC lets us add "attributes" to functions, function
2    parameters, etc. to indicate their properties.
3    See the GCC manual for details. */
4 #define UNUSED __attribute__ ((unused))
5 #define NO_RETURN __attribute__ ((noreturn))
6 #define NO_INLINE __attribute__ ((noinline))
7 #define PRINTF_FORMAT(FMT, FIRST) __attribute__ ((format (printf, FMT, FIRST)))
8
9 /* Halts the OS, printing the source file name, line number, and
```

```
10    function name, plus a user-specific message. */
11 #define PANIC(...) debug_panic (__FILE__, __LINE__, __func__, __VA_ARGS__)
12
13 void debug_panic (const char *file, int line, const char *function,
14                   const char *message, ...) PRINTF_FORMAT (4, 5) NO_RETURN;
```

这里根据NDEBUG状态分两种define， 一个是ASSERT空函数， NOT_REACHED执行死循环， 一个是如果ASSERT参数CONDITION为false的话就调用PANIC输出文件，行数，函数名和用户信息， NOT_REACHED也会输出信息。

有些童鞋在跑测试的时候会出现卡在一个地方不动的状态， 其实不是因为你电脑的问题， 而是当一些错误触发NOT_REACHED之类的问题的时候， 因为非debug环境就一直执行死循环了， 反映出来的行为就是命令行卡住不动没有输出。

注意这里的语法类似__attribute__和((format(printf, m , n)))是面向gcc编译器处理的写法， 这里做的事情其实是参数声明和调用匹配性检查。

好， 继续来看palloc_free_multiple， 用page_idx保存了计算出来了页id， 清空了页指针， 然后还剩下最后两行：

```
1   ASSERT (bitmap_all (pool->used_map, page_idx, page_cnt));
2   bitmap_set_multiple (pool->used_map, page_idx, page_cnt, false);
```

第一个断言：

```
1 /* Returns true if every bit in B between START and START + CNT,
2    exclusive, is set to true, and false otherwise. */
3 bool
4 bitmap_all (const struct bitmap *b, size_t start, size_t cnt)
5 {
6   return !bitmap_contains (b, start, cnt, false);
7 }
```

```
 1 /* Returns true if any bits in B between START and START + CNT,
 2    exclusive, are set to VALUE, and false otherwise. */
 3 bool
 4 bitmap_contains (const struct bitmap *b, size_t start, size_t cnt, bool value)
 5 {
 6   size_t i;
 7
 8   ASSERT (b != NULL);
 9   ASSERT (start <= b->bit_cnt);
10   ASSERT (start + cnt <= b->bit_cnt);
11
12   for (i = 0; i < cnt; i++)
13     if (bitmap_test (b, start + i) == value)
14       return true;
15   return false;
16 }
```

bitmap_contains首先做断言对参数正确性确认， 然后如果所有位处于start到start+cnt都是value的话， 别的都是~value的话， 返回true， 从我们的函数调用来看就是断言位图全是0。

```
1 /* Returns the value of the bit numbered IDX in B. */
2 bool
3 bitmap_test (const struct bitmap *b, size_t idx)
4 {
5   ASSERT (b != NULL);
6   ASSERT (idx < b->bit_cnt);
7   return (b->bits[elem_idx (idx)] & bit_mask (idx)) != 0;
8 }
9
```

```
 1 /* Returns the index of the element that contains the bit
 2    numbered BIT_IDX. */
 3 static inline size_t
```

```
4 elem_idx (size_t bit_idx)
5 {
6   return bit_idx / ELEM_BITS;
7 }
8
9 /* Returns an elem_type where only the bit corresponding to
10    BIT_IDX is turned on. */
11 static inline elem_type
12 bit_mask (size_t bit_idx)
13 {
14   return (elem_type) 1 << (bit_idx % ELEM_BITS);
15 }
```

来看bit_test的实现， 这里直接返回某一位的具体值。

这里直接用elem_idx获取idx对应的index取出位， 然后和bit_mask做与操作， bit_mask就是返回了一个只有idx位是1其他都是0的一个数， 也就是说idx必须为1才返回true对bit_test来说， 否则false。

好， 至此， 对palloc_free_multiple只剩一行了：

```
1   bitmap_set_multiple (pool->used_map, page_idx, page_cnt, false);
```

```
/* Sets the CNT bits starting at START in B to VALUE. */
void
bitmap_set_multiple (struct bitmap *b, size_t start, size_t cnt, bool value)
{
  size_t i;

  ASSERT (b != NULL);
  ASSERT (start <= b->bit_cnt);
  ASSERT (start + cnt <= b->bit_cnt);

  for (i = 0; i < cnt; i++)
    bitmap_set (b, start + i, value);
}
```

这里对位图所有位都做了bitmap_set设置：

```
1 /* Atomically sets the bit numbered IDX in B to VALUE. */
2 void
3 bitmap_set (struct bitmap *b, size_t idx, bool value)
4 {
5   ASSERT (b != NULL);
6   ASSERT (idx < b->bit_cnt);
7   if (value)
8     bitmap_mark (b, idx);
9   else
10    bitmap_reset (b, idx);
11 }
```

很明显这里mark就是设为1, reset就是置为0。

来看一下实现：

```
1 /* Atomically sets the bit numbered BIT_IDX in B to true. */
2 void
3 bitmap_mark (struct bitmap *b, size_t bit_idx)
4 {
5   size_t idx = elem_idx (bit_idx);
6   elem_type mask = bit_mask (bit_idx);
7
8   /* This is equivalent to `b->bits[idx] |= mask' except that it
9      is guaranteed to be atomic on a uniprocessor machine.  See
10     the description of the OR instruction in [IA32-v2b]. */
11  asm ("orl %1, %0" : "=m" (b->bits[idx]) : "r" (mask) : "cc");
12 }
```

```
13
14 /* Atomically sets the bit numbered BIT_IDX in B to false. */
15 void
16 bitmap_reset (struct bitmap *b, size_t bit_idx)
17 {
18   size_t idx = elem_idx (bit_idx);
19   elem_type mask = bit_mask (bit_idx);
20
21   /* This is equivalent to `b->bits[idx] &= ~mask' except that it
22      is guaranteed to be atomic on a uniprocessor machine.  See
23      the description of the AND instruction in [IA32-v2a]. */
24   asm ("andl %1, %0" : "=m" (b->bits[idx]) : "r" (~mask) : "cc");
25 }
```

一样， 最底层的实现依然是用汇编语言实现的， 两个汇编语言实现的就是两个逻辑： 1. b->bits[idx] |= mask 2. b->bits[idx] &= ~mask， 这里mask都是只有idx位为1, 其他为0的mask。

好， 到现在位置palloc_free_multiple已经分析完了， 整理一下逻辑：

其实就是把页的位图全部清0了， 清0代表这这个页表的所有页都是free的， 等于清空了页目录表中的所有页面。

逻辑继续向上回溯：

thread_schedule_tail其实就是获取当前线程， 分配恢复之前执行的状态和现场， 如果当前线程死了就清空资源。

schedule其实就是拿下一个线程切换过来继续run。

thread_yield其实就是把当前线程扔到就绪队列里， 然后重新schedule， 注意这里如果ready队列为空的话当前线程会继续在cpu执行。

最后回溯到我们最顶层的函数逻辑： timer_sleep就是在ticks时间内， 如果线程处于running状态就不断把他扔到就绪队列不让他执行。

好的， 至此我们对原来的timer_sleep的实现方式有了十分清楚的理解了， 我们也很清楚的看到了它的缺点：

线程依然不断在cpu就绪队列和running队列之间来回， 占用了cpu资源， 这并不是我们想要的， 我们希望用一种唤醒机制来实现这个函数。

函数重新实现：

实现思路： 调用timer_sleep的时候直接把线程阻塞掉，然后给线程结构体加一个成员ticks_blocked来记录这个线程被sleep了多少时间， 然后利用操作系统自身的时钟中断（每个tick会执行一次）加入对线程状态的检测， 每次检测将ticks_blocked减1, 如果减到0就唤醒这个线程。

具体代码：

```
1 /* Sleeps for approximately TICKS timer ticks.  Interrupts must
2    be turned on. */
3 void
4 timer_sleep (int64_t ticks)
5 {
6   if (ticks <= 0)
7   {
8     return;
9   }
10  ASSERT (intr_get_level () == INTR_ON);
11  enum intr_level old_level = intr_disable ();
12  struct thread *current_thread = thread_current ();
13  current_thread->ticks_blocked = ticks;
14  thread_block ();
15  intr_set_level (old_level);
16 }
```

注意这里调用的thread_block：

```
1 /* Puts the current thread to sleep.  It will not be scheduled
2    again until awoken by thread_unblock().
3
4    This function must be called with interrupts turned off.  It
5    is usually a better idea to use one of the synchronization
6    primitives in synch.h. */
```

```
void
thread_block (void)
{
  ASSERT (!intr_context ());
  ASSERT (intr_get_level () == INTR_OFF);

  thread_current ()->status = THREAD_BLOCKED;
  schedule ();
}
```

线程的各种原理之前分析都已经剖析过了， 不作过多解释。

给线程的结构体加上我们的ticks_blocked成员：

```
    /* Record the time the thread has been blocked. */
    int64_t ticks_blocked;
```

然后在线程被创建的时候初始化ticks_blocked为0， 加在thread_create函数内：

```
  t->ticks_blocked = 0;
```

然后修改时钟中断处理函数， 加入线程sleep时间的检测， 加在timer_interrupt内：

```
  thread_foreach (blocked_thread_check, NULL);
```

这里的thread_foreach就是对每个线程都执行blocked_thread_check这个函数：

```
/* Invoke function 'func' on all threads, passing along 'aux'.
   This function must be called with interrupts off. */
void
thread_foreach (thread_action_func *func, void *aux)
{
  struct list_elem *e;

  ASSERT (intr_get_level () == INTR_OFF);

  for (e = list_begin (&all_list); e != list_end (&all_list);
       e = list_next (e))
    {
      struct thread *t = list_entry (e, struct thread, allelem);
      func (t, aux);
    }
}
```

aux就是传给这个函数的参数。

然后， 给thread添加一个方法blocked_thread_check即可：

thread.h中声明：

```
void blocked_thread_check (struct thread *t, void *aux UNUSED);
```

thread.c：

```
/* Check the blocked thread */
void
blocked_thread_check (struct thread *t, void *aux UNUSED)
{
  if (t->status == THREAD_BLOCKED && t->ticks_blocked > 0)
  {
      t->ticks_blocked--;
      if (t->ticks_blocked == 0)
      {
          thread_unblock(t);
      }
  }
}
```

thread_unblock就是把线程丢到就绪队列里继续跑：

```
/* Transitions a blocked thread T to the ready-to-run state.
   This is an error if T is not blocked.  (Use thread_yield() to
   make the running thread ready.)

   This function does not preempt the running thread.  This can
   be important: if the caller had disabled interrupts itself,
   it may expect that it can atomically unblock a thread and
   update other data. */
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_push_back (&ready_list, &t->elem);
  t->status = THREAD_READY;
  intr_set_level (old_level);
}
```

好的， 这样timer_sleep函数唤醒机制就实现了。

然后跑测试结果会是这样的：

```
pass tests/threads/alarm-single
pass tests/threads/alarm-multiple
pass tests/threads/alarm-simultaneous
FAIL tests/threads/alarm-priority
pass tests/threads/alarm-zero
pass tests/threads/alarm-negative
```

好， 我们还需要pass掉alarm_priority我们这个实验一就算完成了，继续搞~

Mission2:

实现优先级调度（2.2.3）

先来分析一下线程， 线程成员本身就有一个priority。

```
struct thread
  {
    /* Owned by thread.c. */
    tid_t tid;                          /* Thread identifier. */
    enum thread_status status;          /* Thread state. */
    char name[16];                      /* Name (for debugging purposes). */
    uint8_t *stack;                     /* Saved stack pointer. */
    int priority;                       /* Priority. */
    struct list_elem allelem;           /* List element for all threads list. */

    /* Shared between thread.c and synch.c. */
    struct list_elem elem;              /* List element. */

#ifdef USERPROG
    /* Owned by userprog/process.c. */
    uint32_t *pagedir;                  /* Page directory. */
#endif

    /* Owned by thread.c. */
    unsigned magic;                     /* Detects stack overflow. */

    /* Record the time the thread has been blocked. */
    int64_t ticks_blocked;
  };
```

然后priority的约束：

```
/* Thread priorities. */
#define PRI_MIN 0                       /* Lowest priority. */
#define PRI_DEFAULT 31                  /* Default priority. */
#define PRI_MAX 63                      /* Highest priority. */
```

我们之前分析timer_sleep的时候其实已经对线程的调度有了非常深刻的剖析了，这里实现优先级调度的核心思想就是： 维持就绪队列为一个优先级队列。

换一种说法： 我们在插入线程到就绪队列的时候保证这个队列是一个优先级队列即可。

那么我们在什么时候会把一个线程丢到就绪队列中呢?

1. thread_unblock

2. init_thread

3. thread_yield

那么我们只要在扔的时候维持这个就绪队列是优先级队列即可。

我们来看： thread_unblock现在丢进队列里是这么干的：

```
  list_push_back (&ready_list, &t->elem);
```

这个是直接扔到队列尾部了， 我们并不希望这么做， 于是我们只要改一下这个扔的动作就可以了，因为调度的时候下一个thread是直接取队头的。

那么我们先来研究一下pintos的队列是如何搞的，在/lib/kernel/：

```
/* List element. */
struct list_elem
  {
    struct list_elem *prev;     /* Previous list element. */
    struct list_elem *next;     /* Next list element. */
  };

/* List. */
struct list
  {
    struct list_elem head;      /* List head. */
    struct list_elem tail;      /* List tail. */
  };
```

很常见的队列数据结构， 继续研究一下现在队列有哪些有用的函数可以帮助我们实现优先级队列：

然后喜大普奔地发现了一些神奇的函数：

```
/* Operations on lists with ordered elements. */
void list_sort (struct list *,
                list_less_func *, void *aux);
void list_insert_ordered (struct list *, struct list_elem *,
                          list_less_func *, void *aux);
void list_unique (struct list *, struct list *duplicates,
                  list_less_func *, void *aux);
```

list_insert_ordered不就是为我们这里的实现量身订造的么！搞起！！

```
/* Inserts ELEM in the proper position in LIST, which must be
   sorted according to LESS given auxiliary data AUX.
   Runs in O(n) average case in the number of elements in LIST. */
void
list_insert_ordered (struct list *list, struct list_elem *elem,
                     list_less_func *less, void *aux)
{
  struct list_elem *e;

  ASSERT (list != NULL);
  ASSERT (elem != NULL);
  ASSERT (less != NULL);

```

```
14   for (e = list_begin (list); e != list_end (list); e = list_next (e))
15     if (less (elem, e, aux))
16       break;
17   return list_insert (e, elem);
18 }
```

直接修改thread_unblock函数把list_push_back改成：

```
1   list_insert_ordered (&ready_list, &t->elem, (list_less_func *) &thread_cmp_priority, NULL);
```

然后实现一下比较函数thread_cmp_priority：

```
1 /* priority compare function. */
2 bool
3 thread_cmp_priority (const struct list_elem *a, const struct list_elem *b, void *aux UNUSED)
4 {
5   return list_entry(a, struct thread, elem)->priority > list_entry(b, struct thread, elem)->priority;
6 }
```

然后对thread_yield和thread_init里的list_push_back作同样的修改：

init_thread:

```
1   list_insert_ordered (&all_list, &t->allelem, (list_less_func *) &thread_cmp_priority, NULL);
```

thread_yield:

```
1     list_insert_ordered (&ready_list, &cur->elem, (list_less_func *) &thread_cmp_priority, NULL);
```

做完这些工作之后， 就兴高采烈地发现alarm_priority这个测试pass了。

```
pass tests/threads/alarm-single
pass tests/threads/alarm-multiple
pass tests/threads/alarm-simultaneous
pass tests/threads/alarm-priority
pass tests/threads/alarm-zero
pass tests/threads/alarm-negative
```

好的， 我们实现了一个优先级队列， 但是因为这个优先级机制还会引出一系列的问题， 就是我们接下来要解决的问题。

下面让我们来pass这两个test：

priority-change

priority-preempt

我们直接TDD吧， 测试驱动开发， 来看测试做了什么：

```
1 void
2 test_priority_preempt (void)
3 {
4   /* This test does not work with the MLFQS. */
5   ASSERT (!thread_mlfqs);
6
7   /* Make sure our priority is the default. */
8   ASSERT (thread_get_priority () == PRI_DEFAULT);
9
10  thread_create ("high-priority", PRI_DEFAULT + 1, simple_thread_func, NULL);
11  msg ("The high-priority thread should have already completed.");
12 }
13
14 static void
15 simple_thread_func (void *aux UNUSED)
16 {
17   int i;
18
19   for (i = 0; i < 5; i++)
20     {
21       msg ("Thread %s iteration %d", thread_name (), i);
22       thread_yield ();
23     }
24   msg ("Thread %s done!", thread_name ());
25 }
```

先分析一下抢占式调度的测试， 其实就是在创建一个线程的时候， 如果线程高于当前线程就先执行创建的线程。

然后下面这个测试是基于抢占式调度的基础之上做的测试：

```
void
test_priority_change (void)
{
  /* This test does not work with the MLFQS. */
  ASSERT (!thread_mlfqs);

  msg ("Creating a high-priority thread 2.");
  thread_create ("thread 2", PRI_DEFAULT + 1, changing_thread, NULL);
  msg ("Thread 2 should have just lowered its priority.");
  thread_set_priority (PRI_DEFAULT - 2);
  msg ("Thread 2 should have just exited.");
}

static void
changing_thread (void *aux UNUSED)
{
  msg ("Thread 2 now lowering priority.");
  thread_set_priority (PRI_DEFAULT - 1);
  msg ("Thread 2 exiting.");
}
```

而测试希望的输出结果是这样的：

```
Acceptable output:
  (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
Differences in `diff -u' format:
  (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
```

所以我们得出一个结论就是： 测试线程(我们称为thread1)创建了一个PRI_DEFAULT+1优先级的内核线程thread2，然后由于thread2优先级高，

所以线程执行直接切换到thread2， thread1阻塞， 然后thread2执行的时候调用的是changing_thread， 又把自身优先级调为PRI_DEFAULT-1，

这个时候thread1的优先级就大于thread2了， 此时thread2阻塞于最后一个msg输出， 线程切换到thread1， 然后thread1又把自己优先级改成PRI_DEFAULT-2，

这个时候thread2又高于thread1了， 所以执行thread2， 然后在输出thread1的msg， 于是整个过程就有了图中的测试输出结果。

分析这个测试行为我们得出的结论就是： 在设置一个线程优先级要立即重新考虑所有线程执行顺序， 重新安排执行顺序。

好， 弄清楚思路实现其实就非常简单了， 直接在线程设置优先级的时候调用thread_yield即可， 这样就把当前线程重新丢到就绪队列中继续执行， 保证了执行顺序。

此外， 还有在创建线程的时候， 如果新创建的线程比主线程优先级高的话也要调用thread_yield。

实现代码：

```
/* Sets the current thread's priority to NEW_PRIORITY. */
void
thread_set_priority (int new_priority)
{
  thread_current ()->priority = new_priority;
  thread_yield ();
}
```

然后在thread_create最后把创建的线程unblock了之后加上这些代码：

```
1   if (thread_current ()->priority < priority)
2   {
3     thread_yield ();
4   }
```

然后就搞定了，总共就加了5行代码。

可能有人会想2个test总共才加了5行代码，这个会不会太简单了。

其实不会，实现这个解决方案的前提是对这个调度机制有清楚的认识，对测试有充分完整的分析。

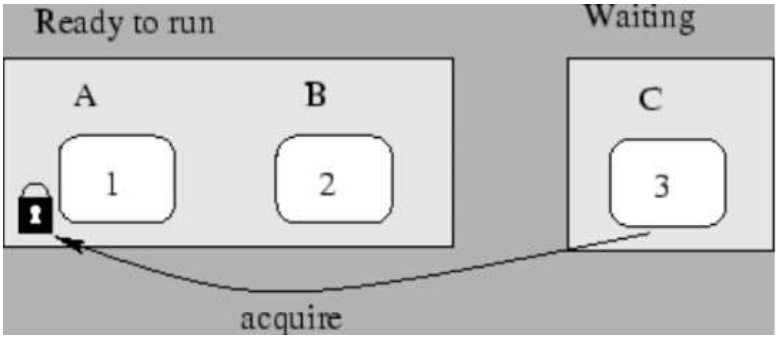
来看测试结果：

```
pass tests/threads/alarm-single
pass tests/threads/alarm-multiple
pass tests/threads/alarm-simultaneous
pass tests/threads/alarm-priority
pass tests/threads/alarm-zero
pass tests/threads/alarm-negative
pass tests/threads/priority-change
FAIL tests/threads/priority-donate-one
FAIL tests/threads/priority-donate-multiple
FAIL tests/threads/priority-donate-multiple2
FAIL tests/threads/priority-donate-nest
FAIL tests/threads/priority-donate-sema
FAIL tests/threads/priority-donate-lower
pass tests/threads/priority-fifo
pass tests/threads/priority-preempt
```

然后就兴高采烈的把priority_change和priority_preempt给pass了，好我们来继续搞别的测试~

接下来我们来把priority-priority-*给过了，这个测试是优先级捐赠的测试，下面我们来分析一下：

这个是一个优先级翻转(priority inversion)问题:



线程 A,B,C 分别具有 1,2,3 优先级(数字越大说明优先级越高), 线程 A,B 目前在就绪队列中等待调度, 线程 A 对一个互斥资源拥有线程锁。而此时, 高优先级的线程 C 也想要访问这个互斥资源, 线程 C 只好在这个资源上等待,不能进入就绪队列。当调度器开始调度时, 它只能从 A 和 B 中进行选择,根据优先级调度原理,线程 B 将会首先运行。

这时就产生了一个问题, 即本来线程 C 优先级比线程 B 高, 但是线程 B 却先运行了,从而产生了优先级翻转问题。(这里复制了课程TA的原话)

怎么解决这个问题?

当发现高优先级的任务因为低优先级任务占用资源而阻塞时，就将低优先级任务的优先级提升到等待它所占有的资源的最高优先级任务的优先级。

继续TDD走起， 先来看priority-donate-one测试代码：

```
1 void
2 test_priority_donate_one (void)
3 {
4   struct lock lock;
5
6   /* This test does not work with the MLFQS. */
7   ASSERT (!thread_mlfqs);
8
9   /* Make sure our priority is the default. */
10  ASSERT (thread_get_priority () == PRI_DEFAULT);
11
12  lock_init (&lock);
13  lock_acquire (&lock);
14  thread_create ("acquire1", PRI_DEFAULT + 1, acquire1_thread_func, &lock);
```

```
15   msg ("This thread should have priority %d.  Actual priority: %d.",
16        PRI_DEFAULT + 1, thread_get_priority ());
17   thread_create ("acquire2", PRI_DEFAULT + 2, acquire2_thread_func, &lock);
18   msg ("This thread should have priority %d.  Actual priority: %d.",
19        PRI_DEFAULT + 2, thread_get_priority ());
20   lock_release (&lock);
21   msg ("acquire2, acquire1 must already have finished, in that order.");
22   msg ("This should be the last line before finishing this test.");
23 }
24
25 static void
26 acquire1_thread_func (void *lock_)
27 {
28   struct lock *lock = lock_;
29
30   lock_acquire (lock);
31   msg ("acquire1: got the lock");
32   lock_release (lock);
33   msg ("acquire1: done");
34 }
35
36 static void
37 acquire2_thread_func (void *lock_)
38 {
39   struct lock *lock = lock_;
40
41   lock_acquire (lock);
42   msg ("acquire2: got the lock");
43   lock_release (lock);
44   msg ("acquire2: done");
45 }
```

分析： 首先当前线程（称为original_thread）是一个优先级为PRI_DEFAULT的线程， 然后第4行创建了一个锁， 接着创建一个线程acquire1，优先级为PRI_DEFAULT+1, 传了一个参数为这个锁的函数过去（线程acquire1执行的时候会调用）。

好， 我们之前实现的抢占式调度会让acquire1马上执行， 来看acquire1_thread_func干了什么， 这里直接获取了这个锁， 来看lock_acquire函数：

```
1 /* Acquires LOCK, sleeping until it becomes available if
2    necessary.  The lock must not already be held by the current
3    thread.
4
5    This function may sleep, so it must not be called within an
6    interrupt handler.  This function may be called with
7    interrupts disabled, but interrupts will be turned back on if
8    we need to sleep. */
9 void
10 lock_acquire (struct lock *lock)
11 {
12   ASSERT (lock != NULL);
13   ASSERT (!intr_context ());
14   ASSERT (!lock_held_by_current_thread (lock));
15
16   sema_down (&lock->semaphore);
17   lock->holder = thread_current ();
18 }
```

这里如我们所想， 直接调用信号量PV操作中的P操作， 来看P操作sema_down：

```
1 /* Down or "P" operation on a semaphore.  Waits for SEMA's value
2    to become positive and then atomically decrements it.
3
4    This function may sleep, so it must not be called within an
5    interrupt handler.  This function may be called with
6    interrupts disabled, but if it sleeps then the next scheduled
7    thread will probably turn interrupts back on. */
8 void
9 sema_down (struct semaphore *sema)
10 {
```

```
11   enum intr_level old_level;
12
13   ASSERT (sema != NULL);
14   ASSERT (!intr_context ());
15
16   old_level = intr_disable ();
17   while (sema->value == 0)
18     {
19       list_push_back (&sema->waiters, &thread_current ()->elem);
20       thread_block ();
21     }
22   sema->value--;
23   intr_set_level (old_level);
24 }
```

和课上描述的一致， 把线程丢到这个信号量的队列waiters里， 阻塞该线程等待唤醒， value--。

注意， 这里acquire1_thread_func阻塞了， msg这个时候并不会输出， 这时会继续执行original_thread, 然后输出msg， 输出当前线程应该的优先级和实际的优先级。

然后继续创建一个线程acquire2, 优先级为PRI_DEFAULT+2， 这里调用和acquire1一致， 然后original_thread继续输出msg。

好， 然后original_thread释放了这个锁（V操作）， 释放的过程会触发被锁着的线程acquire1, acquire2， 然后根据优先级调度， 先执行acquire2, 再acquire1, 最后再执行original_thread。

那么这里应该是acquire2, acquire1分别释放锁然后输出msg， 最后original_thread再输出msg。

好， 我们已经把这个测试程序分析完了， 我们来看它希望的输出：

```
1 # -*- perl -*-
2 use strict;
3 use warnings;
4 use tests::tests;
5 check_expected ([<<'EOF']);
6 (priority-donate-one) begin
7 (priority-donate-one) This thread should have priority 32.  Actual priority: 32.
8 (priority-donate-one) This thread should have priority 33.  Actual priority: 33.
9 (priority-donate-one) acquire2: got the lock
10 (priority-donate-one) acquire2: done
11 (priority-donate-one) acquire1: got the lock
12 (priority-donate-one) acquire1: done
13 (priority-donate-one) acquire2, acquire1 must already have finished, in that order.
14 (priority-donate-one) This should be the last line before finishing this test.
15 (priority-donate-one) end
16 EOF
17 pass;
```

输出行为和我们分析的一致， 来看7,8行， original_thread的优先级分别变成了PRI_DEFAULT+1和PRI_DEFAULT+2。

我们来根据这个结果分析一下优先级捐赠行为：

original_thread拥有的锁被acquire1获取之后， 因为acquire1线程被阻塞于这个锁， 那么acquire1的执行必须要original_thread继续执行释放这个锁， 从优先级的角度来说， original_thread的优先级应该提升到acquire1的优先级，

因为original_thread本身的执行包含了acquire1执行的阻塞， 所以此时acquire1对original_thread做了捐赠， 优先级提到PRI_DEFAULT+1， acquire2行为类似。

好， 支持priority-donate-one分析结束， 我们来分析一下实现：

具体行为肯定是被锁定在了锁的获取和释放上了， 我们的实现思路是：

在一个线程获取一个锁的时候， 如果拥有这个锁的线程优先级比自己低就提高它的优先级，然后在这个线程释放掉这个锁之后把原来拥有这个锁的线程改回原来的优先级。

好， 这里先不急着写代码， 继续来分析其他测试， 全部分析完了再写代码， 因为这些测试都是优先级捐献相关的行为， 我们全部分析完了再写就避免了走弯路了。

来分析测试priority-donte-multiple:

```
1 void
2 test_priority_donate_multiple (void)
3 {
4   struct lock a, b;
5
6   /* This test does not work with the MLFQS. */
7   ASSERT (!thread_mlfqs);
8
9   /* Make sure our priority is the default. */
10  ASSERT (thread_get_priority () == PRI_DEFAULT);
11
12  lock_init (&a);
13  lock_init (&b);
14
15  lock_acquire (&a);
16  lock_acquire (&b);
17
18  thread_create ("a", PRI_DEFAULT + 1, a_thread_func, &a);
19  msg ("Main thread should have priority %d.  Actual priority: %d.",
20       PRI_DEFAULT + 1, thread_get_priority ());
21
22  thread_create ("b", PRI_DEFAULT + 2, b_thread_func, &b);
23  msg ("Main thread should have priority %d.  Actual priority: %d.",
24       PRI_DEFAULT + 2, thread_get_priority ());
25
26  lock_release (&b);
27  msg ("Thread b should have just finished.");
28  msg ("Main thread should have priority %d.  Actual priority: %d.",
29       PRI_DEFAULT + 1, thread_get_priority ());
30
31  lock_release (&a);
32  msg ("Thread a should have just finished.");
33  msg ("Main thread should have priority %d.  Actual priority: %d.",
34       PRI_DEFAULT, thread_get_priority ());
35 }
36
37 static void
38 a_thread_func (void *lock_)
39 {
40  struct lock *lock = lock_;
41
42  lock_acquire (lock);
43  msg ("Thread a acquired lock a.");
44  lock_release (lock);
45  msg ("Thread a finished.");
46 }
47
48 static void
49 b_thread_func (void *lock_)
50 {
51  struct lock *lock = lock_;
52
53  lock_acquire (lock);
54  msg ("Thread b acquired lock b.");
55  lock_release (lock);
56  msg ("Thread b finished.");
57 }
```

一样， original_thread是优先级为PRI_DEFAULT的线程， 然后创建2个锁， 接着创建优先级为PRI_DEFAULT+1的线程a， 把锁a丢给这个线程的执行函数。

这时候线程a抢占式地调用a_thread_func， 获取了a这个锁， 阻塞。

然后original_thread输出线程优先级的msg。

然后再创建一个线程优先级为PRI_DEFAULT+2的线程b， 和a一样做同样的操作。

好， 然后original_thread释放掉了锁b， 此时线程b被唤醒， 抢占式执行b_thread_func。

然后original再输出msg， a同上， 此时我们来看一下测试希望的输出是什么：

```
# -*- perl -*-
use strict;
use warnings;
use tests::tests;
check_expected ([<<'EOF']);
(priority-donate-multiple) begin
(priority-donate-multiple) Main thread should have priority 32.  Actual priority: 32.
(priority-donate-multiple) Main thread should have priority 33.  Actual priority: 33.
(priority-donate-multiple) Thread b acquired lock b.
(priority-donate-multiple) Thread b finished.
(priority-donate-multiple) Thread b should have just finished.
(priority-donate-multiple) Main thread should have priority 32.  Actual priority: 32.
(priority-donate-multiple) Thread a acquired lock a.
(priority-donate-multiple) Thread a finished.
(priority-donate-multiple) Thread a should have just finished.
(priority-donate-multiple) Main thread should have priority 31.  Actual priority: 31.
(priority-donate-multiple) end
EOF
pass;
```

好， 这里输出和我们的分析依然是一致的。 重点在于original_thread的优先级变化， 第一次输出是正常的， priority-donate-one已经测试了这个逻辑。

这里特别的是original_thread拥有两把锁分别给a, b两个线程占有了。

后面是释放了b之后， original_thread的优先级恢复到32, 即当前线程的优先级还是被a的优先级所捐赠着的，最后释放了a之后才回到原来的优先级。

这里测试的行为实际是： 多锁情况下优先级逻辑的正确性。

那么我们对应的实现思路是： 释放一个锁的时候， 将该锁的拥有者改为该线程被捐赠的第二优先级，若没有其余捐赠者， 则恢复原始优先级。

那么我们的线程必然需要一个数据结构来记录所有对这个线程有捐赠行为的线程。

继续来看priority-donate-multiple2这个测试：

```
void
test_priority_donate_multiple2 (void)
{
  struct lock a, b;

  /* This test does not work with the MLFQS. */
  ASSERT (!thread_mlfqs);

  /* Make sure our priority is the default. */
  ASSERT (thread_get_priority () == PRI_DEFAULT);

  lock_init (&a);
  lock_init (&b);

  lock_acquire (&a);
  lock_acquire (&b);

  thread_create ("a", PRI_DEFAULT + 3, a_thread_func, &a);
  msg ("Main thread should have priority %d.  Actual priority: %d.",
       PRI_DEFAULT + 3, thread_get_priority ());

  thread_create ("c", PRI_DEFAULT + 1, c_thread_func, NULL);

  thread_create ("b", PRI_DEFAULT + 5, b_thread_func, &b);
  msg ("Main thread should have priority %d.  Actual priority: %d.",
       PRI_DEFAULT + 5, thread_get_priority ());

  lock_release (&a);
  msg ("Main thread should have priority %d.  Actual priority: %d.",
       PRI_DEFAULT + 5, thread_get_priority ());

  lock_release (&b);
  msg ("Threads b, a, c should have just finished, in that order.");
  msg ("Main thread should have priority %d.  Actual priority: %d.",
       PRI_DEFAULT, thread_get_priority ());
```

```
36 }
37
38 static void
39 a_thread_func (void *lock_)
40 {
41   struct lock *lock = lock_;
42
43   lock_acquire (lock);
44   msg ("Thread a acquired lock a.");
45   lock_release (lock);
46   msg ("Thread a finished.");
47 }
48
49 static void
50 b_thread_func (void *lock_)
51 {
52   struct lock *lock = lock_;
53
54   lock_acquire (lock);
55   msg ("Thread b acquired lock b.");
56   lock_release (lock);
57   msg ("Thread b finished.");
58 }
59
60 static void
61 c_thread_func (void *a_ UNUSED)
62 {
63   msg ("Thread c finished.");
64 }
```

有了之前的分析这里简单说一下： original_thread拥有2个锁， 然后创建PRI_DEFAULT+3的线程a去拿a这个锁， PRI_DEFAULT+5的线程b去拿b这个锁，中间创建了一个PRI_DEFAULT+1的c线程， 但是因为创建的时候当前线程的优先级已经被a线程捐赠了所以抢占调度并没有发生。 然后分别释放掉a和b，释放a的时候线程a被唤醒， 但是优先级依然不如当前线程， 此时当前线程优先级仍然被b捐赠着， 优先级最高继续执行， 然后释放掉b， 释放掉b之后，original_thread的优先级降到初始，应该最后被调用， 线程b抢占调度， 然后线程a， 再是线程c， 最后才original_thread输出msg。

于是就有了以下的输出断言：

```
1 # -*- perl -*-
2 use strict;
3 use warnings;
4 use tests::tests;
5 check_expected ([<<'EOF']);
6 (priority-donate-multiple2) begin
7 (priority-donate-multiple2) Main thread should have priority 34.  Actual priority: 34.
8 (priority-donate-multiple2) Main thread should have priority 36.  Actual priority: 36.
9 (priority-donate-multiple2) Main thread should have priority 36.  Actual priority: 36.
10 (priority-donate-multiple2) Thread b acquired lock b.
11 (priority-donate-multiple2) Thread b finished.
12 (priority-donate-multiple2) Thread a acquired lock a.
13 (priority-donate-multiple2) Thread a finished.
14 (priority-donate-multiple2) Thread c finished.
15 (priority-donate-multiple2) Threads b, a, c should have just finished, in that order.
16 (priority-donate-multiple2) Main thread should have priority 31.  Actual priority: 31.
17 (priority-donate-multiple2) end
18 EOF
19 pass;
```

这里依然测试的是多锁情况下优先级逻辑的正确性。

再来看测试priority-donate-nest：

```
1 void
2 test_priority_donate_nest (void)
3 {
4   struct lock a, b;
5   struct locks locks;
```

```
6
7   /* This test does not work with the MLFQS. */
8   ASSERT (!thread_mlfqs);
9
10  /* Make sure our priority is the default. */
11  ASSERT (thread_get_priority () == PRI_DEFAULT);
12
13  lock_init (&a);
14  lock_init (&b);
15
16  lock_acquire (&a);
17
18  locks.a = &a;
19  locks.b = &b;
20  thread_create ("medium", PRI_DEFAULT + 1, medium_thread_func, &locks);
21  thread_yield ();
22  msg ("Low thread should have priority %d.  Actual priority: %d.",
23       PRI_DEFAULT + 1, thread_get_priority ());
24
25  thread_create ("high", PRI_DEFAULT + 2, high_thread_func, &b);
26  thread_yield ();
27  msg ("Low thread should have priority %d.  Actual priority: %d.",
28       PRI_DEFAULT + 2, thread_get_priority ());
29
30  lock_release (&a);
31  thread_yield ();
32  msg ("Medium thread should just have finished.");
33  msg ("Low thread should have priority %d.  Actual priority: %d.",
34       PRI_DEFAULT, thread_get_priority ());
35 }
36
37 static void
38 medium_thread_func (void *locks_)
39 {
40  struct locks *locks = locks_;
41
42  lock_acquire (locks->b);
43  lock_acquire (locks->a);
44
45  msg ("Medium thread should have priority %d.  Actual priority: %d.",
46       PRI_DEFAULT + 2, thread_get_priority ());
47  msg ("Medium thread got the lock.");
48
49  lock_release (locks->a);
50  thread_yield ();
51
52  lock_release (locks->b);
53  thread_yield ();
54
55  msg ("High thread should have just finished.");
56  msg ("Middle thread finished.");
57 }
58
59 static void
60 high_thread_func (void *lock_)
61 {
62  struct lock *lock = lock_;
63
64  lock_acquire (lock);
65  msg ("High thread got the lock.");
66  lock_release (lock);
67  msg ("High thread finished.");
68 }
```

注意， original_thread只获取了锁a， 它并不拥有锁b。

这里创建了一个locks的结构体装着2个锁a和b， 然后创建medium线程， 优先级为PRI_DEFAULT+1, 把这个locks作为线程medium执行函数的参数。

然后抢占调用medium_thread_func， 此时这个函数获取b这个锁， 此时medium成为锁b的拥有者， 并不阻塞， 继续执行， 然后medium在获取锁a的时候阻塞了。

此时original_thread继续跑， 它的优先级被medium提到了PRI_DEFAULT+1, 输出优先级Msg。

然后创建优先级为PRI_DEFAULT+2的high线程， 抢占调用high_thread_func， 然后这里high拿到了b这个锁， 而b的拥有者是medium， 阻塞， 注意， 这里medium被high捐赠了， 优先级到PRI_DEFAULT+2, 此时original_thread也应该一样提到同样优先级。

然后original_thread输出一下优先级msg之后释放掉锁a， 释放出发了medium_thread_func抢占调用， 输出此时优先级为PRI_DEFAULT+2， 然后medium释放掉a, 释放掉b， 释放b的时候被high_thread_func抢占， high输出完之后medium继续run， 输出两句之后再到original_thread， 输出两句msg完事。

按照这个逻辑， 它的希望输出也是和我们分析的过程一样：

```
# -*- perl -*-
use strict;
use warnings;
use tests::tests;
check_expected ([<<'EOF']);
(priority-donate-nest) begin
(priority-donate-nest) Low thread should have priority 32.  Actual priority: 32.
(priority-donate-nest) Low thread should have priority 33.  Actual priority: 33.
(priority-donate-nest) Medium thread should have priority 33.  Actual priority: 33.
(priority-donate-nest) Medium thread got the lock.
(priority-donate-nest) High thread got the lock.
(priority-donate-nest) High thread finished.
(priority-donate-nest) High thread should have just finished.
(priority-donate-nest) Middle thread finished.
(priority-donate-nest) Medium thread should just have finished.
(priority-donate-nest) Low thread should have priority 31.  Actual priority: 31.
(priority-donate-nest) end
EOF
pass;
```

这个测试是一个优先级嵌套问题， 重点在于medium拥有的锁被low阻塞， 在这个前提下high再去获取medium的说阻塞的话， 优先级提升具有连环效应，就是medium被提升了， 此时它被锁捆绑的low线程应该跟着一起提升。

从实现的角度来说， 我们线程又需要加一个数据结构， 我们需要获取这个线程被锁于哪个线程。

好的， 再看下一个测试priority-donate-sema:

```
void
test_priority_donate_sema (void)
{
  struct lock_and_sema ls;

  /* This test does not work with the MLFQS. */
  ASSERT (!thread_mlfqs);

  /* Make sure our priority is the default. */
  ASSERT (thread_get_priority () == PRI_DEFAULT);

  lock_init (&ls.lock);
  sema_init (&ls.sema, 0);
  thread_create ("low", PRI_DEFAULT + 1, l_thread_func, &ls);
  thread_create ("med", PRI_DEFAULT + 3, m_thread_func, &ls);
  thread_create ("high", PRI_DEFAULT + 5, h_thread_func, &ls);
  sema_up (&ls.sema);
  msg ("Main thread finished.");
}

static void
l_thread_func (void *ls_)
{
  struct lock_and_sema *ls = ls_;

  lock_acquire (&ls->lock);
  msg ("Thread L acquired lock.");
  sema_down (&ls->sema);
```

```
29   msg ("Thread L downed semaphore.");
30   lock_release (&ls->lock);
31   msg ("Thread L finished.");
32 }
33
34 static void
35 m_thread_func (void *ls_)
36 {
37   struct lock_and_sema *ls = ls_;
38
39   sema_down (&ls->sema);
40   msg ("Thread M finished.");
41 }
42
43 static void
44 h_thread_func (void *ls_)
45 {
46   struct lock_and_sema *ls = ls_;
47
48   lock_acquire (&ls->lock);
49   msg ("Thread H acquired lock.");
50
51   sema_up (&ls->sema);
52   lock_release (&ls->lock);
53   msg ("Thread H finished.");
54 }
```

lock_and_sema是包含一个锁和一个信号量的结构体， 初始化信号量为0, 然后创建PRI_DEFAULT+1的线程low, 获取ls内的锁成为拥有者， 然后sema_down（P）阻塞。

然后创建PRI_DEFAULT+3的线程med， 这里也直接调用sema_down(P)阻塞了。

最后创建PRI_DEFAULT+5的线程high, 这里获取锁， 阻塞。

然后回到original_thread， 调用V操作， 此时唤醒了l_thread_func， 因为low被high捐献了优先级高于med， 然后l_thread_func跑， 释放掉了锁。

此时直接触发h_thread_func， 输出， 然后V操作， 释放掉锁， 由于优先级最高所以V操作之后不会被抢占， 这个函数跑完之后m_thread_func开始跑（被V唤醒）， 然后输出一句完事， 再到l_thread_func中输出最后一句回到original_thread。

这里包含了信号量和锁混合触发， 实际上还是信号量在起作用， 因为锁是由信号量实现的。

所以输出是这样的：

```
# -*- perl -*-
use strict;
use warnings;
use tests::tests;
check_expected ([<<'EOF']);
(priority-donate-sema) begin
(priority-donate-sema) Thread L acquired lock.
(priority-donate-sema) Thread L downed semaphore.
(priority-donate-sema) Thread H acquired lock.
(priority-donate-sema) Thread H finished.
(priority-donate-sema) Thread M finished.
(priority-donate-sema) Thread L finished.
(priority-donate-sema) Main thread finished.
(priority-donate-sema) end
EOF
pass;
```

再来看priority-donate-lower:

```
void
test_priority_donate_lower (void)
{
  struct lock lock;

  /* This test does not work with the MLFQS. */
```

```
  ASSERT (!thread_mlfqs);

  /* Make sure our priority is the default. */
  ASSERT (thread_get_priority () == PRI_DEFAULT);

  lock_init (&lock);
  lock_acquire (&lock);
  thread_create ("acquire", PRI_DEFAULT + 10, acquire_thread_func, &lock);
  msg ("Main thread should have priority %d.  Actual priority: %d.",
       PRI_DEFAULT + 10, thread_get_priority ());

  msg ("Lowering base priority...");
  thread_set_priority (PRI_DEFAULT - 10);
  msg ("Main thread should have priority %d.  Actual priority: %d.",
       PRI_DEFAULT + 10, thread_get_priority ());
  lock_release (&lock);
  msg ("acquire must already have finished.");
  msg ("Main thread should have priority %d.  Actual priority: %d.",
       PRI_DEFAULT - 10, thread_get_priority ());
}

static void
acquire_thread_func (void *lock_)
{
  struct lock *lock = lock_;

  lock_acquire (lock);
  msg ("acquire: got the lock");
  lock_release (lock);
  msg ("acquire: done");
}
```

这里当前线程有一个锁， 然后创建acquire抢占式获取了这个锁阻塞， 然后此时original_thread优先级为PRI_DEFAULT+10， 然后这里调用thread_set_priority， 此时当前线程的优先级应该没有改变， 但是它以后如果恢复优先级时候其实是有改变的， 就是说， 我们如果用original_priority来记录他的话， 如果这个线程处于被捐赠状态的话则直接修改original_priority来完成逻辑， 此时函数过后优先级还是PRI_DEFAULT+10， 然后释放掉锁，acquire抢占输出和释放， 然后original_thread的优先级应该变成了PRI_DEFAULT-10。

这里测试的逻辑是当修改一个被捐赠的线程优先级的时候的行为正确性。

还剩下最后3个测试分析， 先看priority-sema：

```
void
test_priority_sema (void)
{
  int i;

  /* This test does not work with the MLFQS. */
  ASSERT (!thread_mlfqs);

  sema_init (&sema, 0);
  thread_set_priority (PRI_MIN);
  for (i = 0; i < 10; i++)
    {
      int priority = PRI_DEFAULT - (i + 3) % 10 - 1;
      char name[16];
      snprintf (name, sizeof name, "priority %d", priority);
      thread_create (name, priority, priority_sema_thread, NULL);
    }

  for (i = 0; i < 10; i++)
    {
      sema_up (&sema);
      msg ("Back in main thread.");
    }
}

static void
priority_sema_thread (void *aux UNUSED)
{
```

```
29   sema_down (&sema);
30   msg ("Thread %s woke up.", thread_name ());
31 }
```

这里创建10个线程阻塞于P操作， 然后用一个循环V操作， 然后看结果：

```
# -*- perl -*-
use strict;
use warnings;
use tests::tests;
check_expected ([<<'EOF']);
(priority-sema) begin
(priority-sema) Thread priority 30 woke up.
(priority-sema) Back in main thread.
(priority-sema) Thread priority 29 woke up.
(priority-sema) Back in main thread.
(priority-sema) Thread priority 28 woke up.
(priority-sema) Back in main thread.
(priority-sema) Thread priority 27 woke up.
(priority-sema) Back in main thread.
(priority-sema) Thread priority 26 woke up.
(priority-sema) Back in main thread.
(priority-sema) Thread priority 25 woke up.
(priority-sema) Back in main thread.
(priority-sema) Thread priority 24 woke up.
(priority-sema) Back in main thread.
(priority-sema) Thread priority 23 woke up.
(priority-sema) Back in main thread.
(priority-sema) Thread priority 22 woke up.
(priority-sema) Back in main thread.
(priority-sema) Thread priority 21 woke up.
(priority-sema) Back in main thread.
(priority-sema) end
EOF
pass;
```

好， 也就是说V唤醒的时候也是优先级高的先唤醒， 换句话说， 信号量的等待队列是优先级队列。

再看priority-condvar测试：

```
void
test_priority_condvar (void)
{
  int i;

  /* This test does not work with the MLFQS. */
  ASSERT (!thread_mlfqs);

  lock_init (&lock);
  cond_init (&condition);

  thread_set_priority (PRI_MIN);
  for (i = 0; i < 10; i++)
    {
      int priority = PRI_DEFAULT - (i + 7) % 10 - 1;
      char name[16];
      snprintf (name, sizeof name, "priority %d", priority);
      thread_create (name, priority, priority_condvar_thread, NULL);
    }

  for (i = 0; i < 10; i++)
    {
      lock_acquire (&lock);
      msg ("Signaling...");
      cond_signal (&condition, &lock);
      lock_release (&lock);
```

```
27     }
28 }
29
30 static void
31 priority_condvar_thread (void *aux UNUSED)
32 {
33   msg ("Thread %s starting.", thread_name ());
34   lock_acquire (&lock);
35   cond_wait (&condition, &lock);
36   msg ("Thread %s woke up.", thread_name ());
37   lock_release (&lock);
38 }
```

这里condition里面装的是一个waiters队列， 看一下con_wait和cond_signal函数：

```
1 /* Atomically releases LOCK and waits for COND to be signaled by
2    some other piece of code.  After COND is signaled, LOCK is
3    reacquired before returning.  LOCK must be held before calling
4    this function.
5
6    The monitor implemented by this function is "Mesa" style, not
7    "Hoare" style, that is, sending and receiving a signal are not
8    an atomic operation.  Thus, typically the caller must recheck
9    the condition after the wait completes and, if necessary, wait
10   again.
11
12   A given condition variable is associated with only a single
13   lock, but one lock may be associated with any number of
14   condition variables.  That is, there is a one-to-many mapping
15   from locks to condition variables.
16
17   This function may sleep, so it must not be called within an
18   interrupt handler.  This function may be called with
19   interrupts disabled, but interrupts will be turned back on if
20   we need to sleep. */
21 void
22 cond_wait (struct condition *cond, struct lock *lock)
23 {
24   struct semaphore_elem waiter;
25
26   ASSERT (cond != NULL);
27   ASSERT (lock != NULL);
28   ASSERT (!intr_context ());
29   ASSERT (lock_held_by_current_thread (lock));
30
31   sema_init (&waiter.semaphore, 0);
32   list_push_back (&cond->waiters, &waiter.elem);
33   lock_release (lock);
34   sema_down (&waiter.semaphore);
35   lock_acquire (lock);
36 }
37
38 /* If any threads are waiting on COND (protected by LOCK), then
39    this function signals one of them to wake up from its wait.
40    LOCK must be held before calling this function.
41
42    An interrupt handler cannot acquire a lock, so it does not
43    make sense to try to signal a condition variable within an
44    interrupt handler. */
45 void
46 cond_signal (struct condition *cond, struct lock *lock UNUSED)
47 {
48   ASSERT (cond != NULL);
49   ASSERT (lock != NULL);
50   ASSERT (!intr_context ());
51   ASSERT (lock_held_by_current_thread (lock));
52
53   if (!list_empty (&cond->waiters))
54     sema_up (&list_entry (list_pop_front (&cond->waiters),
```

```
55                          struct semaphore_elem, elem)->semaphore);
56 }
```

分析： cond_wait和cond_signal就是释放掉锁， 等待signal唤醒， 然后再重新获取锁。

这里的代码逻辑是： 创建10个线程， 每个线程调用的时候获取锁， 然后调用cond_wait把锁释放阻塞于cond_signal唤醒， 然后连续10次循环调用cond_signal。

来看输出：

```
# -*- perl -*-
use strict;
use warnings;
use tests::tests;
check_expected ([<<'EOF']);
(priority-condvar) begin
(priority-condvar) Thread priority 23 starting.
(priority-condvar) Thread priority 22 starting.
(priority-condvar) Thread priority 21 starting.
(priority-condvar) Thread priority 30 starting.
(priority-condvar) Thread priority 29 starting.
(priority-condvar) Thread priority 28 starting.
(priority-condvar) Thread priority 27 starting.
(priority-condvar) Thread priority 26 starting.
(priority-condvar) Thread priority 25 starting.
(priority-condvar) Thread priority 24 starting.
(priority-condvar) Signaling...
(priority-condvar) Thread priority 30 woke up.
(priority-condvar) Signaling...
(priority-condvar) Thread priority 29 woke up.
(priority-condvar) Signaling...
(priority-condvar) Thread priority 28 woke up.
(priority-condvar) Signaling...
(priority-condvar) Thread priority 27 woke up.
(priority-condvar) Signaling...
(priority-condvar) Thread priority 26 woke up.
(priority-condvar) Signaling...
(priority-condvar) Thread priority 25 woke up.
(priority-condvar) Signaling...
(priority-condvar) Thread priority 24 woke up.
(priority-condvar) Signaling...
(priority-condvar) Thread priority 23 woke up.
(priority-condvar) Signaling...
(priority-condvar) Thread priority 22 woke up.
(priority-condvar) Signaling...
(priority-condvar) Thread priority 21 woke up.
(priority-condvar) end
EOF
pass;
```

从结果来看， 这里要求的实质就是： condition的waiters队列是优先级队列

好， 来看最后一个测试priority-donate-chain：

```
void
test_priority_donate_chain (void)
{
  int i;
  struct lock locks[NESTING_DEPTH - 1];
  struct lock_pair lock_pairs[NESTING_DEPTH];

  /* This test does not work with the MLFQS. */
  ASSERT (!thread_mlfqs);

  thread_set_priority (PRI_MIN);

  for (i = 0; i < NESTING_DEPTH - 1; i++)
```

```
14     lock_init (&locks[i]);
15
16   lock_acquire (&locks[0]);
17   msg ("%s got lock.", thread_name ());
18
19   for (i = 1; i < NESTING_DEPTH; i++)
20     {
21       char name[16];
22       int thread_priority;
23
24       snprintf (name, sizeof name, "thread %d", i);
25       thread_priority = PRI_MIN + i * 3;
26       lock_pairs[i].first = i < NESTING_DEPTH - 1 ? locks + i: NULL;
27       lock_pairs[i].second = locks + i - 1;
28
29       thread_create (name, thread_priority, donor_thread_func, lock_pairs + i);
30       msg ("%s should have priority %d.  Actual priority: %d.",
31           thread_name (), thread_priority, thread_get_priority ());
32
33       snprintf (name, sizeof name, "interloper %d", i);
34       thread_create (name, thread_priority - 1, interloper_thread_func, NULL);
35     }
36
37   lock_release (&locks[0]);
38   msg ("%s finishing with priority %d.", thread_name (),
39                                          thread_get_priority ());
40 }
41
42 static void
43 donor_thread_func (void *locks_)
44 {
45   struct lock_pair *locks = locks_;
46
47   if (locks->first)
48     lock_acquire (locks->first);
49
50   lock_acquire (locks->second);
51   msg ("%s got lock", thread_name ());
52
53   lock_release (locks->second);
54   msg ("%s should have priority %d. Actual priority: %d",
55        thread_name (), (NESTING_DEPTH - 1) * 3,
56        thread_get_priority ());
57
58   if (locks->first)
59     lock_release (locks->first);
60
61   msg ("%s finishing with priority %d.", thread_name (),
62                                          thread_get_priority ());
63 }
64
65 static void
66 interloper_thread_func (void *arg_ UNUSED)
67 {
68   msg ("%s finished.", thread_name ());
69 }
```

首先lock_pair是包含两个lock指针的结构体， 然后将当前线程优先级设为PRI_MIN， 然后这里有个locks数组， 容量为7, 然后lock_pairs数组用来装lock_pair， 容量也是7。

然后当前线程获取locks[0]这个锁， 接着跳到7次循环里， 每次循环thread_priority为PRI_MIN+i*3， 也就是3,6,9,12... 然后对应的lock_pairs[i]的first记录locks[i]的指针， second记录locks[i-1]指针，

然后创建线程， 优先级为thread_priority, 执行参数传的是&lock_pairs[i]， 注意这里由于优先级每次都底层， 所以每次循环都会抢占调用donor_thread_func， 然后分别获取lock_pairs[i]里装的锁， 然后每次循环先获取first, 即locks[i], 然后获取second, 由于second是前一个， 而前一个的拥有者一定是前一次循环创建的线程， 第一次拿得的是locks[0]， 最后一次循环first为NULL， second为locks[6]， 即最后一个线程不拥有锁， 但是阻塞于前一个创建的线程， 这里还会输出信息， 即创建的线程阻塞之后会输出当前线程的优先级msg, 当然这里必然是每一次都提升了的， 所以每次都是thread_priority。

然后每次循环最后还创建了1个线程， 优先级为thread_priority-1， 但是这里由于上一个线程创建和阻塞的过程中优先级捐献已经发生， 所以这里并不发生抢占， 只是创建出来了而已。

然后original_thread释放掉locks[0]， 释放掉这个之后thread1得到了唤醒， 输出信息， 释放掉这个锁， 然后输出当前优先级， 由于这个线程还是被后面最高优先级的线程说捐赠的， 所以每次往后优先级都是21, 然后释放掉first， 这里又触发下一个线程继续跑， 注意当后面的全部跑完的时候当前线程的优先级其实是不被捐赠的， 这里就变成了原来的优先级， 但是是所有线程都释放了之后才依次返回输出结束msg。

这个测试其实就是一个链式优先级捐赠， 本质测试的还是多层优先级捐赠逻辑的正确性。

需要注意的是一个逻辑： 释放掉一个锁之后， 如果当前线程不被捐赠即马上改为原来的优先级， 抢占式调度。

好， 我们把所有priority相关的测试都分析了个遍， 现在再来写代码实现这些测试逻辑就是很简单的事情了， 来搞～

先总结一下所有测试整合的逻辑：

1. 在一个线程获取一个锁的时候， 如果拥有这个锁的线程优先级比自己低就提高它的优先级，并且如果这个锁还被别的锁锁着， 将会递归地捐赠优先级， 然后在这个线程释放掉这个锁之后恢复未捐赠逻辑下的优先级。

2. 如果一个线程被多个线程捐赠， 维持当前优先级为捐赠优先级中的最大值（acquire和release之时）。

3. 在对一个线程进行优先级设置的时候， 如果这个线程处于被捐赠状态， 则对original_priority进行设置， 然后如果设置的优先级大于当前优先级， 则改变当前优先级， 否则在捐赠状态取消的时候恢复original_priority。

4. 在释放锁对一个锁优先级有改变的时候应考虑其余被捐赠优先级和当前优先级。

5. 将信号量的等待队列实现为优先级队列。

6. 将condition的waiters队列实现为优先级队列。

7. 释放锁的时候若优先级改变则可以发生抢占。

具体代码实现：

先修改thread数据结构， 加入以下成员：

```
    int base_priority;                  /* Base priority. */
    struct list locks;                  /* Locks that the thread is holding. */
    struct lock *lock_waiting;          /* The lock that the thread is waiting for. */
```

然后给lock加一下成员：

```
    struct list_elem elem;      /* List element for priority donation. */
    int max_priority;          /* Max priority among the threads acquiring the lock. */
```

好， 数据结构只需要增加这些就可以满足我们的需要了， 开写。

先修改lock_acquire函数：

```
void
lock_acquire (struct lock *lock)
{
  struct thread *current_thread = thread_current ();
  struct lock *l;
  enum intr_level old_level;

  ASSERT (lock != NULL);
  ASSERT (!intr_context ());
  ASSERT (!lock_held_by_current_thread (lock));

  if (lock->holder != NULL && !thread_mlfqs)
  {
    current_thread->lock_waiting = lock;
    l = lock;
    while (l && current_thread->priority > l->max_priority)
    {
      l->max_priority = current_thread->priority;
      thread_donate_priority (l->holder);
      l = l->holder->lock_waiting;
    }
  }

  sema_down (&lock->semaphore);

  old_level = intr_disable ();

```

```
28   current_thread = thread_current ();
29   if (!thread_mlfqs)
30   {
31     current_thread->lock_waiting = NULL;
32     lock->max_priority = current_thread->priority;
33     thread_hold_the_lock (lock);
34   }
35   lock->holder = current_thread;
36
37   intr_set_level (old_level);
38 }
```

在P操作之前递归地实现优先级捐赠， 然后在被唤醒之后（此时这个线程已经拥有了这个锁），成为这个锁的拥有者。

这里thread_donate_priority和thread_hold_the_lock封装成函数，注意一下这里优先级捐赠是通过直接修改锁的最高优先级， 然后调用update的时候把现成优先级更新实现的，update下面会写， 实现如下：

```
1 /* Let thread hold a lock */
2 void
3 thread_hold_the_lock(struct lock *lock)
4 {
5   enum intr_level old_level = intr_disable ();
6   list_insert_ordered (&thread_current ()->locks, &lock->elem, lock_cmp_priority, NULL);
7
8   if (lock->max_priority > thread_current ()->priority)
9   {
10    thread_current ()->priority = lock->max_priority;
11    thread_yield ();
12  }
13
14  intr_set_level (old_level);
15 }
```

```
1 /* Donate current priority to thread t. */
2 void
3 thread_donate_priority (struct thread *t)
4 {
5   enum intr_level old_level = intr_disable ();
6   thread_update_priority (t);
7
8   if (t->status == THREAD_READY)
9   {
10    list_remove (&t->elem);
11    list_insert_ordered (&ready_list, &t->elem, thread_cmp_priority, NULL);
12  }
13  intr_set_level (old_level);
14 }
```

锁队列排序函数lock_cmp_priority:

```
1 /* lock comparation function */
2 bool
3 lock_cmp_priority (const struct list_elem *a, const struct list_elem *b, void *aux UNUSED)
4 {
5   return list_entry (a, struct lock, elem)->max_priority > list_entry (b, struct lock, elem)->max_priority;
6 }
```

然后在lock_release函数加入以下语句：

```
1   if (!thread_mlfqs)
2     thread_remove_lock (lock);
```

thread_remove_lock实现如下：

```
/* Remove a lock. */
void
thread_remove_lock (struct lock *lock)
{
  enum intr_level old_level = intr_disable ();
  list_remove (&lock->elem);
  thread_update_priority (thread_current ());
  intr_set_level (old_level);
}
```

当释放掉一个锁的时候， 当前线程的优先级可能发生变化， 我们用thread_update_priority来处理这个逻辑：

```
/* Update priority. */
void
thread_update_priority (struct thread *t)
{
  enum intr_level old_level = intr_disable ();
  int max_priority = t->base_priority;
  int lock_priority;

  if (!list_empty (&t->locks))
  {
    list_sort (&t->locks, lock_cmp_priority, NULL);
    lock_priority = list_entry (list_front (&t->locks), struct lock, elem)->max_priority;
    if (lock_priority > max_priority)
      max_priority = lock_priority;
  }

  t->priority = max_priority;
  intr_set_level (old_level);
}
```

这里如果这个线程还有锁， 就先获取这个线程拥有锁的最大优先级（可能被更高级线程捐赠）， 然后如果这个优先级比base_priority大的话更新的应该是被捐赠的优先级。

然后在init_thread中加入初始化：

```
  t->base_priority = priority;
  list_init (&t->locks);
  t->lock_waiting = NULL;
```

修改一下thread_set_priority：

```
void
thread_set_priority (int new_priority)
{
  if (thread_mlfqs)
    return;

  enum intr_level old_level = intr_disable ();

  struct thread *current_thread = thread_current ();
  int old_priority = current_thread->priority;
  current_thread->base_priority = new_priority;

  if (list_empty (&current_thread->locks) || new_priority > old_priority)
  {
    current_thread->priority = new_priority;
    thread_yield ();
  }

  intr_set_level (old_level);
}
```

好， 至此整个捐赠的逻辑都写完了， 还差两个优先级队列的实现， 搞起～

然后把condition的队列改成优先级队列， 修改如下， 修改cond_signal函数：

```
void
cond_signal (struct condition *cond, struct lock *lock UNUSED)
{
  ASSERT (cond != NULL);
  ASSERT (lock != NULL);
  ASSERT (!intr_context ());
  ASSERT (lock_held_by_current_thread (lock));

  if (!list_empty (&cond->waiters))
  {
    list_sort (&cond->waiters, cond_sema_cmp_priority, NULL);
    sema_up (&list_entry (list_pop_front (&cond->waiters), struct semaphore_elem, elem)->semaphore);
  }
}
```

比较函数：

```
/* cond sema comparation function */
bool
cond_sema_cmp_priority (const struct list_elem *a, const struct list_elem *b, void *aux UNUSED)
{
  struct semaphore_elem *sa = list_entry (a, struct semaphore_elem, elem);
  struct semaphore_elem *sb = list_entry (b, struct semaphore_elem, elem);
  return list_entry(list_front(&sa->semaphore.waiters), struct thread, elem)->priority > list_entry(list_front(&sb->semaphore.waiters), struct thread, elem)->priority;
}
```

然后把信号量的等待队列实现为优先级队列：

修改sema_up：

```
void
sema_up (struct semaphore *sema)
{
  enum intr_level old_level;

  ASSERT (sema != NULL);

  old_level = intr_disable ();
  if (!list_empty (&sema->waiters))
  {
    list_sort (&sema->waiters, thread_cmp_priority, NULL);
    thread_unblock (list_entry (list_pop_front (&sema->waiters), struct thread, elem));
  }

  sema->value++;
  thread_yield ();
  intr_set_level (old_level);
}
```

修改sema_down：

```
void
sema_down (struct semaphore *sema)
{
  enum intr_level old_level;

  ASSERT (sema != NULL);
  ASSERT (!intr_context ());

  old_level = intr_disable ();
  while (sema->value == 0)
    {
```

```
12       list_insert_ordered (&sema->waiters, &thread_current ()->elem, thread_cmp_priority, NULL);
13       thread_block ();
14     }
15   sema->value--;
16   intr_set_level (old_level);
17 }
```

好， 做完这些其实是一气呵成的， 因为之前多测试需求有足够多的分析了， 来看测试结果：

毫无悬念地过了， 至此mission2完成，撒个花， 看个B站继续搞mission3~

Mission3:

实现多级反馈调度（2.2.4）（Advanced Scheduler or 4.4BSD Scheduler）

作用： 减少系统平均响应时间。

在这个mission中我们需要把mlfqs*给过了， 完成之后实验一的所有测试就pass了~

实验具体描述（必读）： http://www.ccs.neu.edu/home/amislove/teaching/cs5600/fall10/pintos/pintos_7.html

> Every thread has a *nice* value between -20 and 20 directly under its control. Each thread also has a priority, between 0 (PRI_MIN) through 63 (PRI_MAX), which is recalculated using the following formula every fourth tick:
>
> *priority* = PRI_MAX - (*recent_cpu* / 4) - (*nice* * 2).
>
> *recent_cpu* measures the amount of CPU time a thread has received "recently." On each timer tick, the running thread's *recent_cpu* is incremented by 1. Once per second, every thread's *recent_cpu* is updated this way:
>
> *recent_cpu* = (2**load_avg*)/(2**load_avg* + 1) * *recent_cpu* + *nice*.
>
> *load_avg* estimates the average number of threads ready to run over the past minute. It is initialized to 0 at boot and recalculated once per second as follows:
>
> *load_avg* = (59/60)**load_avg* + (1/60)**ready_threads*.
>
> where *ready_threads* is the number of threads that are either running or ready to run at time of update (not including the idle thread).

简单来说这里是维持了64个队列， 每个队列对应一个优先级， 从PRI_MIN到PRI_MAX。

然后通过一些公式计算来计算出线程当前的优先级， 系统调度的时候会从高优先级队列开始选择线程执行， 这里线程的优先级随着操作系统的运转数据而动态改变。

然后这个计算又涉及到了浮点数运算的问题， pintos本身并没有实现这个， 需要我们自己来搞。

The following table summarizes how fixed-point arithmetic operations can be implemented in C. In the table, x and y are fixed-point numbers, nis an integer, fixed-point numbers are in signed p.q format where p + q = 31, and f is 1 << q:

Convert n to fixed point:	n * f
Convert x to integer (rounding toward zero):	x / f
Convert x to integer (rounding to nearest):	(x + f / 2) / f if x >= 0, (x - f / 2) / f if x <= 0.
Add x and y:	x + y
Subtract y from x:	x - y
Add x and n:	x + n * f
Subtract n from x:	x - n * f
Multiply x by y:	((int64_t) x) * y / f
Multiply x by n:	x * n
Divide x by y:	((int64_t) x) * f / y
Divide x by n:	x / n

实现思路： 在timer_interrupt中固定一段时间计算更新线程的优先级，这里是每TIMER_FREQ时间更新一次系统load_avg和所有线程的recent_cpu， 每4个timer_ticks更新一次线程优先级， 每个timer_tick running线程的recent_cpu加一， 虽然这里说的是维持64个优先级队列调度， 其本质还是优先级调度，我们保留之前写的优先级调度代码即可， 去掉优先级捐赠（之前donate相关代码已经对需要的地方加了thread_mlfqs的判断了）。

浮点运算逻辑实现在fixed_point.h中：

```
#ifndef __THREAD_FIXED_POINT_H
#define __THREAD_FIXED_POINT_H

/* Basic definitions of fixed point. */
typedef int fixed_t;
/* 16 LSB used for fractional part. */
#define FP_SHIFT_AMOUNT 16
/* Convert a value to fixed-point value. */
#define FP_CONST(A) ((fixed_t)(A << FP_SHIFT_AMOUNT))
/* Add two fixed-point value. */
#define FP_ADD(A,B) (A + B)
/* Add a fixed-point value A and an int value B. */
#define FP_ADD_MIX(A,B) (A + (B << FP_SHIFT_AMOUNT))
/* Substract two fixed-point value. */
#define FP_SUB(A,B) (A - B)
/* Substract an int value B from a fixed-point value A */
#define FP_SUB_MIX(A,B) (A - (B << FP_SHIFT_AMOUNT))
/* Multiply a fixed-point value A by an int value B. */
#define FP_MULT_MIX(A,B) (A * B)
/* Divide a fixed-point value A by an int value B. */
#define FP_DIV_MIX(A,B) (A / B)
/* Multiply two fixed-point value. */
#define FP_MULT(A,B) ((fixed_t)(((int64_t) A) * B >> FP_SHIFT_AMOUNT))
/* Divide two fixed-point value. */
#define FP_DIV(A,B) ((fixed_t)(((int64_t) A) << FP_SHIFT_AMOUNT) / B))
/* Get integer part of a fixed-point value. */
#define FP_INT_PART(A) (A >> FP_SHIFT_AMOUNT)
/* Get rounded integer of a fixed-point value. */
#define FP_ROUND(A) (A >= 0 ? ((A + (1 << (FP_SHIFT_AMOUNT - 1))) >> FP_SHIFT_AMOUNT) \
        : ((A - (1 << (FP_SHIFT_AMOUNT - 1))) >> FP_SHIFT_AMOUNT))

#endif /* thread/fixed_point.h */
```

注意一下这里的实现， 这里用16位数（FP_SHIFT_AMOUNT）作为浮点数的小数部分， 理解了这点就很容易看了， 注意一点是无论什么运算一定要维持整数部分从第17位开始就行。

先实现timer_interrupt的逻辑，加入以下代码：

```
  if (thread_mlfqs)
  {
    thread_mlfqs_increase_recent_cpu_by_one ();
    if (ticks % TIMER_FREQ == 0)
      thread_mlfqs_update_load_avg_and_recent_cpu ();
    else if (ticks % 4 == 0)
      thread_mlfqs_update_priority (thread_current ());
  }
```

然后填一下这里挖的坑：

```
/* Increase recent_cpu by 1. */
void
thread_mlfqs_increase_recent_cpu_by_one (void)
{
  ASSERT (thread_mlfqs);
  ASSERT (intr_context ());

  struct thread *current_thread = thread_current ();
  if (current_thread == idle_thread)
    return;
  current_thread->recent_cpu = FP_ADD_MIX (current_thread->recent_cpu, 1);
}
```

注意这里调用的运算都是浮点运算。

```
/* Every per second to refresh load_avg and recent_cpu of all threads. */
void
thread_mlfqs_update_load_avg_and_recent_cpu (void)
{
  ASSERT (thread_mlfqs);
  ASSERT (intr_context ());

  size_t ready_threads = list_size (&ready_list);
  if (thread_current () != idle_thread)
    ready_threads++;
  load_avg = FP_ADD (FP_DIV_MIX (FP_MULT_MIX (load_avg, 59), 60), FP_DIV_MIX (FP_CONST (ready_threads), 60));

  struct thread *t;
  struct list_elem *e = list_begin (&all_list);
  for (; e != list_end (&all_list); e = list_next (e))
  {
    t = list_entry(e, struct thread, allelem);
    if (t != idle_thread)
    {
      t->recent_cpu = FP_ADD_MIX (FP_MULT (FP_DIV (FP_MULT_MIX (load_avg, 2), FP_ADD_MIX (FP_MULT_MIX (load_avg, 2), 1)), t-
>recent_cpu), t->nice);
      thread_mlfqs_update_priority (t);
    }
  }
}
```

这里的大部分逻辑都是Project给好的算数运算逻辑， 真正的行为逻辑并不复杂， 再填最后一个坑：

```
/* Update priority. */
void
thread_mlfqs_update_priority (struct thread *t)
{
  if (t == idle_thread)
    return;

  ASSERT (thread_mlfqs);
  ASSERT (t != idle_thread);
```

```
10
11   t->priority = FP_INT_PART (FP_SUB_MIX (FP_SUB (FP_CONST (PRI_MAX), FP_DIV_MIX (t->recent_cpu, 4)), 2 * t->nice));
12   t->priority = t->priority < PRI_MIN ? PRI_MIN : t->priority;
13   t->priority = t->priority > PRI_MAX ? PRI_MAX : t->priority;
14 }
```

好， 把这个3个坑给填了之后我们mission3的主体逻辑就已经完成了。

然后处理一些细节改动， thread结构体加入以下成员：

```
1     int nice;                           /* Niceness. */
2     fixed_t recent_cpu;                 /* Recent CPU. */
```

线程初始化的时候初始化这两个新的成员， 在init_thread中加入这些代码：

```
1   t->nice = 0;
2   t->recent_cpu = FP_CONST (0);
```

然后填一下这个系统给你挖好的坑：

```
 1 /* Sets the current thread's nice value to NICE. */
 2 void
 3 thread_set_nice (int nice)
 4 {
 5   thread_current ()->nice = nice;
 6   thread_mlfqs_update_priority (thread_current ());
 7   thread_yield ();
 8 }
 9
10 /* Returns the current thread's nice value. */
11 int
12 thread_get_nice (void)
13 {
14   return thread_current ()->nice;
15 }
16
17 /* Returns 100 times the system load average. */
18 int
19 thread_get_load_avg (void)
20 {
21   return FP_ROUND (FP_MULT_MIX (load_avg, 100));
22 }
23
24 /* Returns 100 times the current thread's recent_cpu value. */
25 int
26 thread_get_recent_cpu (void)
27 {
28   return FP_ROUND (FP_MULT_MIX (thread_current ()->recent_cpu, 100));
29 }
30
```

然后在thread.c中加入全局变量：

```
1 fixed_t load_avg;
```

并在thread_start中初始化：

```
1   load_avg = FP_CONST (0);
```

好， 所有的坑都填完了， 跑一下测试：

```
pass tests/threads/mlfqs-block
pass tests/threads/alarm-single
pass tests/threads/alarm-multiple
pass tests/threads/alarm-simultaneous
pass tests/threads/alarm-priority
pass tests/threads/alarm-zero
pass tests/threads/alarm-negative
pass tests/threads/priority-change
pass tests/threads/priority-donate-one
pass tests/threads/priority-donate-multiple
pass tests/threads/priority-donate-multiple2
pass tests/threads/priority-donate-nest
pass tests/threads/priority-donate-sema
pass tests/threads/priority-donate-lower
pass tests/threads/priority-fifo
pass tests/threads/priority-preempt
pass tests/threads/priority-sema
pass tests/threads/priority-condvar
pass tests/threads/priority-donate-chain
pass tests/threads/mlfqs-load-1
pass tests/threads/mlfqs-load-60
pass tests/threads/mlfqs-load-avg
pass tests/threads/mlfqs-recent-1
pass tests/threads/mlfqs-fair-2
pass tests/threads/mlfqs-fair-20
pass tests/threads/mlfqs-nice-2
pass tests/threads/mlfqs-nice-10
pass tests/threads/mlfqs-block
All 27 tests passed.
```

看到这 “All 27 tests passed.” 还是挺激动的哈哈~

ok， 至此这个project1就全部完成了， 难度还是有的， 特别在优先级调度这个mission上。

历时差不多一周， 从开始做这个东西。

完结撒花。

谢谢。

标签: 操作系统

laiy
关注 - 2
粉丝 - 13
+加关注

7 0

« 上一篇：算法模版：非递归快速幂算法详解
» 下一篇：Sicily1020-大数求余算法及优化

posted @ 2015-04-28 14:40 laiy 阅读(24235) 评论(12) 编辑 收藏

评论列表

#1楼 2015-05-03 23:40 **ccccly**

谢谢~帮大忙了，研究了一晚上的pintos^_^

支持(2) 反对(0)

#2楼[楼主] 2015-05-04 15:16 **laiy**

@ ccccly
With pleasure.

支持(0) 反对(0)

#3楼 2015-05-24 14:49 **Jacck**

感谢分享！

支持(1) 反对(0)

#4楼 2015-06-02 20:40 **我就是那只嵩鼠**

我们写这个写了一学期，但是还是有很多不明白，楼主是如何搞懂这中间千丝万缕的联系？

支持(0) 反对(0)

#5楼[楼主] 2015-06-03 09:21 **laiy**

@ 我就是那只嵩鼠
把源码弄清楚，多下功夫。

支持(0) 反对(0)

#6楼 2015-12-13 17:10 **minghu6**

有2个问题想不明白：
1.
在mission1里，
为什么 sema cond的优先级队列要通过
list_sort排序，
难道list_insert_ordered本身不能保证优先级队列的有序性？
2.
就算要调用list_sort,为什么不直接使用thread_cmp_priority
而要使用cond_sema_cmp_priority
然后在那个函数里，很绕的从

```
list_elem =>semaphore_elem =>semphore =>waiters
```

然后再去waiters队头元素，再转换回struct thread指针
为啥要这么做，而不是直接将struct list_elem* 转换为
struct thread*？

支持(0) 反对(0)

#7楼[楼主] 2016-01-25 12:48 **laiy**

@ minghu6
1. 首先你说的是mission2，因为我不能确定加入队列的时候都是调用的list_insert_ordered，可能部分地方保留了非顺序插入，所以最简单粗暴的方式就是在需要的时候排一次序。
2. 代码可读性你认为直接类型转换强还是有指针逻辑的强？

支持(0) 反对(0)

#8楼 2016-01-25 13:01 **minghu6**

@ laiy
1.那就直接排序好了，似乎也没必要在某些地方顺序插入。

2.原来是从逻辑性讲，但复杂的嵌套可读性真的好吗？

支持(0) 反对(0)

#9楼[楼主] 2016-01-25 15:54 **laiy**

@ minghu6
1. 确实没必要
2. 我认为直接写类型转换别人根本看不懂从哪个逻辑转换过来的，就算嵌套关系复杂但还是能看懂的。就好比你给一个变量命一个很长很长的名字，也比a,b,c的可读性好。

支持(0) 反对(0)

#10楼 2016-01-25 16:29 **minghu6**

让我想起了Ada的故事。。。

支持(0) 反对(0)

#11楼 2018-03-08 07:20 **FlanKira**

大神能不能教下我。。么 有偿求教！

支持(0) 反对(0)

#12楼 2018-03-09 00:05 **FlanKira**

"对PGMASK取反的结果就是一个页面大小全部为0的这么个数"
PGMASK 应该是01111111111吧

支持(0) 反对(0)

刷新评论 刷新页面 返回顶部

注册用户登录后才能发表评论，请 登录 或 注册，访问网站首页。