

华东师范大学软件工程学院实验报告

实验课程：操作系统实践

年级：2023 级

实验成绩：

实验名称：Threads

姓名：顾翌炜

实验编号：project-1

学号：10235101527

实验日期：2024/10/21

指导教师：陈闻杰

组号：01

实验时间：2024/10/21

1 实验目的

试验一的最终任务就是在 threads/中跑 make check 的时候，27 个 test 全 pass。具体内容如下：

- 1) 修复现有的 timer_sleep() 函数，消除忙等现象，提高系统效率。
- 2) 实现一个基于优先级的线程调度系统，确保高优先级线程能够优先执行。

2 实验内容与实验步骤

首先 cd 到 pintos/src/threads 下，输入 make check，测试结果得到：

```
(mlfqs-block) Block thread should have already acquired lock.
(mlfqs-block) end
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
FAIL 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
FAIL tests/threads/priority-fifo
FAIL tests/threads/priority-preempt
FAIL tests/threads/priority-sema
FAIL tests/threads/priority-condvar
FAIL tests/threads/priority-donate-chain
FAIL tests/threads/mlfqs-load-1
FAIL tests/threads/mlfqs-load-60
FAIL tests/threads/mlfqs-load-avg
FAIL tests/threads/mlfqs-recent-1
pass tests/threads/mlfqs-fair-2
pass tests/threads/mlfqs-fair-20
FAIL tests/threads/mlfqs-nice-2
FAIL tests/threads/mlfqs-nice-10
FAIL tests/threads/mlfqs-block
20 of 27 tests failed.
../../tests/Make.tests:26: recipe for target 'check' failed
make[1]: *** [check] Error 1
make[1]: Leaving directory '/home/PKUOS/pintos/src/threads/build'
```

图 1: make check before processing project1

测试点在默认状态下通过了 7/27 个，接下来需要逐步修改完善代码，使本次实验需要的测试点全部通过。

2.1 Mission-1: 重新实现 timer_sleep() 函数

2.1.1 分析原本的 timer_sleep() 函数的实现思路

timer_sleep 函数位于 devices/timer.c 文件中。系统目前使用的是忙等实现方式，即线程不断地循环，直到时间片耗尽。需要更改 timer_sleep 的实现方式。

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

原本的 timer_sleep() 函数

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

在这个函数中，调用了 timer_ticks 函数，查找到这个函数：

timer_ticks() 函数

```
1 /* Returns the number of timer ticks since the OS booted. */
2 int64_t timer_ticks (void)
3 {
4     enum intr_level old_level = intr_disable ();
5     int64_t t = ticks;
6     intr_set_level (old_level);
7     return t;
8 }
```

而在这个 timer_ticks 中，intr_level 和 intr_disable 也可以从代码中找到（此处略）

intr_level 在代码中是一个结构体，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。

get_level() 函数

```
1 /* Returns the current interrupt status. */
2 enum intr_level
3 intr_get_level (void)
4 {
5     uint32_t flags;
```

```

6
7  /* Push the flags register on the processor stack, then pop the
8  value off the stack into `flags'. See [IA32-v2b] "PUSHF"
9  and "POP" and [IA32-v3a] 5.8.1 "Masking Maskable Hardware
10 Interrupts". */
11  asm volatile ("pushfl;_popl_%0" : "=g" (flags));
12
13  return flags & FLAG_IF ? INTR_ON : INTR_OFF;
14 }

```

在这里，没有继续调用函数，所以 `intr_disable()` 函数是调用的最远的地方了。

综上所述：`timer_ticks` 函数中的 `intr_disable` 调用过程为：

首先，`intr_get_level` 函数被用来获取当前的中断级别。随后，`intr_disable` 执行两个关键步骤：一是将中断级别设置为禁止中断，二是返回之前的中断级别。这样，我们就可以理解在 `timer_ticks` 函数的第 4 行，实际上是在执行一个保护操作：确保在获取计时器期间不会被中断，并在操作完成后恢复之前的中断状态，这一过程的中间状态被存储在 `timer_ticks()` 这个函数中的变量 `old_level` 中。

再回到刚刚在看的 `timer_ticks()`，其函数的剩余操作包括：使用局部变量 `t` 获取全局变量 `ticks` 的值。此外，还调用了 `intr_set_level` 函数以恢复之前的中断状态。

set_level() 函数

```

1  /* Enables or disables interrupts as specified by LEVEL and
2  returns the previous interrupt status. */
3  enum intr_level
4  intr_set_level (enum intr_level level)
5  {
6      return level == INTR_ON ? intr_enable () : intr_disable ();
7  }

```

再来观察 `intr_enable` 和 `intr_context` 函数，可以知道：如果之前是允许中断的（`INTR_ON`）则 `enable`，否则就 `disable`。那么我们可以得到：`timer_ticks` 这个函数的完整作用：

获取 `ticks` 的当前值并返回。第 4 行和第 6 行的作用是确保这一过程不会被中断。

从 “`static int64_t ticks;`” 可以知道：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。

timer_sleep() 函数中的部分代码

```

1  while (timer_elapsed (start) < ticks)
2      thread_yield();

```

`timer_sleep` 函数执行时，`start` 获取起始时间。函数断言必须可以被中断，否则会一直死循环。之后，函数进入循环。

在 `timer_sleep` 这个函数中的 `ticks` 作为形参，并不是全局变量，所以观察一下 `timer_elapsed()` 函数。阅读发现：`timer_elapsed` 返回了当前时间距离 `then` 的时间间隔。因此，这个循环实质上是在 `ticks` 时间内不断执行 `thread_yield`。

最后来看 `thread_yield()` 这个函数：

`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 thread_yield (void)
4  {
5      struct thread *cur = thread_current ();
6      enum intr_level old_level;
7
8      ASSERT (!intr_context ());
9
10     old_level = intr_disable ();
11     if (cur != idle_thread)
12         list_push_back (&ready_list, &cur->elem);
13     cur->status = THREAD_READY;
14     schedule ();
15     intr_set_level (old_level);
16 }
```

在这个函数的第 5 行中，用到了 `thread_current()` 函数，其实无需再去阅读具体代码，由其函数名称就可以看出：这个函数的作用就是返回当前线程起始指针。

继续来看 `thread_yield()` 函数，第 8 行的 `ASSERT` 是一个软中断。后面的 `if` 判断语句就是，若当前的线程不是空闲的，那么就调用 `list_push_back` 来将当前的线程放到就绪队列里，并把线程改成 `THREAD_READY` 状态。再调用的 `schedule()` 函数，`schedule()` 其实就是拿下一个线程切换过来继续 `run`。

所以总结来说，`thread_yield()` 其实就是把当前线程扔到就绪队列里，然后重新 `schedule()`。

分析到此，`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。所以它的缺点也就很显而易见了：

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。所以我的想法是：用一种唤醒机制来实现这个函数，使得线程在休眠的时候不会占用 `CPU` 资源。

2.1.2 重新实现 `timer_sleep()` 函数

实现思路：当调用 `timer_sleep()` 函数时，我们会使线程进入阻塞状态，并在线程的数据结构中新增一个成员 `sleep_ticks`，用以追踪线程被暂停的具体时长。接着，我们利用操作系统的时钟中断机制（每次 `tick` 都会触

发一次)，来检查线程的状态。在每次检查过程中，我们会将 sleep_ticks 的值减 1。一旦该值降至 0，即表示线程的等待时间已满，此时将解除线程的阻塞状态，使其恢复运行。

具体代码：

修改后的 timer_sleep() 函数

```

1  /* Sleeps for approximately TICKS timer ticks.  Interrupts must
2  be turned on. */
3  void timer_sleep (int64_t ticks)
4  {
5      if (ticks <= 0)
6      {
7          return; // 当ticks小于0的时候，等待时间已满，直接返回
8      }
9      ASSERT (intr_get_level () == INTR_ON);
10
11     enum intr_level old_level = intr_disable (); // 关闭中断
12     struct thread *current_thread = thread_current ();
13     current_thread->sleep_ticks = ticks; // 设置sleep_ticks
14     thread_block (); // 阻塞当前线程
15     intr_set_level (old_level); // 恢复中断
16 }

```

timer_sleep() 函数中的 intr_disable()、thread_current 等函数的实现无需改变，直接使用原有的函数逻辑即可。

接下来在 Thread 的结构体中加上 sleep_ticks 成员：int64_t sleep_ticks;

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     /* 新增的成员-被暂停的具体时长 */
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 };
```

接下来就需要在函数中初始化这个 `sleep_ticks` 变量，此处 `thread_create()` 函数中，将 `sleep_ticks` 初始化为 0。（下面函数中的第 20 行）

修改后的 `thread_create()` 函数

```
1  tid_t thread_create (const char *name, int priority,
2  thread_func *function, void *aux)
3  {
4      struct thread *t;
5      struct kernel_thread_frame *kf;
6      struct switch_entry_frame *ef;
7      struct switch_threads_frame *sf;
8      tid_t tid;
9
10     ASSERT (function != NULL);
11
12     /* Allocate thread. */
13     t = palloc_get_page (PAL_ZERO);
14     if (t == NULL)
15         return TID_ERROR;
16
17     /* Initialize thread. */
18     init_thread (t, name, priority);
19     tid = t->tid = allocate_tid ();
20     t->sleep_ticks = 0; // 初始化 sleep_ticks 为 0
21
22     /* Stack frame for kernel_thread(). */
23     kf = alloc_frame (t, sizeof *kf);
24     kf->eip = NULL;
25     kf->function = function;
26     kf->aux = aux;
27
28     /* Stack frame for switch_entry(). */
29     ef = alloc_frame (t, sizeof *ef);
30     ef->eip = (void (*) (void)) kernel_thread;
31 }
```

```
32     /* Stack frame for switch_threads(). */
33     sf = alloc_frame (t, sizeof *sf);
34     sf->eip = switch_entry;
35     sf->ebp = 0;
36
37     /* Add to run queue. */
38     thread_unblock (t);
39
40     return tid;
41 }
```

由于我们的 `sleep_ticks` 是在每次检查过程中，将其值减 1 直至 0 时，解除线程的阻塞状态，使其恢复运行。所以需要在调用它的 `timer_sleep()` 函数中，将当前线程的 `sleep_ticks` 设置为其需要等待的时长。（参见前面的代码）

最后一步，修改 `timer_interrupt()` 函数，从而实现对于 `sleep_ticks` 的更新，具体步骤如下：

1. 使用 `thread_foreach()` 函数，遍历所有线程，将 `sleep_ticks` 减 1
2. 如果 `sleep_ticks` 为 0，则将线程唤醒

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  }
```

在这个函数中，有 `thread_tick_sleep` 和 `thread_foreach` 这两个函数，具体如下：

`thread_foreach()` 函数

```
1  /* Invoke function 'func' on all threads, passing along 'aux'.
2  This function must be called with interrupts off. */
3  void thread_foreach (thread_action_func *func, void *aux)
4  {
5      struct list_elem *e;
6
7      ASSERT (intr_get_level () == INTR_OFF);
8
9      for (e=list_begin(&all_list); e!=list_end(&all_list); e=list_next(e))
10     { // 遍历list里所有的线程
```

```

11         struct thread *t = list_entry (e, struct thread, allelem);
12         func (t, aux); // 对每个线程执行func函数
13     }
14 }

```

所以我需要实现一下 `thread_tick_sleep` 函数。

首先在 `thread.h` 文件中声明 `thread_tick_sleep` 函数，然后在 `thread.c` 文件中进行编写。

thread_tick_sleep() 函数

```

1 void thread_tick_sleep (struct thread *t, void *aux UNUSED)
2 {
3     if (t->status==THREAD_BLOCKED && t->sleep_ticks>0) // 如果休眠时间大于0
4     {
5         t->sleep_ticks--; // 休眠时间减一
6         if (t->sleep_ticks == 0) // 如果休眠时间到了
7         {
8             thread_unblock (t); // 唤醒
9         }
10    }
11 }

```

这样 `timer_sleep` 函数唤醒机制就实现了。

2.2 Mission-2: 基于优先级的线程调度

接下来继续解决第二个任务：实现一个基于优先级的线程调度系统，确保高优先级线程能够优先执行。

首先观察 `Thread` 这个结构体（具体代码在 Mission-1 中有展示），发现这个结构体里本身就有个 `priority`，所以我们后续只需要对每个线程的已有的 `priority` 进行比较就可以得到我们想要的结果。并且对于 `priority` 也有一定的约束条件，具体如下：

priority 约束

```

1 /* Thread priorities. */
2 #define PRI_MIN 0 /* Lowest priority. */
3 #define PRI_DEFAULT 31 /* Default priority. */
4 #define PRI_MAX 63 /* Highest priority. */

```

利用之前对 `timer_sleep()` 函数的分析，我们可以知道，这个 Mission 中对于优先级调度的实现的核心思想是：让就绪队列是一个优先级队列（Priority scheduling），那么我们在实现这一目标的时候可以理解为：每次往队列中插入新的线程的时候，是按照优先级来插入的，这样一来就能保证我们的队列是一个优先级队列。

结合代码，我们可以发现，涉及到将线程插入到就绪队列的函数总共有以下三个：

1. `thread_unblock`
2. `init_thread`
3. `thread_yield`

2.2.1 依次修改三个有关插入线程的函数，通过测试 alarm_priority

由刚刚的分析可以得出：只需要在上面三个函数中保持就绪队列是一个优先级队列即可。那接下来就按照顺序来依次分析。

首先来看 `thread_unblock`，这里面的插入线程的代码是：`list_push_back (&ready_list, &t->elem);`，这个的意思是直接将线程放在了队列的尾部，也就是一个先进先出的 FIFO 逻辑，因为每次取出 Thread 都是取出头部，所以我们需要修改一下插入的位置。

阅读 `/lib/kernel/`，研究 `pintos` 里队列的实现：

pintos 里的 list

```
1  /* List element. */
2  struct list_elem
3  {
4      struct list_elem *prev;    /* Previous list element. */
5      struct list_elem *next;    /* Next list element. */
6  };
7
8  /* List. */
9  struct list
10 {
11     struct list_elem head;      /* List head. */
12     struct list_elem tail;      /* List tail. */
13 };
```

这个 list 的实现是一个常见的队列这个数据结构。通过继续翻阅代码，找到了一个叫做 `list_insert_ordered` 的函数，根据函数名就可以知道这是用来给 list 排序的函数，那么我们可以断定，需要修改的就是这个函数了。

list_insert_ordered 函数

```
1  /* Inserts ELEM in the proper position in LIST, which must be
2  sorted according to LESS given auxiliary data AUX.
3  Runs in O(n) average case in the number of elements in LIST. */
4  void list_insert_ordered (struct list *list, struct list_elem *elem,
5                           list_less_func *less, void *aux)
6  {
7
8      struct list_elem *e;
9
10     ASSERT (list != NULL);
11     ASSERT (elem != NULL);
12     ASSERT (less != NULL);
13
14     for (e = list_begin (list); e != list_end (list); e = list_next (e))
15         if (less (elem, e, aux))
16             break;
```

```

15     return list_insert (e, elem);
16 }

```

所以回到一开始看的那个 `thread_unblock` 函数，把 `list_push_back` 改成：`list_insert_ordered(&ready_list, &t->elem, (list_less_func *) &thread_cmp_priority, NULL);`;

然后需要实现的是其中的 `thread_cmp_priority` 函数，具体如下：

thread_cmp_priority 函数

```

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

```

同理，对 `thread_yield` 和 `thread_init` 里的 `list_push_back` 作同样的修改（修改后的代码略）。进行了这些修改以后，`alarm_priority` 这个测试点 `pass` 了。

2.2.2 通过测试 `alarm-priority`, `priority-change`, `priority-fifo` 和 `priority-preempt`

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。

Listing 1: Priority Change Example

```

1  Acceptable output:(priority-change) begin
2      (priority-change) Creating a high-priority thread 2.
3      (priority-change) Thread 2 now lowering priority.
4      (priority-change) Thread 2 should have just lowered its priority.
5      (priority-change) Thread 2 exiting.
6      (priority-change) Thread 2 should have just exited.
7      (priority-change) end
8  Differences in 'diff-u' format:
9      (priority-change) begin
10     (priority-change) Creating a high-priority thread 2.
11     (priority-change) Thread 2 now lowering priority.
12     (priority-change) Thread 2 should have just lowered its priority.
13     (priority-change) Thread 2 exiting.
14     (priority-change) Thread 2 should have just exited.
15     (priority-change) end

```

因此，我们可以总结如下：

在测试中，我们创建了一个名为 `thread1` 的线程，它生成了一个内核线程 `thread2`，其优先级设置为 `PRI_DEFAULT+1`。由于 `thread2` 具有较高的优先级，系统立即将执行权交给了 `thread2`，导致 `thread1` 进入

阻塞状态。在 thread2 执行期间，它调用了 changing_thread 函数，将自己的优先级调整为 PRI_DEFAULT-1，这样一来，thread1 的优先级就相对更高了。

当 thread2 在最后一个消息输出处被阻塞时，控制权转交给了 thread1。随后，thread1 将自己的优先级进一步降低到 PRI_DEFAULT-2，使得 thread2 的优先级再次高于 thread1。于是，执行权又回到了 thread2，它继续执行并输出 thread1 的消息。这一系列的优先级调整和线程切换，最终产生了图中所示的测试输出结果。

所以我们可以知道：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。所以直接在线程设置优先级的时候调用 thread_yield 即可，相当于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。（当然，如果在创建线程的时候，如果新创建的线程比主线程优先级高的话也要调用 thread_yield）

thread_set_priority 函数

```
1  /* Sets the current thread's priority to NEW_PRIORITY. */
2  void thread_set_priority (int new_priority)
3  {
4      thread_current ()->priority = new_priority;
5      thread_yield ();
6  }
```

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。在函数的最后把创建的线程 unblock 了之后加上和主线程的优先级的比较函数：

thread_create 函数结尾补充：

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

完整的 thread_create() 函数如下所示：

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 }
```

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

2.2.3 通过测试 priority-priority-*

接下来需要实现的是优先级捐赠。同理先阅读测试文件，分析需要进行哪些修改。
先来关注 priority-donate-one 的测试代码：

thread_create 函数结尾补充:

```
1 void test_priority_donate_one (void)
2 {
3     struct lock lock;
4
5     /* This test does not work with the MLFQS. */
6     ASSERT (!thread_mlfqs);
7
8     /* Make sure our priority is the default. */
9     ASSERT (thread_get_priority () == PRI_DEFAULT);
10
11     lock_init (&lock);
12     lock_acquire (&lock);
13     thread_create ("acquire1", PRI_DEFAULT + 1, acquire1_thread_func, &lock);
14     msg ("This_thread_should_have_priority%d. Actual_priority:%d.",
15         PRI_DEFAULT + 1, thread_get_priority ());
16     thread_create ("acquire2", PRI_DEFAULT + 2, acquire2_thread_func, &lock);
17     msg ("This_thread_should_have_priority%d. Actual_priority:%d.",
18         PRI_DEFAULT + 2, thread_get_priority ());
19     lock_release (&lock);
20     msg ("acquire2, acquire1 must already have finished, in that order.");
21     msg ("This should be the last line before finishing this test.");
22 }
23
24 static void acquire1_thread_func (void *lock_)
25 {
26     struct lock *lock = lock_;
27
28     lock_acquire (lock);
29     msg ("acquire1: got the lock");
30     lock_release (lock);
31     msg ("acquire1: done");
32 }
33
34 static void acquire2_thread_func (void *lock_)
35 {
36     struct lock *lock = lock_;
37
38     lock_acquire (lock);
39     msg ("acquire2: got the lock");
40     lock_release (lock);
41     msg ("acquire2: done");
42 }
```

分析过程如下：最初主线程主动获取了一个 lock（第 11 行），它的优先级被设定为 PRI_DEFAULT。随后，我们在第 13 和第 16 行主动给创建了两个优先级更高的新的线程，名为 acquire1 和 acquire2，即 PRI_DEFAULT+1 和 PRI_DEFAULT+2，这两个线程被设计为在获取锁时被阻塞。

而后在第 19 行，主线程通过 lock_release 主动释放锁，但是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（由于 acquire2 的优先级更高，它首先获取了锁，acquire1 在 acquire2 线程之后获取了锁）。

再通过 msg 打印测试结果，确认 acquire2 线程先完成，然后再是 acquire1。

简而言之，这项测试旨在确认当一个线程释放锁时，它是否能够正确地将优先级传递给正在等待该锁的其他线程，以确保线程按照优先级来顺序获取锁。

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

接下来修改 lock_release() 函数，要让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。我的想法是在 Thread 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 };

```

随后要做的就是修改 `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，也就是刚刚在结构体里新增的成员变量 `int original_priority`。

修改后的 `init_thread()` 函数

```

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

```

20 }

修改完创建的函数，现在需要来修改释放锁的时候的函数了：

修改后的 lock_release() 函数

```

1 void lock_release (struct lock *lock)
2 {
3     ASSERT (lock != NULL);
4     ASSERT (lock_held_by_current_thread (lock));
5
6     lock->holder = NULL;
7     sema_up (&lock->semaphore);
8
9     thread_set_priority (thread_current ()->original_priority);
10 }

```

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

```

这样，测试 priority-donate-one 就可以 pass 了。

2.2.4 通过测试 priority-donate-multiple 和 priority-donate-multiple2

首先分析这两个测试点的测试函数，具体如下：

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 test_priority_donate_multiple2 (void)
6 {
7     struct lock a, b;
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 (&a);
16    lock_init (&b);
17
18    lock_acquire (&a);
19    lock_acquire (&b);
20
21    thread_create ("a", PRI_DEFAULT + 3, a_thread_func, &a);
22    msg ("Main_thread_should_have_priority_%d. Actual_priority:_%d.",
23    PRI_DEFAULT + 3, thread_get_priority ());
24
25    thread_create ("c", PRI_DEFAULT + 1, c_thread_func, NULL);
26
27    thread_create ("b", PRI_DEFAULT + 5, b_thread_func, &b);
28    msg ("Main_thread_should_have_priority_%d. Actual_priority:_%d.",
29    PRI_DEFAULT + 5, thread_get_priority ());
30
31    lock_release (&a);
32    msg ("Main_thread_should_have_priority_%d. Actual_priority:_%d.",
33    PRI_DEFAULT + 5, thread_get_priority ());
34
35    lock_release (&b);
36    msg ("Threads_b,a,c_should_have_just_finished_in_that_order.");
37    msg ("Main_thread_should_have_priority_%d. Actual_priority:_%d.",
38    PRI_DEFAULT, thread_get_priority ());
39 }
40
41 static void a_thread_func (void *lock_)
42 {
43     struct lock *lock = lock_;
44
45     lock_acquire (lock);
46     msg ("Thread_a_acquired_lock_a.");
47     lock_release (lock);
48     msg ("Thread_a_finished.");
```

```

49 }
50
51 static void b_thread_func (void *lock_)
52 {
53     struct lock *lock = lock_;
54
55     lock_acquire (lock);
56     msg ("Thread_b_acquired_lock_b.");
57     lock_release (lock);
58     msg ("Thread_b_finished.");
59 }
60
61 static void c_thread_func (void *a_ UNUSED)
62 {
63     msg ("Thread_c_finished.");
64 }

```

1. 在这个函数中，首先初始化了 a 和 b 两个锁；
2. 随后主线程主动获取了 a 和 b 这两个锁；
3. 再创建三个优先级更高的线程，分别是 b>a>c；
4. 随后主线程主动释放 a 和 b 两个锁；
5. 打印测试结果

打印测试结果

```

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。

这项测试旨在确认，在多锁环境中，线程释放锁时，是否能够正确地将其优先级调整为其他未释放锁中的最高优先级。这样可以保证线程能够依据它们的优先级顺序来获得锁。

那么接下来开始修改代码：为了确保能够通过这项测试，我们需要首先跟踪记录线程所持有的锁。为此，我

们将在线程的数据结构中添加一个新的成员变量，`struct list locks`。这个列表将用于存储线程当前持有的所有锁的信息。

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 };

```

为了确保锁的获取能够根据线程的优先级进行，我们需要对锁的数据结构进行扩展。具体来说，我们需要添加两个新的字段：

1. `max_priority`: 这是一个整数类型的字段，用于跟踪所有请求该锁的线程中最高的优先级。通过追踪这一信息，我们可以在锁释放时，确定哪个线程应该获得下一个获取锁的机会。

2. `elem`: 这是一个结构体类型的字段，代表一个列表元素。它将用于将锁的请求插入到一个列表中，以便我们可以根据线程的优先级来排序和管理这些请求。

通过这样的设计，我们可以确保在多线程环境中，锁的分配能够遵循线程优先级的规则，从而优化线程的调度和执行效率。

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  };

```

接下来，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  }

```

修改完锁的结构体，现在我们需要修改线程结构体，添加成员变量 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     /* 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 };

```

同样的，修改线程的初始化函数 `init_thread()`，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 init_thread (struct thread *t, const char *name, int priority)
2  {
3      enum intr_level old_level;
4
5      ASSERT (t != NULL);
6      ASSERT (PRI_MIN <= priority && priority <= PRI_MAX);
7      ASSERT (name != NULL);
8
9      memset (t, 0, sizeof *t);
10     t->status = THREAD_BLOCKED;
11     strncpy (t->name, name, sizeof t->name);
12     t->stack = (uint8_t *) t + PGSIZE;
13     t->priority = priority;
14     t->original_priority = priority;
15     t->magic = THREAD_MAGIC;
16
17     list_init (&t->locks); // 初始化锁列表
18
19     old_level = intr_disable ();
20     list_insert_ordered (&all_list, &t->allelem, thread_priority_cmp, NULL);
21     intr_set_level (old_level);
22 }

```

既然要比较线程的优先级，那么我们就需要写一个优先级的比较函数，此处命名为 `lock_priority_cmp`，返回值是一个 `boolean`。

锁的比较函数

```

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

```

下一步，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 lock_acquire (struct lock *lock)
2  {
3      ASSERT (lock != NULL);
4      ASSERT (!intr_context ());
5      ASSERT (!lock_held_by_current_thread (lock));
6
7      if (lock->holder != NULL && lock->holder->priority < thread_current ()->
        priority)
8      {
9          lock->holder->priority = thread_current ()->priority;
10         if (lock->max_priority < thread_current ()->priority)
11             lock->max_priority = thread_current ()->priority;
12     }
13
14     sema_down (&lock->semaphore);
15
16     list_insert_ordered (&thread_current ()->locks, &lock->elem,
        lock_priority_cmp, NULL);
17     lock->holder = thread_current ();
18 }

```

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 lock_release (struct lock *lock)
2  {
3      int max_priority;
4

```

```

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

```

这样，测试 priority-donate-multiple, priority-donate-multiple2 就可以 pass 掉了。

2.2.5 通过测试 priority-donate-nest

老规矩，我们先来分析 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 test_priority_donate_nest (void)
5  {
6      struct lock a, b;
7      struct locks locks;
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 (&a);
16     lock_init (&b);
17

```



```
18     lock_acquire (&a);
19
20     locks.a = &a;
21     locks.b = &b;
22     thread_create ("medium", PRI_DEFAULT + 1, medium_thread_func, &locks);
23     thread_yield ();
24     msg ("Low_thread_should_have_priority%d. Actual_priority:%d.",
25     PRI_DEFAULT + 1, thread_get_priority ());
26
27     thread_create ("high", PRI_DEFAULT + 2, high_thread_func, &b);
28     thread_yield ();
29     msg ("Low_thread_should_have_priority%d. Actual_priority:%d.",
30     PRI_DEFAULT + 2, thread_get_priority ());
31
32     lock_release (&a);
33     thread_yield ();
34     msg ("Medium_thread_should_just_have_finished.");
35     msg ("Low_thread_should_have_priority%d. Actual_priority:%d.",
36     PRI_DEFAULT, thread_get_priority ());
37 }
38
39 static void medium_thread_func (void *locks_)
40 {
41     struct locks *locks = locks_;
42
43     lock_acquire (locks->b);
44     lock_acquire (locks->a);
45
46     msg ("Medium_thread_should_have_priority%d. Actual_priority:%d.",
47     PRI_DEFAULT + 2, thread_get_priority ());
48     msg ("Medium_thread_got_the_lock.");
49
50     lock_release (locks->a);
51     thread_yield ();
52
53     lock_release (locks->b);
54     thread_yield ();
55
56     msg ("High_thread_should_have_just_finished.");
57     msg ("Middle_thread_finished.");
58 }
59
60 static void 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 }

```

来分析一下这个函数的实现过程：

1. 与前一个测试的过程基本一致，先初始化了 a 和 b 两个锁；同时，定义一个结构 locks 包含指向这两个锁的指针。
 2. 主线程主动获取锁；
 3. 创建两个新的线程；
 4. 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。
 5. 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。
 6. medium 线程执行。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。
 7. high 线程执行。线程 high 获取锁 b，执行完毕后释放锁。
- 总结：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。

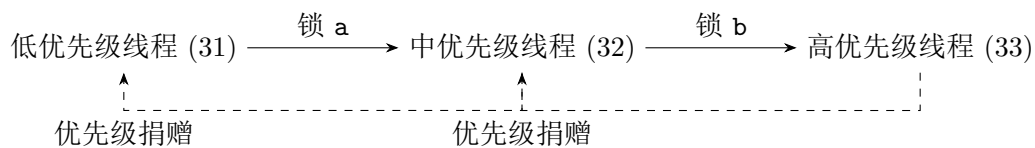


图 2: 线程关系图

通过分析，我们可以得知，优先级捐赠需要递归地进行，也就是当线程 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()`，这个函数是用来递归地进行优先级捐赠。

优先级捐赠函数

```

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

```

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，那么就记录线程等待的锁，

并递归地进行优先级捐赠，也就是递归调用刚刚写的 `priority_donate` 函数。

修改后的 `lock_acquire()` 函数

```

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

这样之后，`priority-donate-nest` 的测试就可以 pass 掉了。

2.2.6 通过测试 `priority-donate-sema`

进行下一步的分析，我们来解决 `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 test_priority_donate_sema (void)
12 {
13     struct lock_and_sema ls;
```

```
14
15  /* This test does not work with the MLFQS. */
16  ASSERT (!thread_mlfqs);
17
18  /* Make sure our priority is the default. */
19  ASSERT (thread_get_priority () == PRI_DEFAULT);
20
21  lock_init (&ls.lock);
22  sema_init (&ls.sema, 0);
23  thread_create ("low", PRI_DEFAULT + 1, l_thread_func, &ls);
24  thread_create ("med", PRI_DEFAULT + 3, m_thread_func, &ls);
25  thread_create ("high", PRI_DEFAULT + 5, h_thread_func, &ls);
26  sema_up (&ls.sema);
27  msg ("Main_thread_finished.");
28 }
29
30 static void l_thread_func (void *ls_)
31 {
32     struct lock_and_sema *ls = ls_;
33
34     lock_acquire (&ls->lock);
35     msg ("Thread_L_acquired_lock.");
36     sema_down (&ls->sema);
37     msg ("Thread_L_downed_semaphore.");
38     lock_release (&ls->lock);
39     msg ("Thread_L_finished.");
40 }
41
42 static void m_thread_func (void *ls_)
43 {
44     struct lock_and_sema *ls = ls_;
45
46     sema_down (&ls->sema);
47     msg ("Thread_M_finished.");
48 }
49
50 static void h_thread_func (void *ls_)
51 {
52     struct lock_and_sema *ls = ls_;
53
54     lock_acquire (&ls->lock);
55     msg ("Thread_H_acquired_lock.");
56
57     sema_up (&ls->sema);
```

```

58     lock_release (&ls->lock);
59     msg ("Thread_H_finished.");
60 }

```

来分析一下这个函数的实现过程：

1. 在测试开始时，通过 `lock_init (&ls.lock)` 初始化了一个包含锁和信号量的结构体；
2. 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。
3. 接下来通过 `static void l_thread_func(void *ls_)`、`static void m_thread_func(void *ls_)` 和 `static void h_thread_func(void *ls_)`；
线程执行的互动具体如下：
 - 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。
4. 主线程通过调用 `sema_up` 释放了信号量；
5. 最后通过 `msg("Main thread finished.");` 来打印测试结果，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。

其实逻辑就是在 `sema_up()` 函数中，在最后的释放信号量的时候进行调度。由此来修改我们的 `sema_up()` 函数。具体如下：

修改后的 `sema_up()` 函数

```

1  void sema_up (struct semaphore *sema)
2  {
3      enum intr_level old_level;
4
5      ASSERT (sema != NULL);
6
7      old_level = intr_disable ();
8      if (!list_empty (&sema->waiters))
9      {
10         list_sort (&sema->waiters, thread_priority_cmp, NULL);
11         thread_unblock (list_entry (list_pop_front (&sema->waiters),
12             struct thread, elem));
13     }
14     sema->value++;
15     intr_set_level (old_level);
16     thread_yield (); // 释放信号量后进行调度
17 }

```

这样我们就可以让测试 priority-donate-sema pass 了。

2.2.7 通过测试 priority-donate-lower

继续分析 priority-donate-lower 测试，还是先来看他的源代码，它的实现原理如下：

priority-donate-lower 测试

```
1 static thread_func acquire_thread_func;
2
3 void test_priority_donate_lower (void)
4 {
5     struct lock lock;
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 (&lock);
14    lock_acquire (&lock);
15    thread_create ("acquire", PRI_DEFAULT + 10, acquire_thread_func, &lock);
16    msg ("Main_thread_should_have_priority%d. Actual_priority:%d.",
17        PRI_DEFAULT + 10, thread_get_priority ());
18
19    msg ("Lowering_base_priority...");
20    thread_set_priority (PRI_DEFAULT - 10);
21    msg ("Main_thread_should_have_priority%d. Actual_priority:%d.",
22        PRI_DEFAULT + 10, thread_get_priority ());
23    lock_release (&lock);
24    msg ("acquire_must_already_have_finished.");
25    msg ("Main_thread_should_have_priority%d. Actual_priority:%d.",
26        PRI_DEFAULT - 10, thread_get_priority ());
27 }
28
29 static void
30 acquire_thread_func (void *lock_)
31 {
32     struct lock *lock = lock_;
33
34    lock_acquire (lock);
35    msg ("acquire:got_the_lock");
36    lock_release (lock);
37    msg ("acquire:done");
```

38 }

之后还是先来分析这个函数的实现过程：（其实这个函数不用过于理解每个函数的意义，仅仅通过 msg 输出的语句就可以知道每一步的实现目的了）

1. 通过 `lock_init(&lock);` 来初始化一个锁，通过 `thread_create(.....)` 来创建一个线程；
2. 检查主线程优先级并输出到终端；
3. 通过 `thread_set_priority(PRI_DEFAULT - 10);` 来强制减少这个线程的优先级；
4. 随后再次检查优先级，并输出到终端；
5. 通过 `lock_release(&lock);` 来释放锁；
6. 最后，再次检查主线程的最终优先级。

总结来说：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。

根据我们的需求和对于代码的阅读，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 thread_set_priority (int new_priority)
2 {
3     enum intr_level old_level = intr_disable ();
4     thread_current ()->original_priority = new_priority;
5     if (list_empty (&thread_current ()->locks) || new_priority > thread_current
        ()->priority)
6     {
7         thread_current ()->priority = new_priority;
8         thread_yield ();
9     }
10    intr_set_level (old_level);
11 }
```

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

但是同时也意外地发现，`priority-sema` 测试也 pass 掉了。

2.2.8 通过测试 `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 test_priority_condvar (void)
6 {
7     int i;
8
9     /* This test does not work with the MLFQS. */
10    ASSERT (!thread_mlfqs);
11
12    lock_init (&lock);
13    cond_init (&condition);
14
15    thread_set_priority (PRI_MIN);
16    for (i = 0; i < 10; i++)
17    {
18        int priority = PRI_DEFAULT - (i + 7) % 10 - 1;
19        char name[16];
20        snprintf (name, sizeof name, "priority_%d", priority);
21        thread_create (name, priority, priority_condvar_thread, NULL);
22    }
23
24    for (i = 0; i < 10; i++)
25    {
26        lock_acquire (&lock);
27        msg ("Signaling...");
28        cond_signal (&condition, &lock);
29        lock_release (&lock);
30    }
31 }
32
33 static void priority_condvar_thread (void *aux UNUSED)
34 {
35    msg ("Thread_%s_starting.", thread_name ());
36    lock_acquire (&lock);
37    cond_wait (&condition, &lock);
38    msg ("Thread_%s_woke_up.", thread_name ());
39    lock_release (&lock);
40 }
```

以下是对测试步骤的分析:

1. 通过 `lock_init(&lock);` 来初始化锁和条件变量;
2. 通过 `thread_set_priority(PRI_MIN);` 和 `thread_create()` 来设置主线程优先级并创建子线程
3. 通过 `cond_signal(&condition, &lock);` 来向条件变量发送信号, 并在后续释放锁
4. 最后的 `priority_condvar_thread` 部分就是子线程等待条件变量

总结：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。

所以可见：我们需要修改的是 `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 cond_signal (struct condition *cond, struct lock *lock UNUSED)
2 {
3     ASSERT (cond != NULL);
4     ASSERT (lock != NULL);
5     ASSERT (!intr_context ());
6     ASSERT (lock_held_by_current_thread (lock));
7
8     if (!list_empty (&cond->waiters))
9     {
10         list_sort (&cond->waiters, sema_priority_cmp, NULL);
11         sema_up (&list_entry (list_pop_front (&cond->waiters),
12             struct semaphore_elem, elem)->semaphore);
13     }
14 }
```

在上面这个 `cond_signal` 函数中的 `list_sort` 中调用了：`sema_priority_cmp` 这个信号量比较大小的函数，具体实现如下所示：

信号量比较函数

```
1 bool sema_priority_cmp (const struct list_elem *a, const struct list_elem *b,
2     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),
7         struct thread, elem);
8     struct thread *tb = list_entry (list_front (&sema_b->semaphore.waiters),
9         struct thread, elem);
10
11     return ta->priority > tb->priority;
12 }
```

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。这样一来，我们就通过了优先级调度的所有测试。

2.2.9 总结该 Mission:

1. 锁获取与优先级调整:

当一个线程请求一个锁时,如果当前持有该锁的线程的优先级低于请求线程,系统将提升持有锁线程的优先级。如果该锁还被其他锁锁定,系统将递归地进行优先级捐赠。当线程释放锁后,其优先级将恢复至捐赠前的设置。

2. 多线程捐赠优先级维护:

如果一个线程接受了来自多个线程的优先级捐赠,其优先级将保持为所有捐赠优先级中的最大值,这一规则适用于锁的获取和释放操作。

3. 优先级设置与捐赠状态管理:

在设置线程优先级时,如果该线程正处于被捐赠状态,系统将记录其原始优先级(original_priority)。如果新设置的优先级高于当前优先级,则更新为新优先级;否则,在捐赠状态结束后,优先级将恢复至原始值。

4. 锁释放与优先级调整:

在释放锁时,系统需要考虑其他被捐赠的优先级和当前优先级,以确保优先级调整的准确性。

5. 信号量等待队列优化:

将信号量的等待队列实现为优先级队列,以优化等待和唤醒机制。

6. 条件变量等待者队列优化:

将条件变量(condition)的等待者队列实现为优先级队列,以提升条件等待和通知的效率。

7. 锁释放与抢占:

当释放锁导致优先级发生变化时,可能会触发线程抢占,以确保高优先级线程能够及时获得处理

2.3 Mission-3: 多级反馈队列调度

这个部分要求我们构建一个实现 MLFQ 的高级调度器,简而言之,维护了 64 个队列,每个队列对应一个特定的优先级,范围从 PRI_MIN 至 PRI_MAX。系统通过一系列公式计算线程的当前优先级。在调度过程中,系统会优先从高优先级队列中选择线程进行执行。值得注意的是,线程的优先级是动态变化的,它会随着操作系统的运行数据而调整。

这一计算过程涉及到浮点数运算,而 Pintos 操作系统本身并未提供这一功能。因此,我们需要自行实现这部分功能。

而我的实现思路是:在 timer_interrupt 中,系统会定期更新线程的优先级。具体来说,每经过 TIMER_FREQ 个时间单位,系统就会重新计算一次系统的平均负载(load_avg)以及所有线程的 recent_cpu 值。每累积 4 个 timer_ticks,即每 4 个定时器中断周期,系统会更新一次线程的优先级。在每个 timer_tick 期间,当前运行线程的 recent_cpu 值会增加 1。

尽管这里提到了维护 64 个优先级队列的调度机制,但其实质仍然是基于优先级的调度。因此,我们可以保留之前编写的优先级调度代码。同时,我们将移除优先级捐赠功能(之前与 donate 相关的代码已经针对需要的地方增加了 thread_mlfqs 的判断逻辑)。

首先来看实现在 fixed_point.h 中的浮点运算逻辑:

fixed_point.h

```

1  #ifndef __THREAD_FIXED_POINT_H
2  #define __THREAD_FIXED_POINT_H
3
4  /* Basic definitions of fixed point. */
5  typedef int fixed_t;
6  /* 16 LSB used for fractional part. */
7  #define FP_SHIFT_AMOUNT 16
8  /* Convert a value to fixed-point value. */
9  #define FP_CONST(A) ((fixed_t)(A << FP_SHIFT_AMOUNT))
10 /* Add two fixed-point value. */
11 #define FP_ADD(A,B) (A + B)
12 /* Add a fixed-point value A and an int value B. */
13 #define FP_ADD_MIX(A,B) (A + (B << FP_SHIFT_AMOUNT))
14 /* Subtract two fixed-point value. */
15 #define FP_SUB(A,B) (A - B)
16 /* Subtract an int value B from a fixed-point value A */
17 #define FP_SUB_MIX(A,B) (A - (B << FP_SHIFT_AMOUNT))
18 /* Multiply a fixed-point value A by an int value B. */
19 #define FP_MULT_MIX(A,B) (A * B)
20 /* Divide a fixed-point value A by an int value B. */
21 #define FP_DIV_MIX(A,B) (A / B)
22 /* Multiply two fixed-point value. */
23 #define FP_MULT(A,B) (((fixed_t)(((int64_t) A) * B >> FP_SHIFT_AMOUNT))
24 /* Divide two fixed-point value. */
25 #define FP_DIV(A,B) (((fixed_t)((((int64_t) A) << FP_SHIFT_AMOUNT) / B))
26 /* Get integer part of a fixed-point value. */
27 #define FP_INT_PART(A) (A >> FP_SHIFT_AMOUNT)
28 /* Get rounded integer of a fixed-point value. */
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 */

```

这里将 FP_SHIFT_AMOUNT 设置为 16，意思就是浮点数的小数部分是 16。所以无论什么运算一定要维持整数部分从第 17 位开始。

首先来实现 timer_interrupt(), 加入以下代码:

函数 timer_interrupt()

```

1  if (thread_mlfqs)
2  {
3      thread_mlfqs_increase_recent_cpu_by_one ();
4      if (ticks % TIMER_FREQ == 0)
5          thread_mlfqs_update_load_avg_and_recent_cpu ();

```

```

6     else if (ticks % 4 == 0)
7         thread_mlfqs_update_priority (thread_current ());
8 }

```

再其中还有一个暂未完成的 mlfqs_increase_recent_cpu_by_one() 函数:

函数 thread_mlfqs_increase_recent_cpu_by_one

```

1  /* Increase recent_cpu by 1. */
2  void
3  thread_mlfqs_increase_recent_cpu_by_one (void)
4  {
5      ASSERT (thread_mlfqs);
6      ASSERT (intr_context ());
7
8      struct thread *current_thread = thread_current ();
9      if (current_thread == idle_thread)
10         return;
11     current_thread->recent_cpu = FP_ADD_MIX (current_thread->recent_cpu, 1);
12 }

```

这里调用的运算都是浮点运算, 利用 Project 给好的算数运算逻辑来实现:

thread_mlfqs_update_priority()

```

1  /* Every per second to refresh load_avg and recent_cpu of all threads. */
2  void
3  thread_mlfqs_update_load_avg_and_recent_cpu (void)
4  {
5      ASSERT (thread_mlfqs);
6      ASSERT (intr_context ());
7
8      size_t ready_threads = list_size (&ready_list);
9      if (thread_current () != idle_thread)
10         ready_threads++;
11     load_avg = FP_ADD (FP_DIV_MIX (FP_MULT_MIX (load_avg, 59), 60), FP_DIV_MIX (
        FP_CONST (ready_threads), 60));
12
13     struct thread *t;
14     struct list_elem *e = list_begin (&all_list);
15     for (; e != list_end (&all_list); e = list_next (e))
16     {
17         t = list_entry(e, struct thread, allelem);
18         if (t != idle_thread)
19         {
20             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);
21     thread_mlfqs_update_priority (t);
22     }
23 }
24 }
```

最后还需要实现一下 thread_mlfqs_update_priority() 函数

thread_mlfqs_update_priority() 函数

```

1  /* Update priority. */
2  void
3  thread_mlfqs_update_priority (struct thread *t)
4  {
5      if (t == idle_thread)
6          return;
7
8      ASSERT (thread_mlfqs);
9      ASSERT (t != idle_thread);
10
11     t->priority = FP_INT_PART (FP_SUB_MIX (FP_SUB (FP_CONST (PRI_MAX),
12         FP_DIV_MIX (t->recent_cpu, 4)), 2 * t->nice));
13     t->priority = t->priority < PRI_MIN ? PRI_MIN : t->priority;
14     t->priority = t->priority > PRI_MAX ? PRI_MAX : t->priority;
15 }
```

我们 mission3 的主体逻辑就已经完成了，但是仍有一些细节需要修改：

thread 结构体加入以下成员：

thread 结构体加入新变量

```

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

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

init_thread 结构体加入新代码

```

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

再继续完善一下系统给出的代码中不足之处：

仍然需要修改的四个函数

```

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     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 }

```

最后补充刚刚用到的变量以及代码：

thread.c 中加入全局变量：

thread.c 加入新全局变量

```
1 fixed_t load_avg;
```

对应的，在 thread_start 中初始化：

在 thread_start 中初始化

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

所以至此，我们已经基本上完成了对于 Project-1 的所有需求实现。接下来就是对于本次实验的过程和分析。

3 实验环境

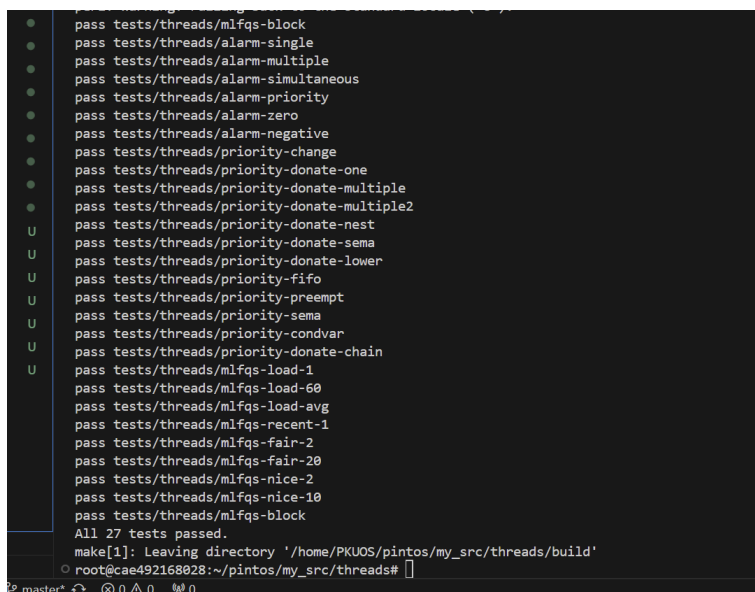
为了确保实验的顺利进行，我们需要搭建一个稳定且符合要求的实验环境。以下是实验所需的详细环境配置：

虚拟机管理软件：实验中使用的虚拟机管理软件为 Oracle VM VirtualBox。这是一个功能强大的虚拟化解决方案，支持多种操作系统的虚拟机运行，并且是免费提供的。

虚拟机：我们已经配置好了运行 pintos 的虚拟机。虚拟机中安装了所有必要的依赖，包括但不限于编译工具链、调试工具和模拟器。

4 实验过程与分析

首先先展示最后的实验的 make check 的结果：



```
pass tests/threads/mlqqs-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/mlqqs-load-1
pass tests/threads/mlqqs-load-60
pass tests/threads/mlqqs-load-avg
pass tests/threads/mlqqs-recent-1
pass tests/threads/mlqqs-fair-2
pass tests/threads/mlqqs-fair-20
pass tests/threads/mlqqs-nice-2
pass tests/threads/mlqqs-nice-10
pass tests/threads/mlqqs-block
All 27 tests passed.
make[1]: Leaving directory '/home/PKUOS/pintos/my_src/threads/build'
root@cae492168028:~/pintos/my_src/threads#
```

图 3: make check after processing project1

4.1 实验过程

在实验中，完成实验的过程如上文所示：

1. Mission-1: 重新实现 timer_sleep() 函数

分析原本的 timer_sleep() 函数的实现思路

重新实现 timer_sleep() 函数

2. Mission-2: 重新实现 timer_sleep() 函数

依次修改三个有关插入线程的函数，通过测试 alarm_priority

通过测试 alarm-priority, priority-change, priority-fifo 和 priority-preempt

通过测试 `priority-priority-*`
通过测试 `priority-donate-multiple` 和 `priority-donate-multiple2`
通过测试 `priority-donate-nest`
通过测试 `priority-donate-sema`
通过测试 `priority-donate-lower`(顺便通过了测试 `priority-sema`)
通过测试 `priority-condvar`(顺便通过了测试 `priority-donate-chain`)

4.2 实验分析

具体实验分析可以参考上文。

通过 `diff -urNa /original-pintos/ /pintos/ > project1.527.GuYiwei.patch` 之后, 得到补丁文件, 部分内容如下图所示:

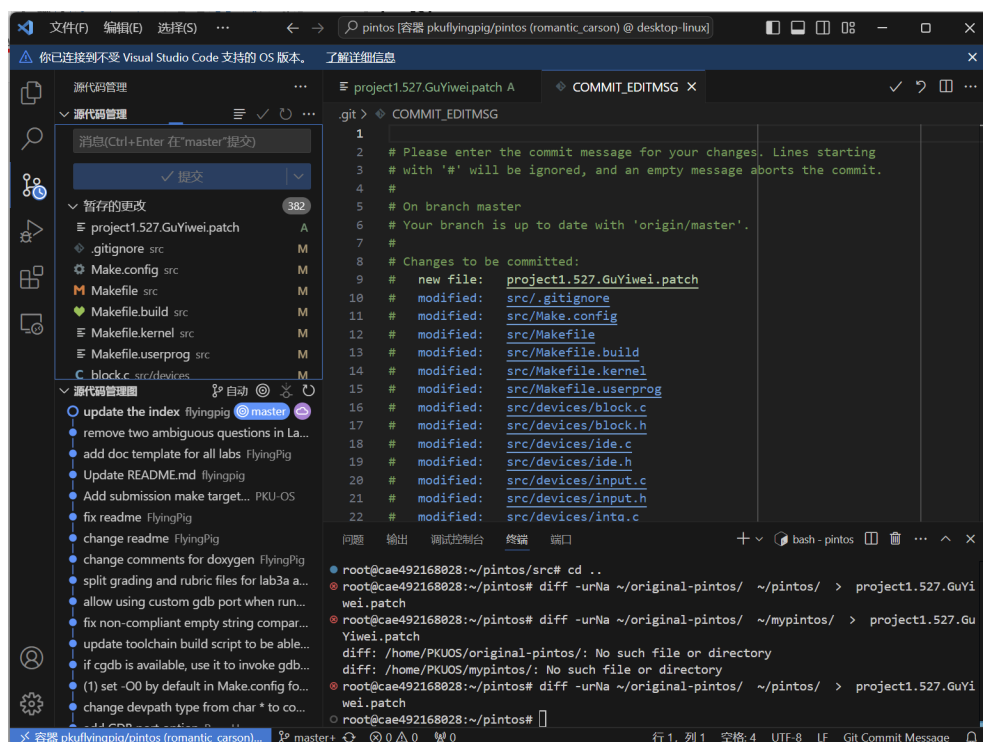


图 4: make patch after finishing project1

5 实验结果总结

在这次实验中, 我对 `timer_sleep()` 函数进行了重新设计, 使其能够支持线程按设定的时间进行休眠, 并在休眠期满后被重新激活。此外, 我还引入了基于优先级的线程调度机制, 确保线程能够依据其优先级获得相应的调度。

通过这些实践活动，我不仅加深了对操作系统线程调度机制的理解，还对操作系统的整体实现有了更加深刻的洞察。

6 附录（源代码）

由于代码篇幅较长，我将修改好的代码放在了 GitHub 仓库；

可以访问：https://github.com/SoftGhostGU/OS/tree/main/OS_project_1。