# 实验报告：修改 alarm-priority

**课程名称**：操作系统实践　　**年级**：大二　　**上机实践成绩**：

**指导教师**：张民　　**姓名**：邱吉尔

**学号**：10235101533　　**上机实践日期**：**2024/11/18**

---

## 一、目的

实现 thread 的 sleep 功能，使得 thread 在规定时间唤醒，不是忙等

## 二、内容与设计思想

不再使用 thread_yield 函数，而是使用 thread_sleep()，将当前线程设置为 sleep 状态，并设置好线程应该醒来的时间，让当前线程放弃 cpu

## 三、使用环境

Docker , VScode

## 四、实验过程

### 1. 展示忙等

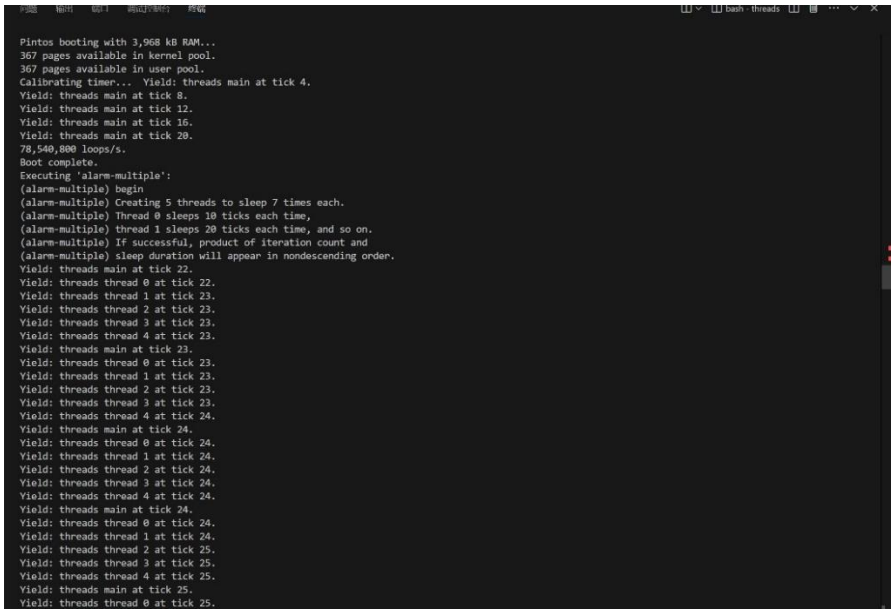
在 thread_yield()中增加 print 语句，编译后运行，查看输出结果

```
void
thread_yield (void)
{
  int64_t cur_ticks=timer_ticks();
  struct thread *cur = thread_current ();
  enum intr_level old_level;

  ASSERT (!intr_context ());

  old_level = intr_disable ();
  if (cur != idle_thread)
    list_insert_ordered (&ready_list, &cur->elem,prio_cmp_fun,0);
    printf("Yield: threads %s at tick %lld.\n",cur->name,cur_ticks);
  cur->status = THREAD_READY;
  schedule ();
  intr_set_level (old_level);
}
```

输出结果：




2. yield 和 sleep 的比较

   ①sleep 是线程主动放弃 CPU，将自己从 running 态变成非 ready 态，并设定一个倒计时时间，倒计时时间到了，就重新唤醒等待执行

   ②yield 是线程主动放弃 CPU，将自己从 running 态变成 ready 态，重新加入到 ready 队列中，让其他线程去执行（在优先级调度机制中，只是让优先级比自己高的线程去执行），主要是为了防止饥饿

3. 不再使用 thread_yield 函数，而改用 thread_sleep，当前线程设置为 sleep 状态，并设置好线程应该醒来的时间，让当前线程主动放弃 cpu

```
void
timer_sleep (int64_t ticks)
{
  // int64_t start = timer_ticks ();

  // ASSERT (intr_get_level () == INTR_ON);
  // while (timer_elapsed (start) < ticks)
  //   thread_yield ();//主动放弃CPU，将自己从running态变成ready态，

  thread_sleep(ticks);
}
```

thread_sleep 具体实现：

```
void thread_sleep (int64_t ticks){
  if(ticks<=0)  return;
  struct thread *cur=thread_current();

  enum intr_level old_level=intr_disable();
  if(cur!=idle_thread){
    cur->status=THREAD_SLEEP;
    cur->wake_time=timer_ticks()+ticks;
    schedule();
  }
  intr_set_level(old_level);

}
```

4.在 timer_interrupt 中设置函数 check_and_wakeup_sleep_thread()以唤醒线程

```
static void
timer_interrupt (struct intr_frame *args UNUSED)
{
  ticks++;
  thread_tick ();
  check_and_wakeup_sleep_thread();
}
```

check_and_wakeup_sleep_thread()具体实现：

```
void check_and_wakeup_sleep_thread(void){
  struct list_elem *e=list_begin(&all_list);
  int64_t cur_ticks = timer_ticks();

  while(e!=list_end(&all_list)){
    struct thread *t = list_entry(e,struct thread,allelem);
    enum intr_level old_level=intr_disable();
    if(t->status==THREAD_SLEEP && cur_ticks >= t->wake_time){
      t->status=THREAD_READY;
      list_insert_ordered(&ready_list,&t->elem,prio_cmp_fun,NULL);
      msg("Wake up thread %s at tick %lld.\n",t->name,cur_ticks);
    }
    e=list_next(e);
    intr_set_level(old_level);
  }

}
```

5.**实现休眠**，运行指令运行指令 pintos -v -- -q run alarm-multiple，发现线程在规定时间唤醒，不是忙等：

```
Kernel command line: -q run alarm-multiple
Pintos booting with 3,968 kB RAM...
367 pages available in kernel pool.
367 pages available in user pool.
Calibrating timer...  Yield: threads main at tick 4.
Yield: threads main at tick 8.
Yield: threads main at tick 12.
Yield: threads main at tick 16.
Yield: threads main at tick 20.
78,540,800 loops/s.
Boot complete.
Executing 'alarm-multiple':
(alarm-multiple) begin
(alarm-multiple) Creating 5 threads to sleep 7 times each.
(alarm-multiple) Thread 0 sleeps 10 ticks each time,
(alarm-multiple) thread 1 sleeps 20 ticks each time, and so on.
(alarm-multiple) If successful, product of iteration count and
(alarm-multiple) sleep duration will appear in nondescending order.
(alarm-multiple) Wake up thread thread 0 at tick 132.

(alarm-multiple) Wake up thread thread 0 at tick 142.

(alarm-multiple) Wake up thread thread 1 at tick 142.

(alarm-multiple) Wake up thread thread 0 at tick 152.

(alarm-multiple) Wake up thread thread 2 at tick 152.

(alarm-multiple) Wake up thread thread 0 at tick 162.

(alarm-multiple) Wake up thread thread 1 at tick 162.

(alarm-multiple) Wake up thread thread 3 at tick 162.

(alarm-multiple) Wake up thread thread 0 at tick 172.

(alarm-multiple) Wake up thread thread 4 at tick 172.

(alarm-multiple) Wake up thread thread 0 at tick 182.

(alarm-multiple) Wake up thread thread 1 at tick 182.

(alarm-multiple) Wake up thread thread 2 at tick 182.
```

6. 实现苏醒后抢占：sleep 的进程醒来时，如果当前 running 的进程优先级比它低，醒来的进程抢占执行

① 首先由 priority 实验可知，要将就绪队列插入方式改为 list_insert_ordered,如下：

```
void
list_insert_ordered (struct list *list, struct list_elem *elem,
                     list_less_func *less, void *aux)
{
  struct list_elem *e;

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

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

② 其次，为了更好的管理休眠线程，可以添加一个全局的睡眠线程队列，辅助在定时器中处理线程睡眠，代码如下：

```
// 声明一个全局的睡眠线程队列
static struct list sleeping_threads;

// // 初始化睡眠线程队列
void sleeping_threads_init(void) {
    list_init(&sleeping_threads);
}
```

③ 由此，准备工作已完成，接下来是 test_alarm_multiple 函数，相较于原来进行了大改：

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

  /* This test does not work with the MLFQS. */
  ASSERT (!thread_mlfqs);//这个测试不能在MLFQS中

  wake_time = timer_ticks () + 5 * TIMER_FREQ;
  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
      snprintf (name, sizeof name, "priority %d", priority);//赋值线程名字
      int64_t sleep_time = (i + 1) * 10;
      thread_create (name, priority, alarm_multiple_thread,  (void *) sleep_time);//创建线程

    }

  thread_set_priority (1);//设置当前线程优先级1，那么优先级是所有线程中最高的，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
  // test_sleep (5, 7);
}
```

首先创建线程的方法不再是原先的利用 test_sleep 方法，而是利用了上一次实验中所用到的 priority 线程的创建方法，将 alarm_priority_thread 改为 alarm_multiple_thread，添加了抢占式调度，如④所示，并且在线程创建之初不仅设置了 priority，也设置了其休眠时间，并将当前线程的优先级设置为最高，这样在后期调度时便会被抢占

④ alarm-multiple 的相关代码如下：

```
static void
alarm_multiple_thread (void *aux)
{
  int64_t sleep_time=(int64_t) aux;//获取传入的休眠时间
  int64_t start_time = timer_ticks ();//记录当前时间

  msg("Thread %s is going to sleep for %lld ticks.", thread_name(), sleep_time);
  timer_sleep (sleep_time);//每线程休眠
  //休眠结束后唤醒
  msg ("Thread %s woke up.", thread_name ());//唤醒了
 // 获取当前正在执行的线程
    struct thread *current_thread = thread_current();

    // 如果当前线程的优先级比正在执行的线程低，进行抢占
    if (thread_get_priority() > current_thread->priority) {
        msg("Thread %s has higher priority than %s and is preempting.",
            thread_name(), current_thread->name);
        thread_sleep(timer_ticks());  // 使用 thread_sleep 让当前线程让出 CPU，抢占式调度
    } else {
        msg("Thread %s does not preempt %s (priority comparison).",
            thread_name(), current_thread->name);
    }
  sema_up (&wait_sema);
}
```

用现在的时间和已经休眠时间两个参数用于判断线程是否休眠完成，并添加了优先级参数，用于判断是否需要抢占

⑤timer_sleep 在上次实验改动后就是直接调用了 thread_sleep 函数，因此我们直

接来看 thread_sleep 函数的改动：

```
610 void thread_sleep(int64_t ticks) {
611     if (ticks <= 0) return;
612 
613     struct thread *cur = thread_current();
614 
615     enum intr_level old_level = intr_disable();
616     if (cur != idle_thread) {
617         cur->status = THREAD_SLEEP;
618         cur->wake_time = timer_ticks() + ticks;
619 
620         // 根据唤醒时间来排序线程，如果唤醒时间相同，按优先级排序
621         list_insert_ordered(&sleeping_threads, &cur->elem, thread_wake_time_less_priority, NULL);
622 
623         schedule();  // 调度其他线程
624     }
625     intr_set_level(old_level);
626 }
627 
```

主要是 list_insert_ordered 函数中传入的参数的改动，将当前线程插入 sleeping_threads 队列中。该队列是一个按照线程唤醒时间排序的链表。

- &sleeping_threads：这是全局的睡眠队列，存储所有正在休眠的线程。
- &cur->elem：当前线程的队列元素。
- thread_wake_time_less_priority：是一个比较函数，用于根据线程的 wake_time 对睡眠队列进行排序。这样队列中的线程按照唤醒时间升序排列。而若是线程醒来后优先级比正在运行的线程优先级更高而未被调度，则会进入 ready 态，并且根据优先级进行排序，优先级低的在前

⑥最后一步就是完成 thread_wake_time_less_priority 比较函数的编写：

```
481 bool thread_wake_time_less_priority(const struct list_elem *a, const struct list_elem *b, void *aux) {
482     struct thread *t_a = list_entry(a, struct thread, elem);
483     struct thread *t_b = list_entry(b, struct thread, elem);
484 
485     // 首先比较唤醒时间
486     if (t_a->wake_time < t_b->wake_time) {
487         return true;
488     } else if (t_a->wake_time > t_b->wake_time) {
489         return false;
490     }
491 
492     // 如果唤醒时间相同，比较优先级
493     return t_a->priority > t_b->priority;  // 优先级低的排在前面，降序排序
494 }
495 
```

先判断唤醒时间，唤醒早的先调度，否则比较优先级，优先级低的先调度

## 五、总结

1. 发现问题：在 thread_yield 调度下该休眠的线程因优先级最高而一直占着 cpu，即忙等待，此次实验目标为在规定时间唤醒线程而不让其忙等
2. 不再使用 thread_yield，而是 thread_sleep，并在线程状态结构体中新增 sleep 状态，使之在后面调度中可告诉 cpu 处于休眠状态
3. 在 thread_sleep 中设置好休眠时间后，设置函数 check_and_wakeup_sleep_thread() 以唤醒线程
4. 由此可以避免忙等待问题
5. 实现苏醒后抢占，虽然后期有一部分代码借助了 ai 才得以完成，但在编写过程中对于线程的调度有了更深刻的理解，同时也对一个相对较大的项目的整体结构有了个大概的认识