实验报告：Pintos Priority

课程名称：操作系统	年级：2023 级本科	上机实践成绩：
指导教师：张民	姓名：张梓卫	
上机实践名称：Pintos Priority	学号：10235101526	上机实践日期：2024/10/28
上机实践编号：（3）	组号：	上机实践时间：2 学时

目录

一 实验目的

这一部分要求在 Pintos 中实现优先级调度。
本次实验作出修改的代码如下所示：
同时上传到了 Github 之上，仓库地址为：https://github.com/Shichien/ECNU-23-SEI-Homework
请在上传的 PDF 文件中直接点击粉色链接即可。

二 内容与设计思想

在本次实验中，我们的主要任务是实现 Pintos 系统中的优先级调度机制。这意味着每个线程会被赋予一个优先级，系统会根据优先级选择合适的线程执行。优先级调度机制的引入可以有效地提升系统的响应速度，使得高优先级的线程能够优先运行，提高系统的实时性。

实现优先级调度主要涉及以下几个方面的设计：

1 线程优先级的存储与初始化

在线程的数据结构 `struct thread` 中含有一个成员变量：Priority，用于表示线程的优先级。在线程创建时，需要确保初始化该优先级后能够正确地参与调度。

2 基于优先级的调度函数实现

在调度函数 `schedule()` 中加入优先级调度逻辑，以确保系统始终选择当前就绪队列中优先级最高的线程来执行。故我实现了新的排序函数 `priority_less_func`，用于比较两个线程的优先级，并对线程列表进行排序，以保证最高优先级的线程在队列的头部。

三 使用环境

使用 Docker v27.1.1 进行 Pintos 的安装实验，基于 Windows 11 操作系统使用 WSL2。
实验报告使用 LaTeX 进行撰写，使用 VSCode + Vim 编辑器进行文本编辑。

四 实验过程与分析

1 进入实验背景

使用命令：

```
pintos -- -q run alarm-priority
```

Init

输出内容如下所示：

```
Boot complete.
Executing 'alarm-priority':
(alarm-priority) begin
(alarm-priority) Thread priority 23 woke up.
(alarm-priority) Thread priority 22 woke up.
(alarm-priority) Thread priority 21 woke up.
(alarm-priority) Thread priority 30 woke up.
(alarm-priority) Thread priority 29 woke up.
(alarm-priority) Thread priority 28 woke up.
(alarm-priority) Thread priority 27 woke up.
(alarm-priority) Thread priority 26 woke up.
(alarm-priority) Thread priority 25 woke up.
(alarm-priority) Thread priority 24 woke up.
(alarm-priority) end
Execution of 'alarm-priority' complete.
Timer: 523 ticks
Thread: 0 idle ticks, 523 kernel ticks, 0 user ticks
Console: 839 characters output
Keyboard: 0 keys pressed
Powering off...
```

图 1: Pintos run alarm-priority

我们接下来的目标是修改进程的优先级，使得 alarm-Priority 进程的优先级得到正确的显示：

2 搜索文件内容

查阅官方文档，可以知道 Pintos 的文件结构如下：

- `threads/`
  基础内核的源代码，你将从项目 1 开始修改它。
- `userprog/`
  用户程序加载器的源代码，你将从项目 2 开始修改它。

- `vm/`
  一个几乎为空的目录。你将在项目 3 中实现虚拟内存。

- `filesys/`
  基本文件系统的源代码。你将在项目 2 中使用该文件系统，但直到项目 4 才需要修改它。

- `devices/`
  I/O 设备接口的源代码：键盘、计时器、磁盘等。你将在项目 1 中修改计时器的实现。除此之外，你不需要更改此代码。

- `lib/`
  一个标准 C 库子集的实现。此目录中的代码会被编译进 Pintos 内核，并且从项目 2 开始，也会编译到在其上运行的用户程序中。在内核代码和用户程序中，均可以使用 `#include <...>` 语法包含此目录中的头文件。你几乎不需要修改此代码。

- `lib/kernel/`
  仅包含在 Pintos 内核中的 C 库部分。它还包括一些数据类型的实现，这些数据类型可以在内核代码中使用：位图、双向链表和哈希表。在内核中，可以使用 `#include <...>` 语法包含此目录中的头文件。

- `lib/user/`
  仅包含在 Pintos 用户程序中的 C 库部分。在用户程序中，可以使用 `#include <...>` 语法包含此目录中的头文件。

- `tests/`
  各项目的测试代码。如果有助于测试你的提交内容，你可以修改此代码，但我们在运行测试之前会将其替换为原始版本。

3 Analysis

程序的入口为 init.c，而我们需要注意的是 threads/thread.c 文件，这其中有一个调度器：schedule() 函数，它会选择就绪队列中优先级最高的线程来执行。

观察这部分的代码：

```
1  static void schedule (void) {
2    struct thread *cur = running_thread ();
3    struct thread *next = next_thread_to_run (); // 指向下一个线程的指针
4    struct thread *prev = NULL;
5
6    ASSERT (intr_get_level () == INTR_OFF); // 测试括号内的表达式是否为真
7    ASSERT (cur->status != THREAD_RUNNING);
8    ASSERT (is_thread (next));
9
10   if (cur != next)
11     prev = switch_threads (cur, next); // 如果没有到达末尾，那么交换下一个线程到当前线程来执行
12   thread_schedule_tail (prev);
13 }
```

schedule()

往内部看，查看更细节的 `next_thread_to_run()` 函数，根据课程 PPT 中的内容，目前的线程队列是通过 FIFO 实现的，所以很可能我们需要解决问题的入口就在这里。

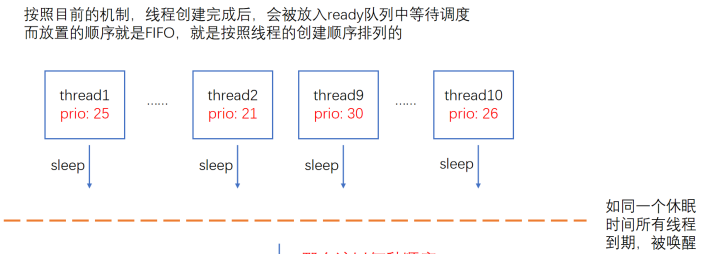

图 2: schedule()

```c
// 选择并返回下一个要调度的线程。应该从运行队列返回线程，除非运行队列是空的。
// （如果正在运行的线程可以继续运行，那么它将在运行队列中。）如果运行队列为空，则返回 idle 线程。
static struct thread *next_thread_to_run (void) {
    if (list_empty (&ready_list))
        return idle_thread;
    else
        return list_entry (list_pop_front (&ready_list), struct thread, elem);
}
```

next_thread_to_run()

在这里面，我们很容易能够看出，`list_entry` 函数采取的是提取队列首部的元素，而我要做的是按照 Priority 排序。

我需要修改为：插入方式是按照优先级插入的；选择方式是优先选择队头元素，机制实现优先级调度。

所以接下来，需要查看一下是哪些部分的函数使得 Ready 序列被添加了线程。

在显而易见的函数：`init_thread()` 中，可以看到 `list_push_back()` 函数被使用。使用 CLion 中的全局搜索：Ctrl + Shift + F，搜索这个函数名字，可以看到有以下三个函数对这个函数进行了调用。

- `thread_unblock()`
- `thread_yield()`
- `init_thread()`

4 自行添加排序逻辑

内核中一定是有相关的队列逻辑实现的，于是我到 kernel/list.h 中查看相关的函数定义，可以看到如下所示：

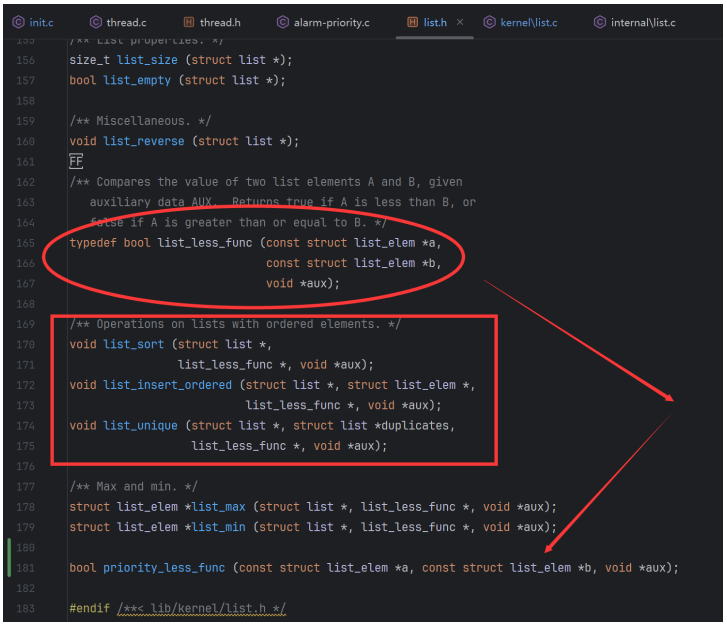

图 3: list 函数集合

已经有函数实现了按照某些特定顺序插入，即 list_insert_ordered() 函数，它需要传递一个自己编写的函数，以实现多种可能的排序方式。

list_insert_ordered 函数会在链表中找到合适的位置，以保持链表的有序性。具体操作步骤如下：

- 使用函数指针 less 指向的比较函数，比较链表中已有元素和新插入元素 elem 的大小关系。
- 根据 less 函数的返回值，将 elem 插入到链表的正确位置，从而保持链表的有序性。

故我自己添加了一个函数，以实现优先级比较：

按照 C 语言的编程规范，在 kernel/list.H 中添加一个函数声明：

```
bool priority_less_func (const struct list_elem *a, const struct list_elem *b, void *aux);
```

在 kernel/list.c 中添加函数实现：

```
/** Returns true if A is less than B, false otherwise. */
bool priority_less_func(const struct list_elem *a, const struct list_elem *b, void *aux) {
  struct thread *thread_a = list_entry(a, struct thread, elem);
  struct thread *thread_b = list_entry(b, struct thread, elem);
  return thread_a->priority > thread_b->priority;
}
```

5 修改实际代码的排序逻辑

那么，在哪里使用了 list_push_back() 函数呢？其实思路明晰之后，我们只需要将所有用到 list_push_back() 函数的地方，都替换为 list_insert_ordered() 函数即可。

根据刚刚的分析，我们只需修改那三个函数的调用即可。

```
231 void
232 thread_unblock (struct thread *t)
233 {
234   enum intr_level old_level;
235
236   ASSERT (is_thread (t));
237
238   old_level = intr_disable ();
239   ASSERT (t->status == THREAD_BLOCKED);
240   // list_push_back (&ready_list, &t->elem);
241   list_insert_ordered(&ready_list, &t->elem, (list_less_func *) &priority_less_func, NULL);
242   t->status = THREAD_READY;
243   intr_set_level (old_level);
244 }
```

图 4: list_insert_ordered()

其余的两个也如此操作：

```
if (cur != idle_thread)
// list_push_back (&ready_list, &cur->elem);
list_insert_ordered(&ready_list, &cur->elem, (list_less_func *) &priority_less_func, NULL);

old_level = intr_disable ();
// list_push_back (&all_list, &t->allelem);
list_insert_ordered(&all_list, &t->allelem, (list_less_func *) &priority_less_func, NULL);
intr_set_level (old_level);
```

至此，所有的核心代码已经修改完成。

6 Make Check

Next，我们应该检查一下这样的修改是否符合了我们的逻辑规范，是否成功实现了目标功能。

但是除此之外的修改，仍未成功能够实现相关的效果，于是我选择重新查看 thread.c 文件，查看是否有哪些函数是我遗漏的。

可以发现，文件中有两个和优先级相关的函数，即 thread_set_priority() 和 thread_get_priority() 函数。

突然想起来，在课程 PPT 中出现了相关的内容：

```
// 优先级测试程序——测试不同的优先级程序在休眠相同时间后，被同时唤醒能否按照优先级顺序进行调度
void test_alarm_priority (void)
{
  int i;
  ASSERT (!thread_mlfqs); /* 这个测试不能在 MLFQS 中 */

  wake_time = timer_ticks () + 5 * TIMER_FREQ;      // 唤醒的时间统一
  sema_init (&wait_sema, 0);

  for (i = 0; i < 10; i++)      // 编写10个线程
    {
      int priority = PRI_DEFAULT - (i + 5) % 10 - 1; // 优先级是25~21，30，29~26
      char name[16];      // 线程名字是：priority 30类似
      snprintf (name, sizeof name, "priority %d", priority);    // 赋值线程名字
      thread_create (name, priority, alarm_priority_thread, NULL);    // 创建线程
    }

  thread_set_priority (PRI_MIN);// 设置当前线程的优先级——0，那么优先级是所有线程中最低的，将会被抢占

  // 主要是为了防止主线程在其他线程未执行完时，就退出
  for (i = 0; i < 10; i++)
    sema_down (&wait_sema);     // 主线程会出现被调度，后执行此步，又被阻塞
}
```

图 5: 优先级

故查看原来的代码，确实这里是调用了 thread_set_priority() 函数的。

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

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

  wake_time = timer_ticks () + 5 * TIMER_FREQ;
  sema_init (&wait_sema, 0);

  for (i = 0; i < 10; i++)
    {
      int priority = PRI_DEFAULT - (i + 5) % 10 - 1;
      char name[16];
      snprintf (name, sizeof name, "priority %d", priority);
      thread_create (name, priority, alarm_priority_thread, NULL);
    }

  thread_set_priority (PRI_MIN);

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

图 6: 优先级

而代码：`int priority = PRI_DEFAULT - (i + 5) % 10 - 1;` 会使得优先级为 25-21, 30, 29-26。

所以，在设置当前线程的优先级时，应该需要对优先级做一些判断。

```
/** Sets the current thread's priority to NEW_PRIORITY. */
void thread_set_priority (int new_priority) {
  int old_priority = thread_current ()->priority;
  thread_current ()->priority = new_priority;
  if (old_priority > new_priority)
    thread_yield();
    // 如果是之前线程的优先级比较高，则当前线程不再是就绪队列中优先级最高的线程.
    // 当前线程的 CPU 使用权交还给调度器，从而使其他优先级更高或相等的线程有机会运行。
}
```

修改此处的代码后，再跑一次试试。

```
root@d03caec50255:~/pintos/src/threads/build# pintos -- -q run alarm-priority
qemu-system-i386 -device isa-debug-exit -drive format=raw,media=disk,index=0,
file=/tmp/ZsPPpO3hHQ.dsk -m 4 -net none -nographic -monitor null
Pintos hda1
Loading..........
Kernel command line: -q run alarm-priority
Pintos booting with 3,968 kB RAM...
367 pages available in kernel pool.
367 pages available in user pool.
Calibrating timer...  157,081,600 loops/s.
Boot complete.
Executing 'alarm-priority':
(alarm-priority) begin
(alarm-priority) Thread priority 30 woke up.
(alarm-priority) Thread priority 29 woke up.
(alarm-priority) Thread priority 28 woke up.
(alarm-priority) Thread priority 27 woke up.
(alarm-priority) Thread priority 26 woke up.
(alarm-priority) Thread priority 25 woke up.
(alarm-priority) Thread priority 24 woke up.
(alarm-priority) Thread priority 23 woke up.
(alarm-priority) Thread priority 22 woke up.
(alarm-priority) Thread priority 21 woke up.
(alarm-priority) end
Execution of 'alarm-priority' complete.
Timer: 522 ticks
Thread: 0 idle ticks, 522 kernel ticks, 0 user ticks
Console: 839 characters output
Keyboard: 0 keys pressed
```

图 7: make check

成功通过 Priority 排序的测试。

7 实验结果

使用 `make check` 命令来检查，alarm-priority 进程的优先级已经按照优先级排序，检查点通过。

```
C:\Windows\system32\cmd.e
  (mlfqs-block) Block thread acquiring lock...
  (mlfqs-block) Main thread spinning for 5 seconds...
  (mlfqs-block) Main thread releasing lock.
- (mlfqs-block) ...got it.
  (mlfqs-block) Block thread should have already acquired lock.
  (mlfqs-block) end
pass tests/threads/alarm-single
pass tests/threads/hello-world
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
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
pass 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
18 of 28 tests failed.
../../tests/Make.tests:26: recipe for target 'check' failed
make: *** [check] Error 1
root@d03caec50255:~/pintos/src/threads/build#
```

图 8: make check

五 总结

在实现优先级调度的过程中，我首先意识到操作系统的任务调度不仅仅是简单的顺序处理，而是需要根据线程的优先级来动态调整。这就涉及到对线程的数据结构的修改和初始化过程的设计。在实际开发过程中，我发现简单的功能要求往往需要触及代码的多个模块和层次，比如从线程的创建到调度，再到具体的数据结构操作，这种模块之间的关联性给了我很多启发。

在实现优先级调度时，函数指针 `list_insert_ordered` 的使用让我体会到 C 语言灵活性与复杂性并存的特性。通过设计 `priority_less_func` 比较函数，我们可以实现基于优先级的队列排序逻辑，从而在操作系统中达到按照优先级执行的效

果。使用函数指针不仅让代码更加模块化，也极大地提高了代码的复用性，体会到这种编程设计思想的优越性。

六 附录

参考资料：

- https://pkuflyingpig.gitbook.io/pintos/project-description/lab1-threads/faq