西安电子科技大学

操作系统课程设计

(2018 年度)

实验报告

实验名称：Alarm-Clock

班　　级：1603019

姓　　名：张俊华

学　　号：16030199025

一、实验内容

源代码 devices/timer.c 中有一个 timer_sleep() 函数。定义如下：

```
/* Sleeps for approximately TICKS timer ticks. Interrupts must
be turned on.*/
void
timer_sleep(int64_t ticks)
{
    int64_t start=timer_ticks();
    ASSERT(intr_get level()==INTR_ON);
    while(timer_elapsed(start)<ticks)
    thread_yield();
}
```

该函数的功能是让调用它的线程睡眠一段时间（ticks），然后唤醒。 事实上，Pintos 已经实现该函数，只是使用的是“忙等待”的方法（见 while 循环）。 **本实验的要求：重新实现 timer_sleep() 函数，避免“忙等待”的发生**

二、分析与设计

原始实现分析

众所周知，线程是 CPU 调度的最小单位，在 Pintos 中也不例外。本次实验，是要使用非「忙等待」的方式，实现线程的睡眠机制。

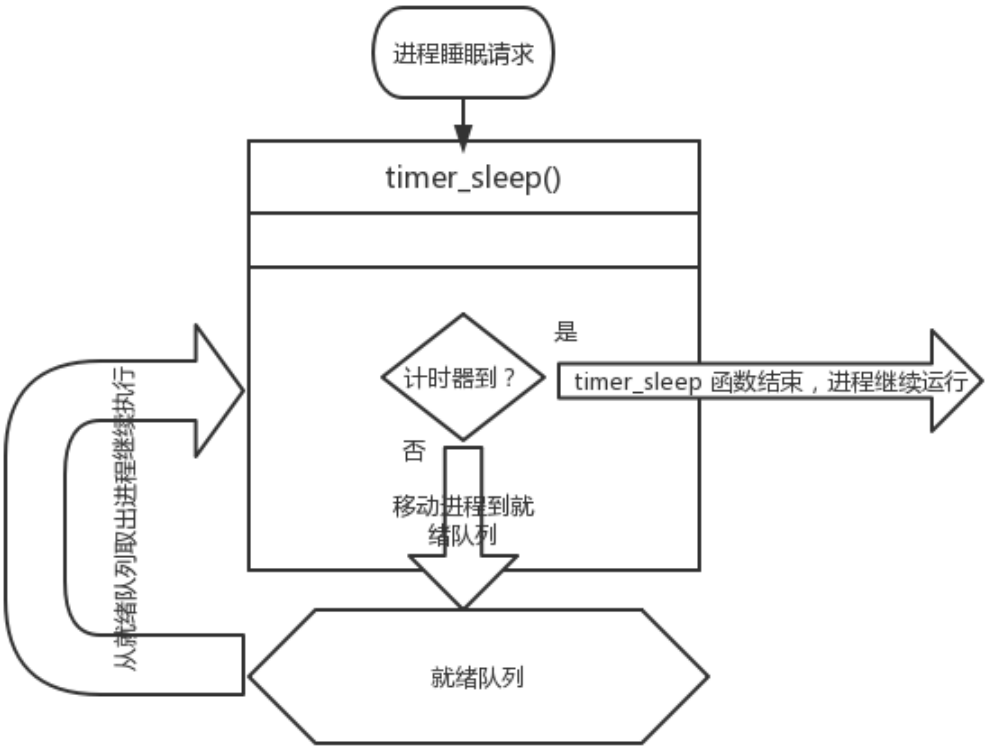
在开始实现睡眠机制前，我先对 Pintos 的 thread 项目进行了编译测试，可以看到，Pintos 已经实现了线程的睡眠功能，Make test 结果中，几个 alarm 相关的测试已经可以通过测试。先来看看 Pintos 原始是如何实现睡眠的。

devices/timer.c 中 timer_sleep() 函数实现了线程的睡眠，这个函数体只有 4 行，首先

```
int64_t start=timer_ticks();
```

获取到了当前计时器从 os 启动开始后的 tick 次数。之后使用一个 while 循环，不断查询已经过去的 ticks 次数（timer_elapsed()返回从 start 时刻起的 ticks 次数），若没有到达定时时间，就调用 thread_yield() 函数，将当前运行的线程重新移动回就绪队列，以让出 CPU 时间。

当前函数实现的流程图是这样的：



可见，虽然 `thread_yield();` 函数将进程暂时移入就绪队列，让出了 CPU，但是 CPU 会不断从就绪队列中将进程取出，不断轮询经过的 ticks 次数。在整个进程睡眠周期中，CPU 总是处于忙碌状态，是「忙等待」的睡眠机制。

非「忙等待」实现设计

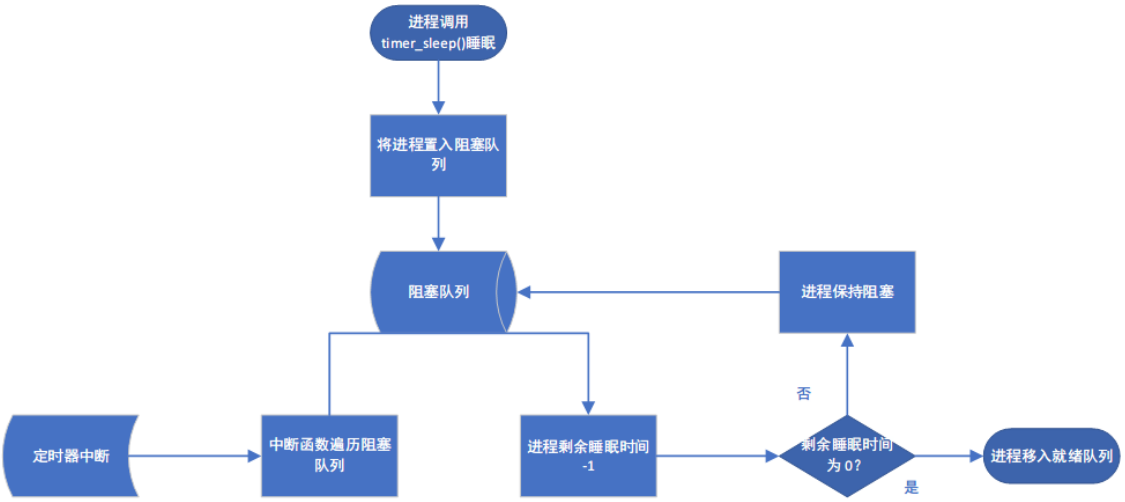
若要实现线程真正睡眠的效果，线程和 `timer_sleep()` 函数必须完全的让出 CPU 资源，按照之前的操作系统知识，只要将这个线程阻塞掉，这个进程就会释放掉它所占有的 CPU 时间片，进入要求的「睡眠」状态。

但是，单纯的 block 掉一个进程，CPU 会转向执行其他线程，这样又引发了一个问题：如何知道被阻塞的线程该结束睡眠状态并将其唤醒呢？类似于人类睡眠后可以设定一个闹钟将我们唤醒，我们也可以为进程设置一个定时器，在定时器结束之后，将线程放入就绪队列，就达到了线程唤醒的效果。这样，我们只要维护好每个进程的定时器，定时器未到之前，就可以不用对线程不断检查了。

由于 Pintos 是一个多线程的实时操作系统，因此，Pintos 一定会把 CPU 按照时间片分配给不同线程，在 Pintos 内部，也一定存在着定时器中断，以便进行进程的调度。阅读 Pintos 相关部分的源码可以发现。定时器中断在 `timer.c` 的 `timer_interrupt()` 函数中实现。故可以在 `timer_interrupt()` 函数中插入代码，在每一次定时器中断时，遍历阻塞线程的睡眠状态，找到应该唤醒的线程，将其放入就绪队列。

而如何判断线程是否该被唤醒呢？可以采用类似倒计时的方法，在线程的 PCB 中存入剩余的睡眠时间，每次遍历时将时间减 1 ，减到 0 之后，将睡眠唤醒。

重新设计之后，进程睡眠的实现方式如下所示：



所以，为了实现目标，现在需要做的任务有

- 在 `timer_sleep()` 函数中，将当前进程阻塞
- 在线程的 PCB 表中，添加计时器字段，记录剩余睡眠时间
- 编写阻塞队列的检查程序，检查线程剩余睡眠时间，按需唤醒
- 修改定时器中断 `timer_interrupt()` 函数，中断时执行阻塞队列检查程序

三、详细实现

`timer_sleep()` 函数修改

为了让线程休眠，我们要将线程阻塞，即将线程置入阻塞队列

```
thread_block() //当前运行的线程阻塞
```

线程头文件定义修改

为了记录剩余的睡眠时间，需要更改 thread 数据结构的定义，在其中增加 sleep*ticks* *字段，该字段保存剩余睡眠的 ticks 数。阅读源代码，发现 Pintos 的 ticks 数据类型为 `int64*t，因此，在 struct_thread` 中增加：

```
/* Owned by timer.c */
 int64_t sleep_ticks;
```

当然，睡眠刚开始时，剩余的睡眠时间数就是所需睡眠的时间。所以，应该在 timer_sleep() 函数中补充对 sleep_ticks 函数的初始化操作：

```
struct thread *t = thread_current();
t->sleep_ticks = ticks;
```

阻塞队列检查函数编写

在 thread.c 中定义 block_check 函数实现对阻塞线程的检查。由于 Pintos 已经为我们写好了 thread_foreach() 这个线程遍历函数，因此 block_check 配合 thread_foreach() 这个函数使用即可。阅读 thread_foreach() 这个函数可以发现，它给我们的回调函数传入了两个参数

```
struct thread *t = list_entry (e, struct thread, allelem);
func (t, aux);
```

当前遍历到的线程 t 以及一个参数 aux

因此，我们事先的 block_check() 函数也应该接受两个参数，函数原型为

```
void
block_check (struct thread *p, void* aux)
```

函数需要判断传入线程的状态，只有是阻塞状态的线程，才应当将其定时器减一，并判断定时器是否为 0 ，为 0 的话，将传入的线程唤醒，置入就绪队列

```
if(p->status == THREAD_BLOCKED  && p->sleep_ticks > 0){
    if (--(p->sleep_ticks); == 0){
        thread_unblock(p);
    }
}
```

修改定时器中断函数

按照之前设计的，需要在定时中断中对阻塞线程进行遍历，有了 block_check 函数作为基础，这个功能只需要在 timer.c 的 timer_interrput() 函数中添加一行代码即可实现

```
thread_foreach(block_check,NULL);
```

四、实验结果

按照之前的设想，Pintos 源文件修改到这里，就应该能实现我们需要的功能了，然而，在重新编译，运行 make test 之后，发现无法通过测试。仔细重新阅读

源码，发现在 thread_block() 函数的注释中有这么一行：

```
/* Puts the current thread to sleep.  It will not be scheduled
   again until awoken by thread_unblock().

   This function must be called with interrupts turned off.  It
   is usually a better idea to use one of the synchronization
   primitives in synch.h. */
void
thread_block (void)
```

由于之前在 time_sleep() 函数中，中断一直处于打开状态，而 thread_block 要求了其操作必须要保持原子性，所以应该在调用 thread_block() 函数前将中断关闭，函数执行完毕再将中断打开，这样才能保证操作的进程被顺利阻塞。故需要这样修改 time_sleep() 函数

```
enum intr_level old_level = intr_disable ();
thread_block();                    //进程暂时阻塞
intr_set_level (old_level);
```

```
pass tests/threads/alarm-single
pass tests/threads/alarm-multiple
pass tests/threads/alarm-simultaneous
FAIL tests/threads/alarm-priority
FAIL tests/threads/alarm-zero
FAIL 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
22 of 27 tests failed.
```

进行修改之后，测试样例还是没有完全通过，观察失败的测试，发现测试名称中带有 zero

negative 进一步推测，可能是当前没有考虑睡眠负数或零时间的情况，进一步调整代码，加入：

```
if (ticks <= 0)
{
  return;
}
```

增加异常处理之后，最终能够通过相关测试：

```
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.
```

五、心得体会

正如老师所讲的一样，刚开始接触到这个题目的时候，总觉得这个题目很复杂 ，之前接触进程之类的概念还仅仅停留在书本的理论阶段。但 Pintos 提供了一个非常友好的学习环境，注释也很详尽。仔细阅读源码，在加上老师的提示和讲解，任务也不难完成。但是，若要真正理解每条语句背后的运行机理，还是有很大的挑战，比如：Pintos 开关中断是怎么实现的？为什么要保持 `thread_block` 操作的原子性？Pintos 仅仅是一个最简化的学习用的操作系统，而进程仅仅是 Pintos 实现的一小部分，要完整阅读其源码就已经非常困难。这次的任务仅仅是冰山一角，若要一窥 Pintos 的全貌，真正有所收获，还需要更大的努力。