操作系统实验分享

从测试代码出发——逐步实现优先级调度
使用 gdb 调试 pintos

李鹏达

2023 年 12 月 11 日

目录

优先级调度的实现

基本：按优先级插入线程 I

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。

基本：按优先级插入线程 II

按顺序插入

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。

定义比较函数

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。

基本：按优先级插入线程 III

按顺序插入

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。

创建时按优先级插入

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。

修改优先级时重新调度

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。

基本：按优先级插入线程 IV

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

测试 priority-donate-one I

分析 priority-donate-one 测试

测试 priority-donate-one II

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**：

测试 priority-donate-one III

```
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**：

测试 priority-donate-one IV

```
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**：

测试 priority-donate-one V

```
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.");
```

测试 priority-donate-one VI

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

测试分析

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。

测试 priority-donate-one VII

修改 lock_acquire() 函数

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 lock_release() 函数

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。

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。

测试 priority-donate-one VIII

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。

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。

测试 priority-donate-one IX

这样，我们就可以通过测试 priority-donate-one 了。

测试 priority-donate-multiple 和 priority-donate-multiple2 I

这两个测试比较类似。这里以 priority-donate-multiple2 为例

测试 priority-donate-multiple 和 priority-donate-multiple2 II

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**：

测试 priority-donate-multiple 和 priority-donate-multiple2 III

```
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**：

测试 priority-donate-multiple 和 priority-donate-multiple2 IV

```
 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 ());
```

测试 priority-donate-multiple 和 priority-donate-multiple2 V

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。

测试分析

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。

测试 priority-donate-multiple 和 priority-donate-multiple2 VI

增加 locks 字段

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。

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

测试 priority-donate-multiple 和 priority-donate-multiple2 VII

修改锁结构体

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。

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`。

测试 priority-donate-multiple 和 priority-donate-multiple2 VIII

实现比较函数

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。

修改 lock_acquire() 函数

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。

测试 priority-donate-multiple 和 priority-donate-multiple2 IX

修改 lock_release() 函数

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）。

测试 priority-donate-multiple 和 priority-donate-multiple2 X

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

测试 priority-donate-nest I

接下来，我们分析 priority-donate-nest 测试

测试 priority-donate-nest II

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**：

测试 priority-donate-nest III

```
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 ());
```

测试 priority-donate-nest IV

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**:

测试 priority-donate-nest V

```
 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
```

测试 priority-donate-nest VI

```
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  }
```

测试 priority-donate-nest VII

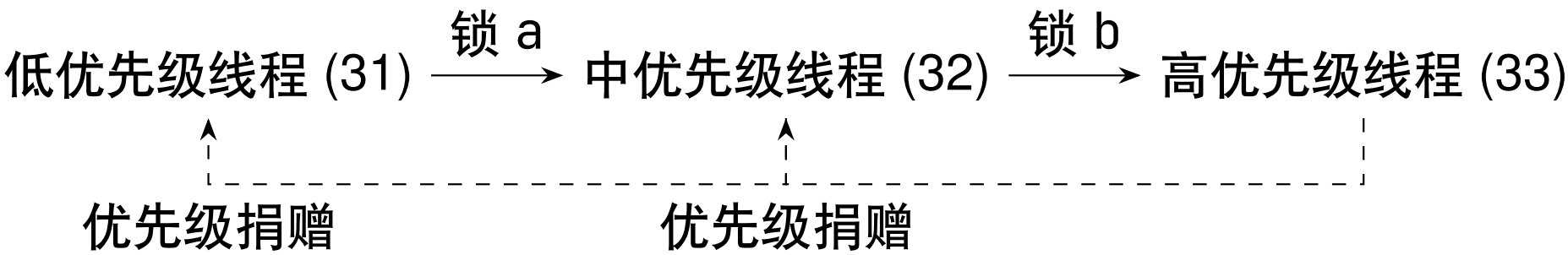
线程 high 获取锁 b，执行完毕后释放锁。

测试 priority-donate-nest VIII

测试分析

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：线程关系图

测试 priority-donate-nest IX

测试分析

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。

测试 priority-donate-nest X

增加 waiting 字段

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。

priority_donate() 函数

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。

测试 priority-donate-nest XI

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。

priority-donate-sema 测试 I

接下来，我们分析 priority-donate-sema 测试

priority-donate-sema 测试 II

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**：

priority-donate-sema 测试 III

```
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**:

priority-donate-sema 测试 IV

```
 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
```

priority-donate-sema 测试 V

```
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。

priority-donate-sema 测试 VI

测试分析

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。

priority-donate-sema 测试 VII

修改 sema_up() 函数

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。

priority-donate-sema 测试 VIII

这样，我们就可以通过测试 priority-donate-sema 了。

测试 priority-donate-lower I

接下来，我们分析 priority-donate-lower 测试

测试 priority-donate-lower II

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：**

测试 priority-donate-lower III

```
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：**

测试 priority-donate-lower IV

```
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());
```

测试 priority-donate-lower V

测试分析

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。

测试 priority-donate-lower VI

修改 thread_set_priority() 函数

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。

测试 priority-donate-lower VII

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。

测试 priority-condvar I

接下来，我们分析 priority-condvar 测试

测试 priority-condvar II

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);
```

测试 priority-condvar III

```
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**:

测试 priority-condvar IV

```
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 }
```

测试 priority-condvar V

测试分析

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。

测试 priority-condvar VI

修改 cond_signal() 函数

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。

信号量比较函数

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。

测试 priority-condvar VII

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。

测试 priority-condvar VIII

这样，我们就通过了优先级调度的所有测试。

使用 gdb 调试 pintos

gdb 命令行 I

- **运行测试文件**

```
1 $ pintos --gdb -- run alarm-multiple
```

- **启动 gdb**:

```
1 $ pintos-gdb kernel.o
```

- **设置 gdb 参数**:

```
1 (gdb) set architecture i386:x86-64
2 (gdb) target remote localhost:1234
```

- **调试**

VS Code gdb 调试 I

- 安装 C/C++ 插件
- 配置 launch.json:

```
{
  "configurations": [
    {
      "name": "gdb调试pintos",
      "type": "cppdbg",
      "request": "launch",
      "program": "${workspaceFolder}/src/threads/build/
          kernel.o",
      "MIMode": "gdb",
      "miDebuggerServerAddress": "localhost:1234",
      "cwd": "${workspaceRoot}",
      "setupCommands": [
          {
              "description": "set architecture i386:x86
                  -64",
```

VS Code gdb 调试 II

```
                "text": "set architecture i386:x86-64",
                "ignoreFailures": true
            }
        ],
    }
  ]
}
```

- **运行测试文件**

```
1 $ pintos --gdb -- run alarm-multiple
```

- **调试**

谢谢