

华东师范大学软件工程学院实验报告

姓 名: 李鹏达

学 号: 10225101460

实验编号: Project 1

实验名称: Threads in Pintos

1 实验目的

- 1) 重新实现 `timer_sleep()` 函数，以避免进程在就绪和运行状态间的不停切换
- 2) 实现优先级调度
- 3) 实现多级反馈调度

2 实验内容与实验步骤

在本实验中，我们将对原有的进程调度功能进行三部分的改进：

- 1) 重新实现 `timer_sleep()` 函数，以避免进程在就绪和运行状态间的不停切换。
- 2) 实现优先级调度
- 3) 实现多级反馈调度

最终达到目标——使该项目的 27 个检测点全部通过。

主要探讨了时钟、进程状态、优先级调度（抢占）、信号量这四大问题。

2.1 重新实现 `timer_sleep()` 函数

首先我们先查看原本的 `timer_sleep()` 函数的实现。

原本的 `timer_sleep()` 函数

```
1 /* Sleeps for approximately TICKS timer ticks. Interrupts must
2 be turned on. */
3 void
4 timer_sleep (int64_t ticks)
5 {
6     int64_t start = timer_ticks ();
7
8     ASSERT (intr_get_level () == INTR_ON);
9     while (timer_elapsed (start) < ticks)
10         thread_yield ();
11 }
```

可以发现，它的实现是通过查询当前等待时间是否小于应等待时间，如果小于应等待时间，则不断的调用 `thread_yield()` 函数来实现的。

接下来，我们查看 `thread_yield()` 函数的实现。

原本的 `thread_yield()` 函数

```
1  /* Yields the CPU. The current thread is not put to sleep and
2  may be scheduled again immediately at the scheduler's whim. */
3  void
4  thread_yield (void)
5  {
6      struct thread *cur = thread_current ();
7      enum intr_level old_level;
8
9      ASSERT (!intr_context ());
10
11     old_level = intr_disable ();
12     if (cur != idle_thread)
13         list_push_back (&ready_list, &cur->elem);
14     cur->status = THREAD_READY;
15     schedule ();
16     intr_set_level (old_level);
17 }
```

可以发现，`thread_yield()` 函数的实现，是不断检查当前线程是否为空闲线程，如果不是，则将当前线程放置到就绪队列的末尾，然后将当前线程的状态设置为 `THREAD_READY`，最后调用 `schedule()` 函数来进行调度。

这样的实现方式，会导致线程在就绪和运行状态间的不停切换，从而导致 CPU 资源的浪费。所以我们需要重新实现 `timer_sleep()` 函数，使得线程在休眠的时候不会占用 CPU 资源。

可以考虑在线程结构体中增加一个成员变量 `int64_t sleep_ticks`，用来记录线程应该休眠的时间。

修改后的线程结构体

```
1  struct thread
2  {
3      /* Owned by thread.c. */
4      tid_t tid; /* Thread identifier. */
5      enum thread_status status; /* Thread state. */
6      char name[16]; /* Name (for debugging purposes). */
7      uint8_t *stack; /* Saved stack pointer. */
8      int priority; /* Priority. */
9      struct list_elem allelem; /* List element for all threads list.
10         */
11     /* Shared between thread.c and synch.c. */
```

```

12     struct list_elem elem;                /* List element. */
13
14     /* 应该休眠的时间 */
15     int64_t sleep_ticks;                  /* Sleep ticks. */
16
17 #ifdef USERPROG
18     /* Owned by userprog/process.c. */
19     uint32_t *pagedir;                    /* Page directory. */
20 #endif
21
22     /* Owned by thread.c. */
23     unsigned magic;                       /* Detects stack overflow. */
24 };

```

接下来，我们需要初始化这个成员变量，我们在 `thread_create()` 函数中，将 `sleep_ticks` 初始化为 0。

修改后的 `thread_create()` 函数

```

1  tid_t
2  thread_create (const char *name, int priority,
3                thread_func *function, void *aux)
4  {
5      struct thread *t;
6      struct kernel_thread_frame *kf;
7      struct switch_entry_frame *ef;
8      struct switch_threads_frame *sf;
9      tid_t tid;
10
11     ASSERT (function != NULL);
12
13     /* Allocate thread. */
14     t = palloc_get_page (PAL_ZERO);
15     if (t == NULL)
16         return TID_ERROR;
17
18     /* Initialize thread. */
19     init_thread (t, name, priority);
20     tid = t->tid = allocate_tid ();
21     t->sleep_ticks = 0; // 初始化 sleep_ticks 为 0
22
23     /* Stack frame for kernel_thread(). */
24     kf = alloc_frame (t, sizeof *kf);
25     kf->eip = NULL;
26     kf->function = function;

```

```

27  kf->aux = aux;
28
29  /* Stack frame for switch_entry(). */
30  ef = alloc_frame (t, sizeof *ef);
31  ef->eip = (void *) (void) kernel_thread;
32
33  /* Stack frame for switch_threads(). */
34  sf = alloc_frame (t, sizeof *sf);
35  sf->eip = switch_entry;
36  sf->ebp = 0;
37
38  /* Add to run queue. */
39  thread_unblock (t);
40
41  return tid;
42 }

```

接下来在 `timer_sleep()` 函数中，将当前线程的 `sleep_ticks` 设置为应该等待的时间，接着，我们调用 `thread_block()` 函数将当前线程阻塞。注意，我们需要先关闭中断，防止在设置 `sleep_ticks` 和阻塞线程的过程中，被其他线程抢占，从而导致 `sleep_ticks` 被修改，或者线程被重新加入就绪队列。需要注意的是，如果休眠时间小于等于 0，则直接返回。

修改后的 `timer_sleep()` 函数

```

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) // 如果休眠时间小于等于 0
7     {
8         return; // 直接返回
9     }
10
11     ASSERT (intr_get_level () == INTR_ON);
12
13     enum intr_level old_level = intr_disable (); // 关闭中断
14     thread_current ()->sleep_ticks = ticks; // 设置 sleep_ticks
15     thread_block (); // 阻塞当前线程
16     INTR_set_level (old_level); // 恢复中断
17 }

```

接下来，我们需要修改 `timmer_interrupt()` 函数，来实现对 `sleep_ticks` 的更新。首先，我们需要遍历所有线程，将 `sleep_ticks` 减一，如果 `sleep_ticks` 为 0，则将线程唤醒。为了遍历线程，我们需要使用 `thread_foreach()` 函数，它可以对每个线程调用一个函数，我们需要编写这个函数来实现对 `sleep_ticks`

的更新。

修改后的 timer_interrupt() 函数

```

1  /* Timer interrupt handler. */
2  static void
3  timer_interrupt (struct intr_frame *args UNUSED)
4  {
5      ticks++;
6      thread_tick ();
7      thread_foreach (thread_tick_sleep, NULL); // 每次中断都调用
8  }
9
10 /* 休眠时间到了就唤醒 */
11 void
12 thread_tick_sleep (struct thread *t, void *aux UNUSED)
13 {
14     if (t->status == THREAD_BLOCKED && t->sleep_ticks > 0) // 如果休眠时间大于 0
15     {
16         t->sleep_ticks--; // 休眠时间减一
17         if (t->sleep_ticks == 0) // 如果休眠时间到了
18         {
19             thread_unblock (t); // 唤醒
20         }
21     }
22 }
```

这样，我们就完成了对 timer_sleep() 函数的重新实现。

2.2 实现优先级调度

2.2.1 测试 alarm-priority, priority-change, priority-fifo 和 priority-preempt

在原本的实现中，线程的调度是按照 FIFO 的顺序进行的，这样会导致优先级高的线程无法优先执行。所以我们需要实现优先级调度，使得优先级高的线程能够优先执行。

由于在 pintos 中有一个已经定义好的按顺序插入的函数 list_insert_ordered()，所以我们可以使用这个函数来实现优先级调度。list_insert_ordered() 函数的定义如下：

list_insert_ordered() 函数的定义

```

1  /* Inserts ELEM in the list in sorted order. */
2  void
3  list_insert_ordered (struct list *list, struct list_elem *elem,
4                      list_less_func *less_func, void *aux)
5  {
6      struct list_elem *e;
```

```

7
8  ASSERT (list != NULL);
9  ASSERT (elem != NULL);
10 ASSERT (less_func != NULL);
11
12 for (e = list_begin (list); e != list_end (list); e = list_next (e))
13     if (less_func (elem, e, aux))
14         break;
15 list_insert (e, elem);
16 }

```

首先，我们需要实现一个比较函数，用来比较两个线程的优先级。

比较函数

```

1  /* 比较函数 */
2  bool
3  thread_priority_cmp (const struct list_elem *a, const struct list_elem *b,
4                      void *aux UNUSED)
5  {
6      struct thread *ta = list_entry (a, struct thread, elem);
7      struct thread *tb = list_entry (b, struct thread, elem);
8      return ta->priority > tb->priority;
9  }

```

然后，我们需要修改 `thread_unblock()`、`thread_yield()` 和 `thread_init()` 这三个函数，使得线程在被唤醒、主动让出 CPU 和初始化的时候，按照优先级的顺序插入就绪队列。

对相关函数的修改

```

1  void
2  thread_unblock (struct thread *t)
3  {
4      enum intr_level old_level;
5
6      ASSERT (is_thread (t));
7
8      old_level = intr_disable ();
9      ASSERT (t->status == THREAD_BLOCKED);
10     list_insert_ordered (&ready_list, &t->elem, thread_priority_cmp, NULL); //
        按优先级排序
11     t->status = THREAD_READY;
12     intr_set_level (old_level);
13 }
14
15 void

```

```

16 thread_yield (void)
17 {
18     struct thread *cur = thread_current ();
19     enum intr_level old_level;
20
21     ASSERT (!intr_context ());
22
23     old_level = intr_disable ();
24     if (cur != idle_thread)
25         list_insert_ordered (&ready_list, &cur->elem, thread_priority_cmp, NULL);
26         // 按优先级排序
27     cur->status = THREAD_READY;
28     schedule ();
29     intr_set_level (old_level);
30 }
31
32 static void
33 init_thread (struct thread *t, const char *name, int priority)
34 {
35     enum intr_level old_level;
36
37     ASSERT (t != NULL);
38     ASSERT (PRI_MIN <= priority && priority <= PRI_MAX);
39     ASSERT (name != NULL);
40
41     memset (t, 0, sizeof *t);
42     t->status = THREAD_BLOCKED;
43     strcpy (t->name, name, sizeof t->name);
44     t->stack = (uint8_t *) t + PGSIZE;
45     t->priority = priority;
46     t->magic = THREAD_MAGIC;
47
48     old_level = intr_disable ();
49     list_insert_ordered (&all_list, &t->elem, thread_priority_cmp, NULL); //
50     // 按优先级排序
51     intr_set_level (old_level);
52 }

```

接下来，我们需要修改 `thread_create()` 函数，使得线程在创建的时候，按照优先级的顺序插入就绪队列。

修改后的 `thread_create()` 函数

```

1 tid_t
2 thread_create (const char *name, int priority,

```

```
3         thread_func *function, void *aux)
4 {
5     struct thread *t;
6     struct kernel_thread_frame *kf;
7     struct switch_entry_frame *ef;
8     struct switch_threads_frame *sf;
9     tid_t tid;
10
11     ASSERT (function != NULL);
12
13     /* Allocate thread. */
14     t = palloc_get_page (PAL_ZERO);
15     if (t == NULL)
16         return TID_ERROR;
17
18     /* Initialize thread. */
19     init_thread (t, name, priority);
20     tid = t->tid = allocate_tid ();
21     t->sleep_ticks = 0;
22
23     /* Stack frame for kernel_thread(). */
24     kf = alloc_frame (t, sizeof *kf);
25     kf->eip = NULL;
26     kf->function = function;
27     kf->aux = aux;
28
29     /* Stack frame for switch_entry(). */
30     ef = alloc_frame (t, sizeof *ef);
31     ef->eip = (void *) (void) kernel_thread;
32
33     /* Stack frame for switch_threads(). */
34     sf = alloc_frame (t, sizeof *sf);
35     sf->eip = switch_entry;
36     sf->ebp = 0;
37
38     /* Add to run queue. */
39     thread_unblock (t);
40
41     /* 优先级调度 */
42     if (thread_current ()->priority < priority)
43     {
44         thread_yield ();
45     }
46
```



```

47     return tid;
48 }

```

然后，我们需要修改 `thread_set_priority()` 函数，使得线程在修改优先级的时候，重新调度。

修改后的 `thread_set_priority()` 函数

```

1 void
2 thread_set_priority (int new_priority)
3 {
4     enum intr_level old_level = intr_disable ();
5     int old_priority = thread_current ()->priority;
6     thread_current ()->priority = new_priority;
7     thread_yield (); // 重新调度
8     intr_set_level (old_level);
9 }

```

这样，我们就可以通过测试 `alarm-priority`, `priority-change`, `priority-fifo` 和 `priority-preempt` 了。

接下来，我们需要实现优先级捐赠。

我们可以先阅读测试文件，分析需要进行哪些修改。

2.2.2 测试 `priority-donate-one`

首先，我们分析 `priority-donate-one` 测试，它的代码如下：

`priority-donate-one` 测试

```

1 static thread_func acquire1_thread_func;
2 static thread_func acquire2_thread_func;
3
4 void
5 test_priority_donate_one (void)
6 {
7     struct lock lock;
8
9     /* This test does not work with the MLFQS. */
10    ASSERT (!thread_mlfqs);
11
12    /* Make sure our priority is the default. */
13    ASSERT (thread_get_priority () == PRI_DEFAULT);
14
15    lock_init (&lock);
16    lock_acquire (&lock);
17    thread_create ("acquire1", PRI_DEFAULT + 1, acquire1_thread_func, &lock);
18    msg ("This thread should have priority %d. Actual priority: %d.",

```

```
19     PRI_DEFAULT + 1, thread_get_priority ());
20     thread_create ("acquire2", PRI_DEFAULT + 2, acquire2_thread_func, &lock);
21     msg ("This thread should have priority %d. Actual priority: %d.",
22         PRI_DEFAULT + 2, thread_get_priority ());
23     lock_release (&lock);
24     msg ("acquire2, acquire1 must already have finished, in that order.");
25     msg ("This should be the last line before finishing this test.");
26 }
27
28 static void
29 acquire1_thread_func (void *lock_)
30 {
31     struct lock *lock = lock_;
32
33     lock_acquire (lock);
34     msg ("acquire1: got the lock");
35     lock_release (lock);
36     msg ("acquire1: done");
37 }
38
39 static void
40 acquire2_thread_func (void *lock_)
41 {
42     struct lock *lock = lock_;
43
44     lock_acquire (lock);
45     msg ("acquire2: got the lock");
46     lock_release (lock);
47     msg ("acquire2: done");
48 }
```

以下是测试的主要步骤和分析：

1. 初始化锁：

```
1 struct lock lock;
2 lock_init(&lock);
```

在测试开始时，创建一个锁以供线程之间同步和竞争。

2. 主线程获取锁：

```
1 lock_acquire(&lock);
```

主线程获取锁，此时它的优先级是默认优先级 PRI_DEFAULT。

3. 创建两个高优先级的线程：

```
1 thread_create("acquire1", PRI_DEFAULT + 1, acquire1_thread_func, &lock);
2 thread_create("acquire2", PRI_DEFAULT + 2, acquire2_thread_func, &lock);
```

创建两个新线程，分别命名为 **acquire1** 和 **acquire2**，它们的优先级分别比主线程高 1 和 2。这两个线程被设计为在获取锁时被阻塞。

4. 释放锁：

```
1 lock_release(&lock);
```

主线程释放锁。由于这时两个新线程的优先级高于主线程，它们会优先尝试获取锁。

5. **acquire2** 线程先获取锁：

```
1 acquire2_thread_func(void *lock_)
2 {
3     // ...
4     lock_acquire(lock);
5     msg("acquire2: got the lock");
6     lock_release(lock);
7     // ...
8 }
```

由于 **acquire2** 的优先级更高，它首先获取了锁。

6. **acquire1** 线程后获取锁：

```
1 acquire1_thread_func(void *lock_)
2 {
3     // ...
4     lock_acquire(lock);
5     msg("acquire1: got the lock");
6     lock_release(lock);
7     // ...
8 }
```

由于 **acquire1** 的优先级次高，它在 **acquire2** 线程之后获取了锁。

7. 测试完成：

```
1 msg("acquire2, acquire1 must already have finished, in that order.");
2 msg("This should be the last line before finishing this test.");
```

打印测试结果，确认 **acquire2** 线程先完成，然后是 **acquire1** 线程。

总体来说，这个测试验证了线程在释放锁时是否正确地将其优先级传递给等待该锁的其他线程，从而确保线程按照它们的优先级顺序获取锁。

在这个测试中，主线程先持有锁，随后两个优先级更高、等待这个锁的线程被创建，于是这两个线程便会向主线程捐赠优先级。当主线程释放锁时，两个线程中优先级更高的线程会先获取锁，然后是优先级次高的线程。

为了通过这个测试，首先，我们需要修改 `lock_acquire()` 函数，使得线程在获取锁的时候，如果锁已经被占用，且当前线程的优先级大于锁的持有者的优先级，则将当前线程的优先级赋值给锁的持有者。

修改后的 `lock_acquire()` 函数

```

1 void
2 lock_acquire (struct lock *lock)
3 {
4     ASSERT (lock != NULL);
5     ASSERT (!intr_context ());
6     ASSERT (!lock_held_by_current_thread (lock));
7
8     if (lock->holder != NULL && lock->holder->priority < thread_current ()->
        priority)
9     {
10         lock->holder->priority = thread_current ()->priority;
11     }
12
13     sema_down (&lock->semaphore);
14     lock->holder = thread_current ();
15 }

```

然后，我们需要修改 `lock_release()` 函数，使得线程在释放锁的时候，将当前线程的优先级恢复为原来的优先级。

为此，我们需要在线程结构体中增加一个成员变量 `int original_priority`，用来记录线程的原始优先级。

修改后的线程结构体

```

1 struct thread
2 {
3     /* Owned by thread.c. */
4     tid_t tid; /* Thread identifier. */
5     enum thread_status status; /* Thread state. */
6     char name[16]; /* Name (for debugging purposes). */
7     uint8_t *stack; /* Saved stack pointer. */
8     int priority; /* Priority. */
9     struct list_elem allelem; /* List element for all threads list. */
10
11     /* Shared between thread.c and synch.c. */
12     struct list_elem elem; /* List element. */
13
14     /* 应该休眠的时间 */
15     int64_t sleep_ticks; /* Sleep ticks. */
16 }

```

```

17     /* 原始优先级 */
18     int original_priority;          /* Original priority. */
19
20 #ifdef USERPROG
21     /* Owned by userprog/process.c. */
22     uint32_t *pagedir;             /* Page directory. */
23 #endif
24
25     /* Owned by thread.c. */
26     unsigned magic;                /* Detects stack overflow. */
27 };

```

接下来，我们需要修改 `init_thread()` 函数，使得线程在创建的时候，将线程的原始优先级设置为线程的优先级。

修改后的 `init_thread()` 函数

```

1  static void
2  init_thread (struct thread *t, const char *name, int priority)
3  {
4      enum intr_level old_level;
5
6      ASSERT (t != NULL);
7      ASSERT (PRI_MIN <= priority && priority <= PRI_MAX);
8      ASSERT (name != NULL);
9
10     memset (t, 0, sizeof *t);
11     t->status = THREAD_BLOCKED;
12     strlcpy (t->name, name, sizeof t->name);
13     t->stack = (uint8_t *) t + PGSIZE;
14     t->priority = priority;
15     t->original_priority = priority;
16     t->magic = THREAD_MAGIC;
17
18     old_level = intr_disable ();
19     list_insert_ordered (&all_list, &t->allelem, thread_priority_cmp, NULL);
20     intr_set_level (old_level);
21 }

```

然后，我们修改 `lock_release()` 函数，使得线程在释放锁的时候，将当前线程的优先级恢复为原来的优先级。

修改后的 `lock_release()` 函数

```

1  void
2  lock_release (struct lock *lock)

```

```

3 {
4     ASSERT (lock != NULL);
5     ASSERT (lock_held_by_current_thread (lock));
6
7     lock->holder = NULL;
8     sema_up (&lock->semaphore);
9
10    thread_set_priority (thread_current ()->original_priority);
11 }

```

最后，我们修改 `sema_down` 函数，保证等待锁的线程按照优先级顺序排列。

修改后的 `sema_down()` 函数

```

1 void
2 sema_down (struct semaphore *sema)
3 {
4     enum intr_level old_level;
5
6     ASSERT (sema != NULL);
7     ASSERT (!intr_context ());
8
9     old_level = intr_disable ();
10    while (sema->value == 0)
11    {
12        list_insert_ordered (&sema->waiters, &thread_current ()->elem,
13                             thread_priority_cmp, NULL);
14        thread_block ();
15    }
16    sema->value--;
17    intr_set_level (old_level);
18 }

```

这样，我们就可以通过测试 `priority-donate-one` 了。

2.2.3 测试 `priority-donate-multiple` 和 `priority-donate-multiple2`

接下来，我们分析 `priority-donate-multiple2` 测试，它的代码如下：

`priority-donate-multiple` 测试

```

1 static thread_func a_thread_func;
2 static thread_func b_thread_func;
3 static thread_func c_thread_func;
4
5 void

```

```
6 test_priority_donate_multiple2 (void)
7 {
8     struct lock a, b;
9
10    /* This test does not work with the MLFQS. */
11    ASSERT (!thread_mlfqs);
12
13    /* Make sure our priority is the default. */
14    ASSERT (thread_get_priority () == PRI_DEFAULT);
15
16    lock_init (&a);
17    lock_init (&b);
18
19    lock_acquire (&a);
20    lock_acquire (&b);
21
22    thread_create ("a", PRI_DEFAULT + 3, a_thread_func, &a);
23    msg ("Main thread should have priority %d. Actual priority: %d.",
24         PRI_DEFAULT + 3, thread_get_priority ());
25
26    thread_create ("c", PRI_DEFAULT + 1, c_thread_func, NULL);
27
28    thread_create ("b", PRI_DEFAULT + 5, b_thread_func, &b);
29    msg ("Main thread should have priority %d. Actual priority: %d.",
30         PRI_DEFAULT + 5, thread_get_priority ());
31
32    lock_release (&a);
33    msg ("Main thread should have priority %d. Actual priority: %d.",
34         PRI_DEFAULT + 5, thread_get_priority ());
35
36    lock_release (&b);
37    msg ("Threads b, a, c should have just finished, in that order.");
38    msg ("Main thread should have priority %d. Actual priority: %d.",
39         PRI_DEFAULT, thread_get_priority ());
40 }
41
42 static void
43 a_thread_func (void *lock_)
44 {
45     struct lock *lock = lock_;
46
47     lock_acquire (lock);
48     msg ("Thread a acquired lock a.");
49     lock_release (lock);
```

```
50  msg ("Thread a finished.");
51  }
52
53  static void
54  b_thread_func (void *lock_)
55  {
56      struct lock *lock = lock_;
57
58      lock_acquire (lock);
59      msg ("Thread b acquired lock b.");
60      lock_release (lock);
61      msg ("Thread b finished.");
62  }
63
64  static void
65  c_thread_func (void *a_ UNUSED)
66  {
67      msg ("Thread c finished.");
68  }
```

以下是对测试步骤的分析：

1. 初始化锁：

```
1  struct lock a, b;
2  lock_init(&a);
3  lock_init(&b);
```

在测试开始时，创建两个锁，命名为 **a** 和 **b**，用于线程之间的同步。

2. 主线程获取锁 **a** 和 **b**：

```
1  lock_acquire(&a);
2  lock_acquire(&b);
```

主线程获取两个锁 **a** 和 **b**。

3. 创建三个高优先级的线程：

```
1  thread_create("a", PRI_DEFAULT + 3, a_thread_func, &a);
2  thread_create("c", PRI_DEFAULT + 1, c_thread_func, NULL);
3  thread_create("b", PRI_DEFAULT + 5, b_thread_func, &b);
```

创建三个新线程，分别命名为 **a**、**c** 和 **b**，它们的优先级分别比主线程高 3、1 和 5。

4. 释放锁 **a** 和 **b**：

```
1  lock_release(&a);
2  lock_release(&b);
```


主线程释放两个锁。

5. 测试结果输出:

```

1 msg ("Main thread should have priority %d. Actual priority: %d.",
2 PRI_DEFAULT + 3, thread_get_priority ());
3 // ...
4 msg ("Main thread should have priority %d. Actual priority: %d.",
5 PRI_DEFAULT + 5, thread_get_priority ());
6 // ...
7 msg ("Main thread should have priority %d. Actual priority: %d.",
8 PRI_DEFAULT + 5, thread_get_priority ());
9 // ...
10 msg("Threads b, a, c should have just finished, in that order.");
11 msg("Main thread should have priority %d. Actual priority: %d.",
12     PRI_DEFAULT, thread_get_priority ());

```

打印测试结果，确认主线程拥有正确的优先级，且最后线程 b、a、c 已经完成，按照优先级的顺序。

总体来说，这个测试验证了在有多个锁的情况下，线程在释放锁时，是否正确地将其优先级设为剩余锁的最高优先级，从而确保线程按照它们的优先级顺序获取锁。

为了通过这个测试，首先，我们必须记录线程持有的锁，为此，我们需要在线程结构体中增加一个成员变量 `struct list locks`，用来记录线程持有的锁。

修改后的线程结构体

```

1  struct thread
2  {
3      /* Owned by thread.c. */
4      tid_t tid; /* Thread identifier. */
5      enum thread_status status; /* Thread state. */
6      char name[16]; /* Name (for debugging purposes). */
7      uint8_t *stack; /* Saved stack pointer. */
8      int priority; /* Priority. */
9      struct list_elem allelem; /* List element for all threads list. */
10
11     /* Shared between thread.c and synch.c. */
12     struct list_elem elem; /* List element. */
13
14     /* 应该休眠的时间 */
15     int64_t sleep_ticks;
16
17     /* 原本的优先级 */
18     int original_priority; /* Original Priority*/
19
20     /* 线程的锁 */
21     struct list locks; /* Locks that the thread is holding */

```

```

22
23 #ifdef USERPROG
24     /* Owned by userprog/process.c. */
25     uint32_t *pagedir;          /* Page directory. */
26 #endif
27
28     /* Owned by thread.c. */
29     unsigned magic;             /* Detects stack overflow. */
30 };

```

同时，我们还必须记录想要获取这个锁的线程的最高优先级，为此，我们需要在锁结构体中增加成员变量 `int max_priority` 和 `struct list_elem elem`，用来记录想要获取这个锁的线程的最高优先级和用于插入列表的变量。

修改后的锁结构体

```

1 struct lock
2 {
3     struct thread *holder;      /* Thread holding lock (for debugging). */
4     struct semaphore semaphore; /* Binary semaphore controlling access. */
5     int max_priority;           /* Maximum priority of threads waiting for lock.
6     */
7     struct list_elem elem;      /* List element for priority donation. */
8 };

```

然后，我们修改 `lock_init()` 函数，使得线程在初始化锁的时候，将锁的最高优先级初始化为 `PRI_MIN`。

修改后的 `lock_init()` 函数

```

1 void
2 lock_init (struct lock *lock)
3 {
4     ASSERT (lock != NULL);
5
6     lock->holder = NULL;
7     lock->max_priority = PRI_MIN; // 初始化为 PRI_MIN
8     sema_init (&lock->semaphore, 1);
9 }

```

接下来，我们修改线程结构体，添加成员变量 `struct list_elem locks`，用来记录线程持有的锁。

修改后的线程结构体

```

1 struct thread
2 {
3     /* Owned by thread.c. */
4     tid_t tid;                /* Thread identifier. */

```

```

5  enum thread_status status;          /* Thread state. */
6  char name[16];                      /* Name (for debugging purposes). */
7  uint8_t *stack;                     /* Saved stack pointer. */
8  int priority;                       /* Priority. */
9  struct list_elem allelem;           /* List element for all threads list. */
10
11  /* Shared between thread.c and synch.c. */
12  struct list_elem elem;              /* List element. */
13
14  /* 应该休眠的时间 */
15  int64_t sleep_ticks;
16
17  /* 原本的优先级 */
18  int original_priority;               /* Original Priority*/
19
20  /* 持有的锁 */
21  struct list locks;                  /* List of locks that the thread is
    holding. */
22
23 #ifdef USERPROG
24     /* Owned by userprog/process.c. */
25     uint32_t *pagedir;               /* Page directory. */
26 #endif
27
28     /* Owned by thread.c. */
29     unsigned magic;                  /* Detects stack overflow. */
30 };

```

然后，我们修改 `init_thread()` 函数，使得线程在创建的时候，初始化线程持有的锁列表。

修改后的 `init_thread()` 函数

```

1  static void
2  init_thread (struct thread *t, const char *name, int priority)
3  {
4      enum intr_level old_level;
5
6      ASSERT (t != NULL);
7      ASSERT (PRI_MIN <= priority && priority <= PRI_MAX);
8      ASSERT (name != NULL);
9
10     memset (t, 0, sizeof *t);
11     t->status = THREAD_BLOCKED;
12     strlcpy (t->name, name, sizeof t->name);
13     t->stack = (uint8_t *) t + PGSIZE;

```

```

14  t->priority = priority;
15  t->original_priority = priority;
16  t->magic = THREAD_MAGIC;
17
18  list_init (&t->locks); // 初始化锁列表
19
20  old_level = intr_disable ();
21  list_insert_ordered (&all_list, &t->allelem, thread_priority_cmp, NULL);
22  intr_set_level (old_level);
23 }

```

然后，我们实现一个锁的比较函数，用来比较两个锁的最高优先级。

锁的比较函数

```

1  bool
2  lock_priority_cmp (const struct list_elem *a, const struct list_elem *b, void *
    aux UNUSED)
3  {
4      struct lock *la = list_entry (a, struct lock, elem);
5      struct lock *lb = list_entry (b, struct lock, elem);
6      return la->max_priority > lb->max_priority;
7  }

```

接下来，我们修改 `lock_acquire()` 函数，使得线程在获取锁的时候，如果锁已经被占用，且当前线程的优先级大于锁的持有者的优先级，则将当前线程的优先级赋值给锁的持有者，并将锁加入线程持有的锁列表中，并更新锁的最高优先级。

修改后的 `lock_acquire()` 函数

```

1  void
2  lock_acquire (struct lock *lock)
3  {
4      ASSERT (lock != NULL);
5      ASSERT (!intr_context ());
6      ASSERT (!lock_held_by_current_thread (lock));
7
8      if (lock->holder != NULL && lock->holder->priority < thread_current ()->
    priority)
9      {
10         lock->holder->priority = thread_current ()->priority;
11         if (lock->max_priority < thread_current ()->priority)
12             lock->max_priority = thread_current ()->priority;
13     }
14
15     sema_down (&lock->semaphore);

```

```

16
17  list_insert_ordered (&thread_current ()->locks, &lock->elem, lock_priority_cmp
    , NULL);
18  lock->holder = thread_current ();
19 }

```

最后，我们修改 `lock_release()` 函数，使得线程在释放锁的时候，将当前线程的优先级恢复（如果不持有锁，则恢复为初始优先级；若持有锁，则设置为所持有锁的最高优先级）。

修改后的 `lock_release()` 函数

```

1  void
2  lock_release (struct lock *lock)
3  {
4      int max_priority;
5
6      ASSERT (lock != NULL);
7      ASSERT (lock_held_by_current_thread (lock));
8
9      list_remove (&lock->elem);
10     max_priority = thread_current ()->original_priority;
11     if (!list_empty (&thread_current ()->locks))
12     {
13         list_sort (&thread_current ()->locks, lock_priority_cmp, NULL);
14         struct lock *l = list_entry (list_front (&thread_current ()->locks), struct
            lock, elem);
15         if (l->max_priority > max_priority)
16             max_priority = l->max_priority;
17     }
18
19     thread_current ()->priority = max_priority;
20
21     lock->holder = NULL;
22     sema_up (&lock->semaphore);
23 }

```

这样，我们就可以通过测试 `priority-donate-multiple`, `priority-donate-multiple2` 了。

2.2.4 测试 `priority-donate-nest`

接下来，我们分析 `priority-donate-nest` 测试，它的代码如下：

`priority-donate-nest` 测试

```

1  static thread_func medium_thread_func;
2  static thread_func high_thread_func;

```

```
3
4 void
5 test_priority_donate_nest (void)
6 {
7     struct lock a, b;
8     struct locks locks;
9
10    /* This test does not work with the MLFQS. */
11    ASSERT (!thread_mlfqs);
12
13    /* Make sure our priority is the default. */
14    ASSERT (thread_get_priority () == PRI_DEFAULT);
15
16    lock_init (&a);
17    lock_init (&b);
18
19    lock_acquire (&a);
20
21    locks.a = &a;
22    locks.b = &b;
23    thread_create ("medium", PRI_DEFAULT + 1, medium_thread_func, &locks);
24    thread_yield ();
25    msg ("Low thread should have priority %d. Actual priority: %d.",
26        PRI_DEFAULT + 1, thread_get_priority ());
27
28    thread_create ("high", PRI_DEFAULT + 2, high_thread_func, &b);
29    thread_yield ();
30    msg ("Low thread should have priority %d. Actual priority: %d.",
31        PRI_DEFAULT + 2, thread_get_priority ());
32
33    lock_release (&a);
34    thread_yield ();
35    msg ("Medium thread should just have finished.");
36    msg ("Low thread should have priority %d. Actual priority: %d.",
37        PRI_DEFAULT, thread_get_priority ());
38 }
39
40 static void
41 medium_thread_func (void *locks_)
42 {
43     struct locks *locks = locks_;
44
45     lock_acquire (locks->b);
46     lock_acquire (locks->a);
```

```
47
48 msg ("Medium thread should have priority %d. Actual priority: %d.",
49      PRI_DEFAULT + 2, thread_get_priority ());
50 msg ("Medium thread got the lock.");
51
52 lock_release (locks->a);
53 thread_yield ();
54
55 lock_release (locks->b);
56 thread_yield ();
57
58 msg ("High thread should have just finished.");
59 msg ("Middle thread finished.");
60 }
61
62 static void
63 high_thread_func (void *lock_)
64 {
65     struct lock *lock = lock_;
66
67     lock_acquire (lock);
68     msg ("High thread got the lock.");
69     lock_release (lock);
70     msg ("High thread finished.");
71 }
```

以下是对测试步骤的分析：

1. 初始化锁：

```
1 struct lock a, b;
2 struct locks
3 {
4     struct lock *a;
5     struct lock *b;
6 };
7 struct locks locks;
8
9 lock_init(&a);
10 lock_init(&b);
```

在测试开始时，创建两个锁，命名为 **a** 和 **b**。同时，定义一个结构 **locks** 包含指向这两个锁的指针。

2. 主线程获取锁 **a**：

```
1 lock_acquire(&a);
```

低优先级的主线程获取锁 a。

3. 创建两个新线程：

```
1 thread_create("medium", PRI_DEFAULT + 1, medium_thread_func, &locks);
2 thread_yield();
3 msg("Low thread should have priority %d. Actual priority: %d.",
4     PRI_DEFAULT + 1, thread_get_priority ());
5
6 thread_create("high", PRI_DEFAULT + 2, high_thread_func, &b);
7 thread_yield();
8 msg("Low thread should have priority %d. Actual priority: %d.",
9     PRI_DEFAULT + 2, thread_get_priority ());
```

创建一个中优先级的线程 **medium** 和一个高优先级的线程 **high**。此时，线程 **medium** 尝试获取锁 **a** 和锁 **b** 并阻塞，而线程 **high** 尝试获取锁 **b**。

4. 释放锁 a：

```
1 lock_release(&a);
2 thread_yield();
3 msg("Medium thread should just have finished.");
4 msg("Low thread should have priority %d. Actual priority: %d.",
5     PRI_DEFAULT, thread_get_priority ());
```

主线程释放锁 **a**，线程 **medium** 获取了锁并执行完毕。主线程此时重新获取控制，检查其优先级。

5. **medium** 线程执行：

```
1 static void medium_thread_func(void *locks_)
2 {
3     struct locks *locks = locks_;
4
5     lock_acquire(locks->b);
6     lock_acquire(locks->a);
7
8     msg("Medium thread should have priority %d. Actual priority: %d.",
9         PRI_DEFAULT + 2, thread_get_priority ());
10    msg("Medium thread got the lock.");
11
12    lock_release(locks->a);
13    thread_yield();
14
15    lock_release(locks->b);
16    thread_yield();
17
18    msg("High thread should have just finished.");
19    msg("Middle thread finished.");
```



```
20 }
```

线程 `medium` 获取锁 `b`，然后获取锁 `a`。由于线程 `high` 阻塞在锁 `b` 上，线程 `high` 的优先级捐赠给了线程 `medium`。线程 `medium` 执行完毕，释放锁 `a` 和 `b`。

6. `high` 线程执行：

```
1 static void high_thread_func(void *lock_)
2 {
3     struct lock *lock = lock_;
4
5     lock_acquire(lock);
6     msg("High thread got the lock.");
7     lock_release(lock);
8     msg("High thread finished.");
9 }
```

线程 `high` 获取锁 `b`，执行完毕后释放锁。

总体来说，这个测试验证了在不同优先级的线程之间进行嵌套优先级捐赠的情况。

可以作出如下关系图：

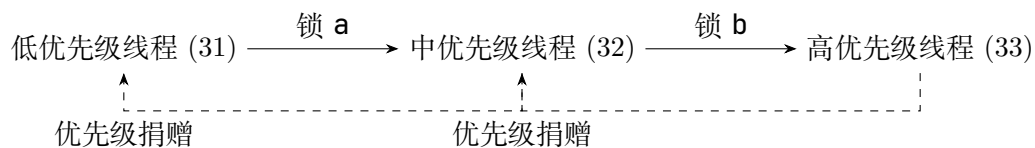


图 1: 线程关系图

通过分析，我们可以得知，优先级捐赠需要递归地进行，即当线程 `high` 捐赠优先级给线程 `medium` 时，线程 `medium` 也需要捐赠优先级给线程 `low`。

为了实现这个功能，我们必须知道线程等待的锁，为此，我们需要在线程结构体中增加一个成员变量 `struct lock *waiting`，用来记录线程等待的锁。

修改后的线程结构体

```
1 struct thread
2 {
3     /* Owned by thread.c. */
4     tid_t tid; /* Thread identifier. */
5     enum thread_status status; /* Thread state. */
6     char name[16]; /* Name (for debugging purposes). */
7     uint8_t *stack; /* Saved stack pointer. */
8     int priority; /* Priority. */
9     struct list_elem allelem; /* List element for all threads list. */
10 }
```

```

11  /* Shared between thread.c and synch.c. */
12  struct list_elem elem;          /* List element. */
13
14  /* 应该休眠的时间 */
15  int64_t sleep_ticks;
16
17  /* 原本的优先级 */
18  int original_priority;          /* Original Priority*/
19
20  /* 持有的锁 */
21  struct list locks;              /* List of locks that the thread is
    holding. */
22
23  /* 等待的锁 */
24  struct lock *waiting;           /* The lock that the thread is waiting
    for. */
25
26 #ifdef USERPROG
27  /* Owned by userprog/process.c. */
28  uint32_t *pagedir;             /* Page directory. */
29 #endif
30
31  /* Owned by thread.c. */
32  unsigned magic;                 /* Detects stack overflow. */
33 };

```

接下来，我们将优先级捐赠的过程提取出来，作为一个函数 `priority_donate()`，用来递归地进行优先级捐赠。

优先级捐赠函数

```

1  void
2  priority_donate (struct thread *t, struct lock *l)
3  {
4      if (l != NULL && t->priority > l->max_priority)
5      {
6          l->holder->priority = t->priority;
7          if (l->max_priority < t->priority)
8              l->max_priority = t->priority;
9          priority_donate (t, l->holder->waiting);
10     }
11 }

```

然后，我们修改 `lock_acquire()` 函数，使得线程在获取锁时，如果锁已被占用，则记录线程等待的锁并递归地进行优先级捐赠。

修改后的 lock_acquire() 函数

```

1 void
2 lock_acquire (struct lock *lock)
3 {
4     ASSERT (lock != NULL);
5     ASSERT (!intr_context ());
6     ASSERT (!lock_held_by_current_thread (lock));
7
8     if (lock->holder != NULL && !thread_mlfqs)
9     {
10         thread_current ()->waiting = lock;
11         priority_donate (thread_current (), lock);
12     }
13
14     sema_down (&lock->semaphore);
15
16     list_insert_ordered (&thread_current ()->locks, &lock->elem, lock_priority_cmp
17         , NULL);
18     thread_current ()->waiting = NULL;
19     lock->max_priority = thread_current ()->priority;
20     lock->holder = thread_current ();
21 }

```

2.2.5 测试 priority-donate-sema

接下来，我们分析 priority-donate-sema 测试，它的代码如下：

priority-donate-sema 测试

```

1 struct lock_and_sema
2 {
3     struct lock lock;
4     struct semaphore sema;
5 };
6
7 static thread_func l_thread_func;
8 static thread_func m_thread_func;
9 static thread_func h_thread_func;
10
11 void
12 test_priority_donate_sema (void)
13 {
14     struct lock_and_sema ls;
15

```

```
16  /* This test does not work with the MLFQS. */
17  ASSERT (!thread_mlfqs);
18
19  /* Make sure our priority is the default. */
20  ASSERT (thread_get_priority () == PRI_DEFAULT);
21
22  lock_init (&ls.lock);
23  sema_init (&ls.sema, 0);
24  thread_create ("low", PRI_DEFAULT + 1, l_thread_func, &ls);
25  thread_create ("med", PRI_DEFAULT + 3, m_thread_func, &ls);
26  thread_create ("high", PRI_DEFAULT + 5, h_thread_func, &ls);
27  sema_up (&ls.sema);
28  msg ("Main thread finished.");
29 }
30
31 static void
32 l_thread_func (void *ls_)
33 {
34     struct lock_and_sema *ls = ls_;
35
36     lock_acquire (&ls->lock);
37     msg ("Thread L acquired lock.");
38     sema_down (&ls->sema);
39     msg ("Thread L downed semaphore.");
40     lock_release (&ls->lock);
41     msg ("Thread L finished.");
42 }
43
44 static void
45 m_thread_func (void *ls_)
46 {
47     struct lock_and_sema *ls = ls_;
48
49     sema_down (&ls->sema);
50     msg ("Thread M finished.");
51 }
52
53 static void
54 h_thread_func (void *ls_)
55 {
56     struct lock_and_sema *ls = ls_;
57
58     lock_acquire (&ls->lock);
59     msg ("Thread H acquired lock.");
```

```
60
61     sema_up (&ls->sema);
62     lock_release (&ls->lock);
63     msg ("Thread H finished.");
64 }
```

以下是对测试步骤的分析：

- 初始化结构体：

```
1  struct lock_and_sema
2  {
3      struct lock lock;
4      struct semaphore sema;
5  };
```

在测试开始时，创建了一个包含锁和信号量的结构体。

- 创建线程并初始化结构体：

```
1  struct lock_and_sema ls;
2  lock_init(&ls.lock);
3  sema_init(&ls.sema, 0);
4  thread_create("low", PRI_DEFAULT + 1, l_thread_func, &ls);
5  thread_create("med", PRI_DEFAULT + 3, m_thread_func, &ls);
6  thread_create("high", PRI_DEFAULT + 5, h_thread_func, &ls);
7  sema_up(&ls.sema);
```

创建了三个线程，分别命名为 low、med 和 high，并将初始化的结构体传递给它们。主线程通过调用 sema_up 释放了信号量。

- 线程执行及互动：

```
1  static void l_thread_func(void *ls_)
2  {
3      struct lock_and_sema *ls = ls_;
4      lock_acquire(&ls->lock);
5      msg("Thread L acquired lock.");
6      sema_down(&ls->sema);
7      msg("Thread L downed semaphore.");
8      lock_release(&ls->lock);
9      msg("Thread L finished.");
10 }
11
12 static void m_thread_func(void *ls_)
13 {
```

```

14  struct lock_and_sema *ls = ls_;
15  sema_down(&ls->sema);
16  msg("Thread M finished.");
17  }
18
19  static void h_thread_func(void *ls_)
20  {
21  struct lock_and_sema *ls = ls_;
22  lock_acquire(&ls->lock);
23  msg("Thread H acquired lock.");
24  sema_up(&ls->sema);
25  lock_release(&ls->lock);
26  msg("Thread H finished.");
27  }

```

- Thread L: 获取锁，等待信号量，释放锁。
- Thread M: 等待信号量。
- Thread H: 获取锁，发送信号量，释放锁。

• 测试结果输出:

```
1 msg("Main thread finished.");
```

打印测试结果，确认主线程已经完成。

总体来说，这个测试涉及了使用锁和信号量的线程之间的互动，以验证在这种情况下优先级捐赠的正确性。为了通过这个测试，我们需要保证在进行 V 操作时，能够正确的根据线程的优先级来进行调度。

因此，我们需要修改 `sema_up()` 函数，使得线程在释放信号量的时候，能够根据线程的优先级来进行调度。

修改后的 `sema_up()` 函数

```

1  void
2  sema_up (struct semaphore *sema)
3  {
4  enum intr_level old_level;
5
6  ASSERT (sema != NULL);
7
8  old_level = intr_disable ();
9  if (!list_empty (&sema->waiters))
10 {
11     list_sort (&sema->waiters, thread_priority_cmp, NULL);
12     thread_unblock (list_entry (list_pop_front (&sema->waiters),
13                                     struct thread, elem));

```

```

14 }
15 sema->value++;
16 intr_set_level (old_level);
17 thread_yield (); // 释放信号量后进行调度
18 }

```

这样，我们就可以通过测试 `priority-donate-sema` 了。

2.2.6 测试 `priority-donate-lower`

接下来，我们分析 `priority-donate-lower` 测试，它的代码如下：

`priority-donate-lower` 测试

```

1 static thread_func acquire_thread_func;
2
3 void
4 test_priority_donate_lower (void)
5 {
6     struct lock lock;
7
8     /* This test does not work with the MLFQS. */
9     ASSERT (!thread_mlfqs);
10
11     /* Make sure our priority is the default. */
12     ASSERT (thread_get_priority () == PRI_DEFAULT);
13
14     lock_init (&lock);
15     lock_acquire (&lock);
16     thread_create ("acquire", PRI_DEFAULT + 10, acquire_thread_func, &lock);
17     msg ("Main thread should have priority %d. Actual priority: %d.",
18         PRI_DEFAULT + 10, thread_get_priority ());
19
20     msg ("Lowering base priority...");
21     thread_set_priority (PRI_DEFAULT - 10);
22     msg ("Main thread should have priority %d. Actual priority: %d.",
23         PRI_DEFAULT + 10, thread_get_priority ());
24     lock_release (&lock);
25     msg ("acquire must already have finished.");
26     msg ("Main thread should have priority %d. Actual priority: %d.",
27         PRI_DEFAULT - 10, thread_get_priority ());
28 }
29
30 static void
31 acquire_thread_func (void *lock_)

```

```
32 {  
33     struct lock *lock = lock_;  
34  
35     lock_acquire (lock);  
36     msg ("acquire: got the lock");  
37     lock_release (lock);  
38     msg ("acquire: done");  
39 }
```

以下是对测试步骤的分析：

- 初始化锁和创建线程：

```
1 struct lock lock;  
2 lock_init(&lock);  
3 lock_acquire(&lock);  
4 thread_create("acquire", PRI_DEFAULT + 10, acquire_thread_func, &lock);
```

- 检查主线程优先级：

```
1 msg("Main thread should have priority %d. Actual priority: %d.",  
2     PRI_DEFAULT + 10, thread_get_priority());
```

- 降低基本优先级：

```
1 msg("Lowering base priority...");  
2 thread_set_priority(PRI_DEFAULT - 10);
```

- 再次检查主线程优先级：

```
1 msg("Main thread should have priority %d. Actual priority: %d.",  
2     PRI_DEFAULT + 10, thread_get_priority());
```

- 释放锁：

```
1 lock_release(&lock);
```

- 检查主线程的最终优先级：

```
1 msg("acquire must already have finished.");  
2 msg("Main thread should have priority %d. Actual priority: %d.",  
3     PRI_DEFAULT - 10, thread_get_priority());
```


总体来说，这个测试验证了在降低线程的基本优先级后，如果该线程被捐赠，那么优先级的降低应该发生在释放锁之后。

为了通过这个测试，我们需要修改 `thread_set_priority()` 函数，使得线程在降低基本优先级后，如果线程被捐赠，那么修改其 `original_priority`，使得优先级的降低发生在释放锁之后。

修改后的 `thread_set_priority()` 函数

```

1 void
2 thread_set_priority (int new_priority)
3 {
4     enum intr_level old_level = intr_disable ();
5     thread_current ()->original_priority = new_priority;
6     if (list_empty (&thread_current ()->locks) || new_priority > thread_current ()
7         ->priority)
8     {
9         thread_current ()->priority = new_priority;
10        thread_yield ();
11    }
12    intr_set_level (old_level);
13 }

```

这样，我们就可以通过测试 `priority-donate-lower` 了。意外地发现，我们还通过了 `priority-sema` 测试。

2.2.7 测试 `priority-condvar`

接下来，我们分析 `priority-condvar` 测试，它的代码如下：

`priority-condvar` 测试

```

1 static thread_func priority_condvar_thread;
2 static struct lock lock;
3 static struct condition condition;
4
5 void
6 test_priority_condvar (void)
7 {
8     int i;
9
10    /* This test does not work with the MLFQS. */
11    ASSERT (!thread_mlfqs);
12
13    lock_init (&lock);
14    cond_init (&condition);
15

```

```
16  thread_set_priority (PRI_MIN);
17  for (i = 0; i < 10; i++)
18  {
19      int priority = PRI_DEFAULT - (i + 7) % 10 - 1;
20      char name[16];
21      snprintf (name, sizeof name, "priority %d", priority);
22      thread_create (name, priority, priority_condvar_thread, NULL);
23  }
24
25  for (i = 0; i < 10; i++)
26  {
27      lock_acquire (&lock);
28      msg ("Signaling...");
29      cond_signal (&condition, &lock);
30      lock_release (&lock);
31  }
32 }
33
34 static void
35 priority_condvar_thread (void *aux UNUSED)
36 {
37     msg ("Thread %s starting.", thread_name ());
38     lock_acquire (&lock);
39     cond_wait (&condition, &lock);
40     msg ("Thread %s woke up.", thread_name ());
41     lock_release (&lock);
42 }
```

以下是对测试步骤的分析：

- 初始化锁和条件变量：

```
1  static struct lock lock;
2  static struct condition condition;
3  lock_init(&lock);
4  cond_init(&condition);
```

- 设置主线程优先级并创建子线程：

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

```
7 }
```

- 向条件变量发送信号：

```
1 for (int i = 0; i < 10; i++) {
2     lock_acquire(&lock);
3     msg("Signaling...");
4     cond_signal(&condition, &lock);
5     lock_release(&lock);
6 }
```

- 子线程等待条件变量：

```
1 static void priority_condvar_thread(void *aux UNUSED) {
2     msg("Thread %s starting.", thread_name());
3     lock_acquire(&lock);
4     cond_wait(&condition, &lock);
5     msg("Thread %s woke up.", thread_name());
6     lock_release(&lock);
7 }
```

总体来说，这个测试验证了在使用条件变量时，线程能够正确地在被唤醒时执行，并且线程的优先级在等待条件变量期间能够正确地起到作用，即条件变量能按照优先级顺序唤醒线程。

为了通过这个测试，我们需要修改 `cond_signal()` 函数，使得线程在被唤醒时，能够根据线程的优先级来进行调度。

修改后的 `cond_signal()` 函数

```
1 void
2 cond_signal (struct condition *cond, struct lock *lock UNUSED)
3 {
4     ASSERT (cond != NULL);
5     ASSERT (lock != NULL);
6     ASSERT (!intr_context ());
7     ASSERT (lock_held_by_current_thread (lock));
8
9     if (!list_empty (&cond->waiters))
10    {
11        list_sort (&cond->waiters, sema_priority_cmp, NULL);
12        sema_up (&list_entry (list_pop_front (&cond->waiters),
13                                     struct semaphore_elem, elem)->semaphore);
14    }
15 }
```

同时，我们还需要实现一个信号量的比较函数，用来比较两个信号量的最高优先级。

信号量比较函数

```

1 bool
2 sema_priority_cmp (const struct list_elem *a, const struct list_elem *b, void *
    aux UNUSED)
3 {
4     struct semaphore_elem *sema_a = list_entry (a, struct semaphore_elem, elem);
5     struct semaphore_elem *sema_b = list_entry (b, struct semaphore_elem, elem);
6     struct thread *ta = list_entry (list_front (&sema_a->semaphore.waiters),
        struct thread, elem);
7     struct thread *tb = list_entry (list_front (&sema_b->semaphore.waiters),
        struct thread, elem);
8
9     return ta->priority > tb->priority;
10 }

```

这样，我们就可以通过测试 `priority-condvar` 了。同时，我们也通过了 `priority-donate-chain` 测试。

这样，我们就通过了优先级调度的所有测试。

2.3 多级反馈队列调度

接下来，我们开始实现多级反馈队列调度。

首先，由于 `pintos` 中没有定义浮点数类型，因此我们需要自己实现浮点数的运算。新建 `fixed_point.h` 文件，定义浮点数运算宏。

`fixed_point.h`

```

1 #ifndef __THREAD_FIXED_POINT_H
2 #define __THREAD_FIXED_POINT_H
3
4 /* 浮点数的基本定义。*/
5 typedef int fixed_t;
6 /* 用于小数部分的 16 位。*/
7 #define FP_SHIFT_AMOUNT 16
8 /* 将一个值转换为浮点数。*/
9 #define FP_CONST(A) ((fixed_t)(A << FP_SHIFT_AMOUNT))
10 /* 添加两个浮点数值。*/
11 #define FP_ADD(A,B) (A + B)
12 /* 将浮点数值 A 和整数值 B 相加。*/
13 #define FP_ADD_MIX(A,B) (A + (B << FP_SHIFT_AMOUNT))
14 /* 减去两个浮点数值。*/
15 #define FP_SUB(A,B) (A - B)

```

```

16 /* 从浮点数值 A 减去整数值 B。*/
17 #define FP_SUB_MIX(A,B) (A - (B << FP_SHIFT_AMOUNT))
18 /* 将浮点数值 A 乘以整数值 B。*/
19 #define FP_MULT_MIX(A,B) (A * B)
20 /* 将浮点数值 A 除以整数值 B。*/
21 #define FP_DIV_MIX(A,B) (A / B)
22 /* 将两个浮点数值相乘。*/
23 #define FP_MULT(A,B) (((fixed_t)(((int64_t) A) * B >> FP_SHIFT_AMOUNT))
24 /* 将两个浮点数值相除。*/
25 #define FP_DIV(A,B) (((fixed_t)((((int64_t) A) << FP_SHIFT_AMOUNT) / B))
26 /* 获取浮点数值的整数部分。*/
27 #define FP_INT_PART(A) (A >> FP_SHIFT_AMOUNT)
28 /* 获取浮点数值的四舍五入整数部分。*/
29 #define FP_ROUND(A) (A >= 0 ? ((A + (1 << (FP_SHIFT_AMOUNT - 1))) >>
    FP_SHIFT_AMOUNT) \
30 : ((A - (1 << (FP_SHIFT_AMOUNT - 1))) >> FP_SHIFT_AMOUNT))
31
32 #endif /* thread/fixed_point.h */

```

然后，我们需要修改线程结构体，定义 recent_cpu 和 nice 值。

修改后的线程结构体

```

1  struct thread
2  {
3      /* Owned by thread.c. */
4      tid_t tid; /* Thread identifier. */
5      enum thread_status status; /* Thread state. */
6      char name[16]; /* Name (for debugging purposes). */
7      uint8_t *stack; /* Saved stack pointer. */
8      int priority; /* Priority. */
9      struct list_elem allelem; /* List element for all threads list. */
10
11     /* Shared between thread.c and synch.c. */
12     struct list_elem elem; /* List element. */
13
14     /* 应该休眠的时间 */
15     int64_t sleep_ticks;
16
17     /* 原本的优先级 */
18     int original_priority; /* Original Priority*/
19
20     /* 持有的锁 */
21     struct list locks; /* List of locks that the thread is
        holding. */

```

```

22
23     /* 等待的锁 */
24     struct lock *waiting;           /* The lock that the thread is waiting
        for. */
25
26     fixed_t recent_cpu;             /* Recent CPU time. */
27
28     int nice;                       /* Nice value. */
29
30 #ifdef USERPROG
31     /* Owned by userprog/process.c. */
32     uint32_t *pagedir;             /* Page directory. */
33 #endif
34
35     /* Owned by thread.c. */
36     unsigned magic;                /* Detects stack overflow. */
37 };

```

接下来，我们定义三个函数，用来计算 `load_avg`、`recent_cpu` 和 `priority`。根据公式进行计算。

计算函数

```

1 void
2 increase_recent_cpu (void)
3 {
4     struct thread *t = thread_current ();
5     if (t == idle_thread)
6         return;
7     t->recent_cpu = FP_ADD_MIX (t->recent_cpu, 1);
8 }
9
10 void
11 mlfqs_update_priority (struct thread *t)
12 {
13     if (t == idle_thread)
14         return;
15
16     t->priority = FP_INT_PART (FP_SUB_MIX (FP_SUB (FP_CONST (PRI_MAX), FP_DIV_MIX
        (t->recent_cpu, 4)), 2 * t->nice));
17     t->priority = t->priority < PRI_MIN ? PRI_MIN : t->priority;
18     t->priority = t->priority > PRI_MAX ? PRI_MAX : t->priority;
19 }
20
21 void
22 update_load_avg (void)

```

```

23 {
24     size_t ready_threads = list_size (&ready_list);
25     if (thread_current () != idle_thread)
26         ready_threads++;
27     load_avg = FP_ADD (FP_DIV_MIX (FP_MULT_MIX (load_avg, 59), 60), FP_DIV_MIX (
        FP_CONST (ready_threads), 60));
28
29     struct thread *t;
30     struct list_elem *e = list_begin (&all_list);
31     for (; e != list_end (&all_list); e = list_next (e))
32     {
33         t = list_entry(e, struct thread, allelem);
34         if (t != idle_thread)
35         {
36             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);
37             mlfqs_update_priority (t);
38         }
39     }
40 }

```

接着，我们修改 `timer_interrupt()` 函数，使得每次时钟中断时，更新 `recent_cpu`，并根据时间决定是否更新 `load_avg` 和 `priority`。

修改后的 `timer_interrupt()` 函数

```

1 static void
2 timer_interrupt (struct intr_frame *args UNUSED)
3 {
4     ticks++;
5     if (thread_mlfqs)
6     {
7         increase_recent_cpu();
8         if (ticks % TIMER_FREQ == 0)
9             update_load_avg ();
10        else if (ticks % 4 == 0)
11            mlfqs_update_priority (thread_current ());
12    }
13    thread_tick ();
14    thread_foreach (thread_tick_sleep, NULL); // 每次中断都调用 thread_tick_sleep
15 }

```

然后，我们需要实现源代码中未实现的与 `nice`，`recent_cpu` 和 `load_avg` 相关的函数。

相关函数

```
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      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 }
```

最后，我们修改一些以前定义的函数，使它们能够正常地在 mlfqs 模式下运行。

修改后的函数

```
1  /* Sets the current thread's priority to NEW_PRIORITY. */
2  void
3  thread_set_priority (int new_priority)
4  {
5      if (thread_mlfqs)
6          return;
7      enum intr_level old_level = intr_disable ();
8      thread_current ()->original_priority = new_priority;
9      if (list_empty (&thread_current ()->locks) || new_priority > thread_current ()
          ->priority)
10     {
```



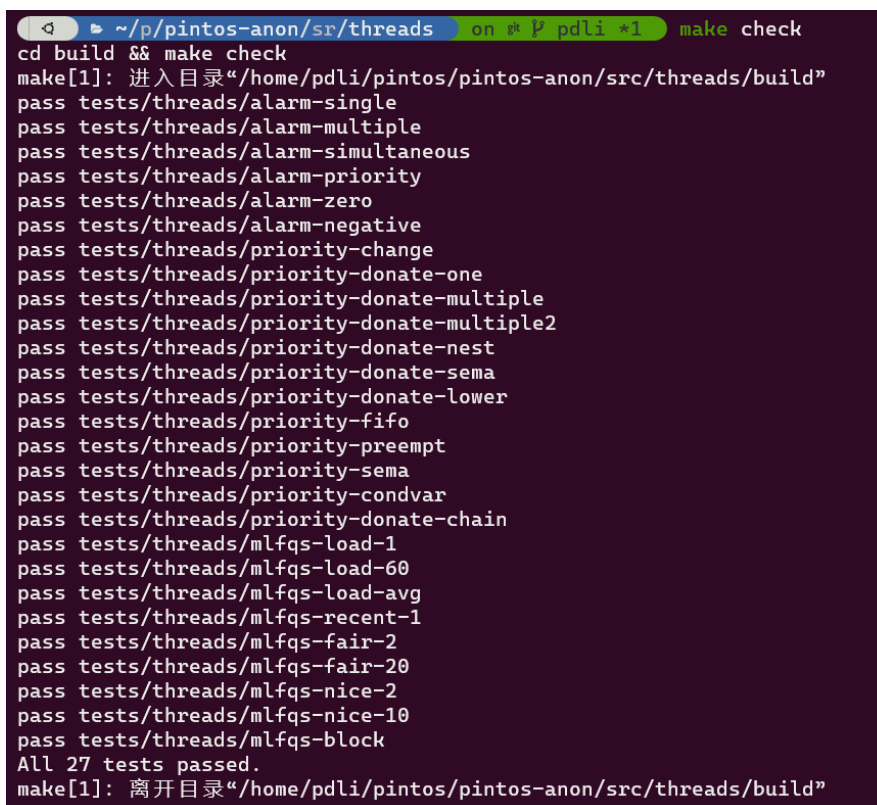
```
11     thread_current ()->priority = new_priority;
12     thread_yield ();
13 }
14 intr_set_level (old_level);
15 }
16
17 void
18 lock_acquire (struct lock *lock)
19 {
20     ASSERT (lock != NULL);
21     ASSERT (!intr_context ());
22     ASSERT (!lock_held_by_current_thread (lock));
23
24     if (lock->holder != NULL && !thread_mlfqs)
25     {
26         thread_current ()->waiting = lock;
27         priority_donate (thread_current (), lock);
28     }
29
30     sema_down (&lock->semaphore);
31
32     if (!thread_mlfqs)
33     {
34         list_insert_ordered (&thread_current ()->locks, &lock->elem,
35                             lock_priority_cmp, NULL);
36         thread_current ()->waiting = NULL;
37         lock->max_priority = thread_current ()->priority;
38     }
39     lock->holder = thread_current ();
40 }
41
42 void
43 lock_release (struct lock *lock)
44 {
45     int max_priority;
46
47     ASSERT (lock != NULL);
48     ASSERT (lock_held_by_current_thread (lock));
49
50     if (!thread_mlfqs) {
51         list_remove (&lock->elem);
52         max_priority = thread_current ()->original_priority;
53         if (!list_empty (&thread_current ()->locks))
```

```
54     list_sort (&thread_current ()->locks, lock_priority_cmp, NULL);
55     struct lock *l = list_entry (list_front (&thread_current ()->locks),
56                                   struct lock, elem);
57     if (l->max_priority > max_priority)
58         max_priority = l->max_priority;
59 }
60 thread_current ()->priority = max_priority;
61 }
62
63 lock->holder = NULL;
64 sema_up (&lock->semaphore);
65 }
```

这样，我们就完成了多级反馈队列调度的实现。

3 实验过程与分析

实验的测试截图如下：



```
~/p/pintos-anon/src/threads on *P pdli *1 make check
cd build && make check
make[1]: 进入目录"/home/pdli/pintos/pintos-anon/src/threads/build"
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.
make[1]: 离开目录"/home/pdli/pintos/pintos-anon/src/threads/build"
```

图 2: 实验的测试截图

在实验中，完成实验的过程如下图所示：

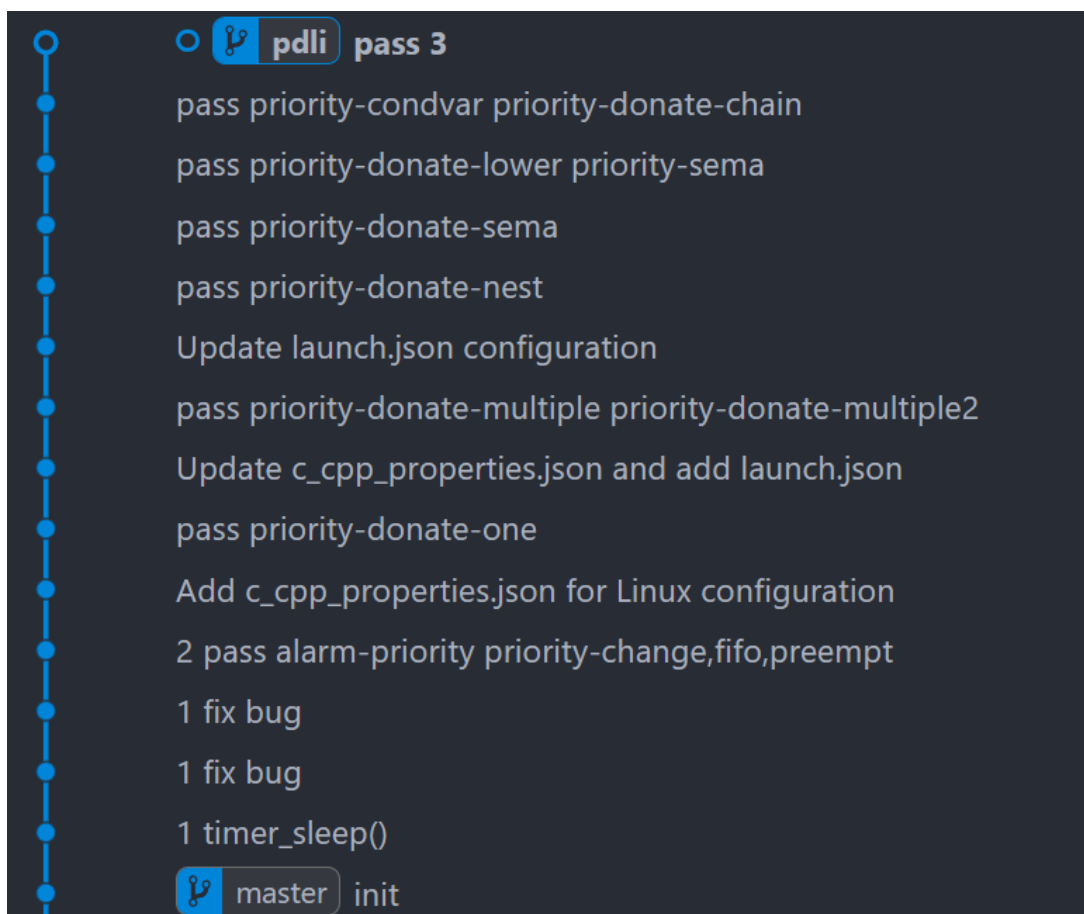


图 3: 实验过程

具体实验分析可以参考上文。

4 实验结果总结

在本次实验中，我重新实现了 `timer_sleep()` 函数，使得线程能够在指定的时间内休眠，并且能够在休眠结束后被唤醒。此外，还实现了优先级调度，使得线程能够根据优先级来进行调度。最后，实现了多级反馈队列调度，使得线程能够根据优先级和时间片来进行调度。

通过本次实验，我对操作系统的线程调度有了更深入的了解，同时也对操作系统的实现有了更深的认识。

5 附录 (源代码)

见 `source-code-pintos-anon.tar.gz` 或 `source-code-pintos-anon.zip`。
也可以访问仓库 <https://github.com/llipenqda/pintos-anon/tree/pdli>。