课程名称：操作系统实践	年级：2023级	上机实践成绩：
指导教师：张民	姓名：李彤	
上机实践名称：忙等待实验	学号：10235101500	上机实践日期：2024.11.04
上机实践编号：	组号：	上机实践时间：

展示忙等待

原因分析

当前线程主动将 CPU 执行权释放，然后经过一段时间（ticks）后，系统将该线程唤醒，将其重新加入到就绪队列中等待调度。但是由于调度队列是优先级调度，导致在一定时间内，该线程被反复加入就绪队列（ready）以及从队列中取出执行（running），这样的过程非常占用 CPU 并且唤醒顺序非常混乱。所以线程状态从running 变成 ready 会造成 CPU 忙等待。

观察发现的一些问题

在运行最开始的代码时，可以发现，所有`time_ticks()`的返回值都是负数，显然不符合逻辑，所以我在`yield()`中的调试语句中使用强制类型转换将其值转化成非负数，并且将后续`check_and_wakeup_sleep_thread`中的`time_ticks()`返回值也进行强制转化。

```
Pintos hda1
Loading...........
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:thread main at tick -4611106404000858108.
Yield:thread main at tick -4611106404000858104.
Yield:thread main at tick -4611106404000858100.
Yield:thread main at tick -4611106404000858096.
Yield:thread main at tick -4611106404000858092.
130,867,200 loops/s.
Boot complete.
Executing 'alarm-multiple':
(alarm-multiple)Yield:thread main at tick -4611106404000858088.
 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.
Yield:thread main at tick -4611106404000858088.
Yield:thread thread 0 at tick -4611106404000858088.
Yield:thread thread 1 at tick -4611106404000858087.
Yield:thread thread 2 at tick -4611106404000858087.
Yield:thread thread 3 at tick -4611106404000858087.
Yield:thread thread 4 at tick -4611106404000858087.
Yield:thread main at tick -4611106404000858087.
Yield:thread thread 0 at tick -4611106404000858087.
Yield:thread thread 1 at tick -4611106404000858087.
Yield:thread thread 2 at tick -4611106404000858087.
```

修改代码

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

  ASSERT(!intr_context());

  old_level = intr_disable();
  if (cur != idle_thread)
    // list_push_back (&ready_list, &cur->elem);
  list_insert_ordered(&ready_list, &cur->elem, prio_cmp_func, NULL);
  printf("Yield:thread %s at tick %lld.\n", cur->name, (uint64_t)timer_ticks());
  cur->status = THREAD_READY;
  schedule();
  intr_set_level(old_level);
}
```

这是修正后的结果

```
Pintos hda1
Loading...........
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:thread main at tick 4.
Yield:thread main at tick 8.
Yield:thread main at tick 12.
Yield:thread main at tick 16.
Yield:thread main at tick 20.
Yield:thread main at tick 24.
116,121,600 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.
Yield:thread main at tick 27.
Yield:thread thread 0 at tick 27.
Yield:thread thread 1 at tick 27.
Yield:thread thread 2 at tick 27.
Yield:thread thread 3 at tick 27.
Yield:thread thread 4 at tick 27.
Yield:thread main at tick 27.
Yield:thread thread 0 at tick 27.
Yield:thread thread 1 at tick 27.
Yield:thread thread 2 at tick 27.
```

通过观察忙等待的输出可以发现，线程的调度并没有按照优先级实现。 ~~这里先放一下，等到最后和正确代码比较。~~

```
(alarm-multiple) thread 4: duration=50, iteration=2, product=100
(alarm-multiple) thread 1: duration=20, iteration=5, product=100
```

实验步骤

1. 首先添加两个我们需要用到的变量

其中`thread_status`用来记录每一个线程的状态，在这里我们添加`THREAD_SLEEP`用来表示进程处于 **休眠状态** 。

在`thread`结构体中添加变量`wake_time`用来记录线程 **休眠结束** 的时间。

```
enum thread_status
  {
    THREAD_RUNNING,     /**< Running thread. */
    THREAD_READY,       /**< Not running but ready to run. */
    THREAD_BLOCKED,     /**< Waiting for an event to trigger. */
    THREAD_DYING,       /**< About to be destroyed. */
    THREAD_SLEEP        /**< Slepping thread,litong*/
  };
```

```
    unsigned magic;                     /**< Detects stack overflow. */
    int64_t wake_time;                  /**< Time when thread wake up. litong*/
};
```

2. 进程休眠实现

先修改timer_sleep函数的内部接口

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

  // ASSERT (intr_get_level () == INTR_ON);
  // while (timer_elapsed (start) < ticks)
  //   thread_yield ();

  thread_sleep(ticks);
}
```

接下来，我们就要实现`thread_sleep`函数了。

`thread_sleep`函数首先判断了休眠时长是否合法，如果不合法，就直接退出。然后获取当前进程，`cur != idle_thread`确保了CPU不是在空等待，接着设置进程的状态和休眠结束的时间，`schedule`函数调度进程，将进程插入 **ready队列** 而不是直接执行，也不是插入waiting队列。~~等到下次调用ready队列中的进程时，再按照~~

~~优先级重新调度~~

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

  // 禁用中断并保存当前中断级别
  enum intr_level old_level = intr_disable();

  if (cur != idle_thread)    // 确保CPU不是在空等待
  {
    cur->status = THREAD_SLEEP;        // 将当前进程状态改为休眠
    cur->wake_time = timer_ticks() + ticks;    // 设置进程休眠结束的时间
    schedule();      // 调度进程，将进程插入ready队列而不是直接执行，也不是插入waiting队列
  }

  // 恢复之前的中断级别
  intr_set_level(old_level);
}
```

3. 唤醒进程

还是先介绍一下函数`check_and_wakeup_sleep_thread()`，这个函数会遍历当前所有进程，并判断是否有进程休眠结束，如果有，那么它就会把这个进程按照优先级有序地插入ready队列等待执行，同时输出一些调试信息。

```
void check_and_wakeup_sleep_thread(){
  struct list_elem *e = list_begin(&all_list);
  int64_t cur_ticks = (uint64_t)timer_ticks();

  // 遍历所有线程列表
  while(e != list_end(&all_list)){
    struct thread *t = list_entry(e, struct thread, allelem);
    enum intr_level old_level = intr_disable();

    // 如果线程处于睡眠状态且当前时间已达到或超过唤醒时间
    if(t->status == THREAD_SLEEP && cur->ticks >= t->wake_time) {
      t->status = THREAD_READY;

      // 将线程插入到就绪队列中，按优先级排序
      list_insert_ordered(&ready_list,&t->elem, prio_cmp_func, NULL);
      printf("Wake up thread %s at tick %lld.\n", t->name, cur_ticks);  // 提示信息
    }

    // 移动到下一个进程
    e = list_next(e);
    // 恢复之前的中断级别
    intr_set_level(old_level);
  }
}
```

因此，我们可以通过把函数`check_and_wakeup_sleep_thread`加到时钟中断里面，这样每发生一次时钟中断，我们就可以排查一次进程链表，确保那些休眠结束的进程能够及时进入等待序列（不是waiting，仍指

ready)

```
/** Timer interrupt handler. */
static void
timer_interrupt (struct intr_frame *args UNUSED)
{
  ticks++;
  thread_tick ();

  // 检查当前时间是否到了线程的睡眠时间，如果是，则唤醒该线程
  check_and_wakeup_sleep_thread();
}
```

抢占式优先级调度的实现

这里我主要是在`check_and_wakeup_sleep_thread`函数内进行修改。
主要思路是，用一个thread指针，记录被唤醒线程中最高优先级线程，等到遍历唤醒结束以后，再将该线程的优先级与当前running线程的优先级进行比较，如果该优先级更大， 就发生抢占，否则不抢占。具体实现代码如下：

```
void check_and_wakeup_sleep_thread(void) {
  int64_t cur_ticks = timer_ticks();  // 获取当前滴答数
  struct thread *highest_priority_thread = NULL;  // 记录被唤醒线程中最高优先级线程

  // 遍历所有线程列表
  struct list_elem *e = list_begin(&all_list);
  while (e != list_end(&all_list)) {
    enum intr_level old_level = intr_disable();  // 在整个操作中禁用中断，减少频繁的中断开关

    struct thread *t = list_entry(e, struct thread, allelem);
    e = list_next(e);  // 提前获取下一个线程，防止后续对当前线程状态的修改影响遍历

    // 如果线程处于休眠状态且当前时间已达到或超过唤醒时间
    if (t->status == THREAD_SLEEP && cur_ticks >= t->wake_time) {
      // 禁用中断以进行安全操作
      enum intr_level old_level = intr_disable();

      t->status = THREAD_READY;  // 将线程状态设置为就绪
      list_insert_ordered(&ready_list, &t->elem, prio_cmp_func, NULL); // 插入到就绪队列

      // 更新被唤醒线程中最高优先级线程
      if (!highest_priority_thread || t->priority > highest_priority_thread->priority) {
        highest_priority_thread = t;
      }
      printf("Wake up thread %s at tick %lld.\n", t->name, cur_ticks);  // 输出日志

      // 恢复中断状态
      intr_set_level(old_level);
```

```
      }
    }

    // 抢占逻辑
    if (highest_priority_thread) {
      struct thread *current = thread_current(); // 获取当前运行线程
      // 如果唤醒的线程优先级高于当前线程，进行抢占
      if (highest_priority_thread->priority > current->priority) {
        thread_yield();
      }
    }
  }
```

一开始我的抢占函数直接用的是`thread_yield()`函数，但是发现会爆如下的错误，大概就是上下文切换的问题，在上网查询资料后才知道这里还需要确保线程不在中断上下文中，因此我们可以用`intr_yield_on_return`函数替换， 它会在中断返回时立即触发调度，切换到优先级更高的线程，保证了调度的安全。
（也可以在`thread_yield()`函数前加一个`if(!intr_context())`的判断语句，确保线程不在中断上下文中。这种方法输出除了少了一条 `Yield:thread idle is at ...` 语句其他都差不多，所以我觉得这两种方法应该都可选

```
Loading...........
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...  Kernel PANIC at ../../threads/thread.c:344 in thread_yield(): assertion `!intr_context()' failed.
Call stack: 0xc00296f8 0xc0020efb 0xc0020af0 0xc0024013 0xc0021b7f 0xc0021d9f 0xc0024031 0xc0023dc6 0xc002033e.
The `backtrace' program can make call stacks useful.
Read "Backtraces" in the "Debugging Tools" chapter
of the Pintos documentation for more information.
Timer: 1 ticks
Thread: 0 idle ticks, 1 kernel ticks, 0 user ticks
```

```
  // 抢占逻辑
  if (highest_priority_thread) {
    struct thread *current = thread_current(); // 获取当前运行线程
    // 如果唤醒的线程优先级高于当前线程，进行抢占
    if (highest_priority_thread->priority > current->priority) {
      // thread_yield();
      intr_yield_on_return();
    }
  }
}
```

但是似乎还是有点问题，running_thread竟然都是idle，cpu在空等待，令人费解。

```
(alarm-multiple) sleep duration will appear in nondescending order.
Wake up thread thread 0 at tick 134.
Yield:thread idle at tick 134.
Wake up thread thread 0 at tick 144.
Wake up thread thread 1 at tick 144.
Yield:thread idle at tick 144.
Wake up thread thread 0 at tick 154.
Wake up thread thread 2 at tick 154.
Yield:thread idle at tick 154.
Wake up thread thread 0 at tick 164.
Wake up thread thread 1 at tick 164.
Wake up thread thread 3 at tick 164.
Yield:thread idle at tick 164.
```

~~在经过7200s的调试后~~ 疑似发现问题出在那里了，我给代码加了一段调试语句：

```
printf("Ready list contents:\n");
for (e = list_begin(&ready_list); e != list_end(&ready_list); e = list_next(e)) {
struct thread *t = list_entry(e, struct thread, elem);
printf("Thread %s with priority %d\n", t->name, t->priority);
}
```

输出结果如下：

```
Ready list contents:
Ready list contents:
Ready list contents:
Ready list contents:
Ready list contents:
Ready list contents:
Wake up thread thread 3 at tick 286.
```

ready_list竟然是空的！！！！这下就能解释为什么前面cpu都在空转了——压根没有进程在等待执行，发现问题之后，就要准备解决问题了。

按照同样的方法，我也输出了一下all_list链表，似乎也有点不对劲：

```
All list contents:
Thread  with priority 0
Thread  with priority 0
Thread  with priority 0
Ready list contents:
All list contents:
Thread  with priority 0
Thread  with priority 0
Thread  with priority 0
Ready list contents:
```

and 在原本的输出日志里多加了priority的输出，发现所有进程的优先级都相同（所以抢占不了）

```
printf("Wake up thread %s at tick %lld with priority %d.\n", t->name, cur_ticks, t->priority);  // 输出日志
```

```
Wake up thread thread 3 at tick 363 with priority 31.
Yield:thread idle at tick 363.
Wake up thread thread 4 at tick 373 with priority 31.
Yield:thread idle at tick 373.
Wake up thread thread 3 at tick 403 with priority 31.
```

由此我们可以知道，如果要实现抢占式的优先级调度，那必然要先设计算法使创建的thread优先级能够发生改变。

测试代码

before

前面讲到在运行原始代码时，进程并没有按照正确的顺序被调度，这是因为忙等待导致进程被唤醒顺序非常混乱。

```
(alarm-multiple) thread 4: duration=50, iteration=2, product=100
(alarm-multiple) thread 1: duration=20, iteration=5, product=100
```

after

在修改代码之后，系统能够按照正确的优先级顺序调度进程。

```
(alarm-multiple) thread 3: duration=40, iteration=2, product=80
(alarm-multiple) thread 2: duration=30, iteration=3, product=90
(alarm-multiple) thread 1: duration=20, iteration=5, product=100
(alarm-multiple) thread 4: duration=50, iteration=2, product=100
(alarm-multiple) thread 1: duration=20, iteration=6, product=120
(alarm-multiple) thread 2: duration=30, iteration=4, product=120
```

为了确保实验结果的正确性，我还跑了多次测试，结果都符合预期。下面给出了其中两次测试的结果。

```
Pintos hda1
Loading...........
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:thread main at tick 4.
Yield:thread main at tick 8.
Yield:thread main at tick 12.
Yield:thread main at tick 16.
Yield:thread main at tick 20.
104,755,200 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.
Wake up thread thread 0 at tick 130.
Wake up thread thread 0 at tick 140.
Wake up thread thread 1 at tick 140.
Wake up thread thread 0 at tick 150.
Wake up thread thread 2 at tick 150.
Wake up thread thread 0 at tick 160.
Wake up thread thread 1 at tick 160.
Wake up thread thread 3 at tick 160.
Wake up thread thread 0 at tick 170.
Wake up thread thread 4 at tick 170.
Wake up thread thread 0 at tick 180.
Wake up thread thread 1 at tick 180.
```

```
Pintos hda1
Loading...........
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:thread main at tick 4.
Yield:thread main at tick 8.
Yield:thread main at tick 12.
Yield:thread main at tick 16.
Yield:thread main at tick 20.
Yield:thread main at tick 24.
114,073,600 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.
Yield:thread main at tick 28.
Wake up thread thread 0 at tick 137.
Wake up thread thread 0 at tick 147.
Wake up thread thread 1 at tick 147.
Wake up thread thread 0 at tick 157.
Wake up thread thread 2 at tick 157.
Wake up thread thread 0 at tick 167.
Wake up thread thread 1 at tick 167.
Wake up thread thread 3 at tick 167.
Wake up thread thread 0 at tick 177.
Wake up thread thread 4 at tick 177.
```

遗留问题

1. 输出乱序

但是，我发现在最终的输出中，总是会有一些很"碍眼"的存在，如下

```
(alarm-multiple) thread 1: duration=20, iteration=7, product=140
(alarm-multiple) thread 2: duration=30, iteration=5, product=150
(alarm-multiple) thread 4: duration=5Yield:thread main at tick 581.
0, iteration=3, product=150
(alarm-multiple) thread 3: duration=40, iteration=4, product=160
(alarm-multiple) thread 2: duration=30, iteration=6, product=180
(alarm-multiple) thread 3: duration=40, iteration=5, product=200
```

可以看出，在这一段的输出中发生了混乱，两条输出语句同时进行，这就导致了内容交错输出。
结合我之前的了解 ~~（看过一点xv6的代码）~~ ，我认为是某个语句块没有加锁，导致在输出时CPU被其他进程抢占，这就造成了内容交替输出的局面，后面还去问了助教 ~~（and热心同学的解答）~~ ，确实是多线程的问题。

这里先保留问题，课后还可以再了解一下多线程和锁的相关内容。

2. 线程优先级的修改和ready_list的插入

因为近期考试比较多，所以没花太多时间继续编写、调试关于线程优先级的修改和ready_list的插入的代码.
关于抢占式优先级调度的实现基本就是前面代码所示，最终的结果大概就是下面这样，如果能实现线程优先级的修改，应该是能够实现抢占了。

```
Wake up thread thread 2 at tick 304 with priority 31.
Wake up thread thread 3 at tick 324 with priority 31.
Wake up thread thread 4 at tick 324 with priority 31.
Wake up thread thread 2 at tick 334 with priority 31.
Wake up thread thread 3 at tick 364 with priority 31.
Wake up thread thread 4 at tick 374 with priority 31.
Wake up thread thread 3 at tick 404 with priority 31.
Wake up thread thread 4 at tick 424 with priority 31.
Wake up thread thread 4 at tick 474 with priority 31.
Wake up thread main at tick 574 with priority 31.
(alarm-multiple) thread 0: duration=10, iteration=1, product=10
(alarm-multiple) thread 0: duration=10, iteration=2, product=20
(alarm-multiple) thread 1: duration=20, iteration=1, product=20
(alarm-multiple) thread 0: duration=10, iteration=3, product=30
(alarm-multiple) thread 2: duration=30, iteration=1, product=30
(alarm-multiple) thread 0: duration=10, iteration=4, product=40
(alarm-multiple) thread 1: duration=20, iteration=2, product=40
(alarm-multiple) thread 3: duration=40, iteration=1, product=40
(alarm-multiple) thread 0: duration=10, iteration=5, product=50
(alarm-multiple) thread 4: duration=50, iteration=1, product=50
(alarm-multiple) thread 0: duration=10, iteration=6, product=60
(alarm-multiple) thread 1: duration=20, iteration=3, product=60
```

~~这里ready_list为什么会是空链表我也暂时还没有找到原因~~
后续又调试了一下，发现ready_list不是空链表，还是有插入的，只不过比较少，所以都被其他调试语句淹没了。那现在剩下的问题应该就只有线程优先级的修改了。

```
if(list_empty(&ready_list))
  printf("Empty\n");
else printf("Something\n");
```

```
Empty
Wake up thread thread 4 at tick 475 with priority 31.
Something
Empty
```

心得体会

通过这个实验，我主要加深了对下面这些事物的了解和认识:

- 进程休眠的实现原理，以及休眠结束后进程的去向，进程调度的大致流程;
- 出现忙等待的原因;
- thread_yield()和thread_sleep()两个函数的底层实现以及功能差异;
- 抢占式优先级调度的实现原理，以及发生抢占时所需要的条件。

虽然每次都是看着课件做的实验，但都多多少少会出现一些问题，随着对操作系统理解的深入，现在已经能够自己根据出现的Error调试修改代码了，而不是像之前那样什么都先抛到浏览器上。