|  |  |
| --- | --- |
|  |  |
|  |  |





Threads实验报告

学院：计算机与信息技术学院

专业： 计算机科学与技术

成员： 张家珲 黄景涛

王 欣 彭铁虎

教师： 何永忠

2017年 4月 22日

目录

[1. 实验目标 2](#_Toc480814680)

[2. 实验原理 2](#_Toc480814681)

[3. 实验内容 2](#_Toc480814682)

[3.1 了解线程 2](#_Toc480814683)

[3.2 阅读源文件 2](#_Toc480814684)

[**3.2.1 threads目录下的文件** 3](#_Toc480814685)

[**3.2.2 devices目录下的文件** 4](#_Toc480814686)

[**3.2.3 lib目录下的文件** 5](#_Toc480814687)

[3.3 设计文件 6](#_Toc480814688)

[3.3 闹钟 7](#_Toc480814689)

[**3.3.1 源码解析** 7](#_Toc480814690)

[**3.3.2 timer\_sleep()重新实现** 8](#_Toc480814691)

[**3.3.3 timer\_sleep()唤醒机制测试结果** 12](#_Toc480814692)

[4. 遇到的问题及解决方案 12](#_Toc480814693)

1. 实验目标

Pintos是80x86架构的简单操作系统框架。Threads是pintos的第一个项目，在这个项目中，pintos提供了一个最低功能的线程系统，此次实验的目标是在此基础上拓展此系统的功能，以更好地了解线程同步问题。此次实验主要在Threads目录中进行工作，并在devices目录中改写一些作业，编译过程应该在Threads目录中完成。

实验最终的目标是在Threads中进行make check工作时，27个test全部pass。

**2. 实验原理**

通过 bochs 加载 pintos 操作系统， 该操作系统会根据 pintos 的实现打印运行结果，通过比较标准输出文档和实际输出，来判断 pintos 实现是否符合要求。

3. 实验内容

以下是Threads实验的流程：

3.1 了解线程

Pintos已经实现了线程的创建和完成，使用简单的调度程序在线程之间切换，同时完成了同步原语（信号量，锁，条件变量和优化障碍）的功能。

创建线程时，会创建一个需要排定的上下文，线程创建时提供了一个在此上下文中运行的函数作为thread\_create()的参数，线程第一次调度并运行，它从该函数的开始开始，并在该上下文中执行。当函数返回时，线程终止。因此，每个线程都像在Pintos中运行的一个小程序，其功能被传递给thread\_create()。

在任何给定的时间，只有一个线程运行，其余的线程（如果有的话）变为非活动状态。调度程序决定下一个运行哪个线程。如果在任何给定的时间内没有线程准备好，那么特殊的“空闲”线程idle()就会运行。当一个线程需要等待另一个线程时 ，同步原语可以强制上下文切换。

上下文切换的机制是thread / switch.S，它是80 x 86的汇编代码。它保存当前正在运行的线程的状态，并恢复我们切换到的线程的状态。

3.2 阅读源文件

以下是Threads实验中涉及到的文件的简要概述：

**3.2.1 threads目录下的文件**

**loader.S**

**loader.h**

内核加载器。装载为512字节的代码和数据，由PC BIOS加载到存储器中，并跳转到start.S中的start()。

**start.S**

在80 x 86 CPU 上进行内存保护和32位操作所需的基本设置。与加载程序不同，这段代码实际上是内核的一部分。

**kernel.lds.S**

用于链接内核的链接描述文件。设置内核的加载地址和start.S的安排，使其接近内核映像的开头。

**init.c**

**init.h**

内核初始化，包括内核的“主程序”，可能需要在此添加自己的初始化代码。

**thread.c**

**thread.h**

基本线程支持。大部分工作将在这些文件中进行。 thread.h定义struct thread，可能需要在所有四个项目中进行修改。

**switch.S**

**switch.h**

用于切换线程的汇编语言程序。

**palloc.c**

**palloc.h**

页面分配器，以4 kB页面的倍数输出系统内存。

**malloc.c**

**malloc.h**

内核的malloc()和free()的简单实现。

**interrupt.c**

**interrupt.h**

基本的中断处理和打开和关闭中断的功能。

**intr-stubs.S**

**intr-stubs.h**

用于低级别中断处理的汇编代码。

**synch.c**

**synch.h**

基本同步原语：信号量，锁，条件变量和优化障碍。实验中将需要在所有四个项目中使用它们进行同步。

**io.h**

I / O端口访问功能。

**vaddr.h**

**pte.h**

用于处理虚拟地址和页表项的功能和宏。这些在项目3中较为重要，目前不需考虑。

**flags.h**

在80 x 86“标志”寄存器中定义几个bits内容的宏。

**3.2.2 devices目录下的文件**

**timer.c**

**timer.h**

系统定时器默认为100次。此项目中需要修改此代码。

**vga.c**

**vga.h**

VGA显示驱动。负责将文字写入屏幕。printf() 调用VGA显示驱动程序。

**serial.c**

**serial.h**

串口驱动。它通过将其传递到输入层来处理串行输入（见下文）。

**block.c**

**block.h**

块设备的抽象层，即随机访问，组织为固定大小块数组的类似磁盘的设备。开箱即用，Pintos支持两种类型的块设备：IDE磁盘和分区。

**ide.c**

**ide.h**

支持最多4个IDE磁盘的读写扇区。

**partition.c**

**partition.h**

了解磁盘上分区的结构，允许将单个磁盘刻成多个区域（分区）以供独立使用。

**kbd.c**

**kbd.h**

键盘驱动程序。处理按键的方式为将它们传递到输入层。

**input.c**

**input.h**

输入层。队列输入的字符由键盘或串行驱动程序传递。

**intq.c**

**intq.h**

中断队列，用于管理内核线程和中断处理程序想要访问的循环队列。由键盘和串行驱动程序使用。

**rtc.c**

**rtc.h**

实时时钟驱动，使内核能够确定当前的日期和时间。默认情况下，这仅由线程/ init.c 用于为随机数生成器选择初始种子。

**speaker.c**

**speaker.h**

可以在PC扬声器上产生音调的驱动程序。

**pit.c**

**pit.h**

配置8254可编程中断定时器的代码。/ timer.c和devices / speaker.c 都使用这个代码，因为每个设备都使用PIT的一个输出通道。

**3.2.3 lib目录下的文件**

**debug.c**

**debug.h**

函数和宏，作用为帮助调试。

**random.c**

**random.h**

伪随机数发生器。随机值的实际顺序将不会随着Pintos的运行而变化，除非执行以下三项操作之一：在-rs内核命令行中指定一个新的随机种子值用于每次运行，或使用除Bochs之外的模拟器，或为pintos指定-r选项。

**round.h**

四舍五入的宏。

**syscall-nr.h**

系统号码。

**kernel / list.c**

**kernel / list.h**

双链表实现。使用了所有的Pintos代码。

**kernel / bitmap.c**

**kernel / bitmap.h**

位图实现。

**kernel / hash.c**

**kernel / hash.h**

哈希表实现。

**kernel / console.c**

**kernel / console.h**

**kernel / stdio.h**

实现printf()和其他一些功能。

3.3 设计文件

在打开项目之前，需要将项目1的设计文档模板复制到源代码树中，名称为”pintos/src/threads/DESIGNDOC”，并根据实际情况做出更改。

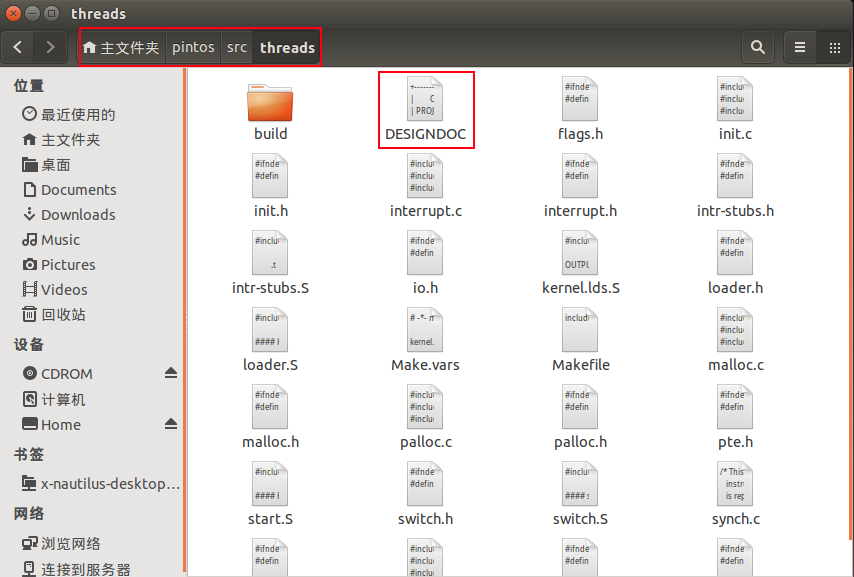


图 1 项目设计文档位置

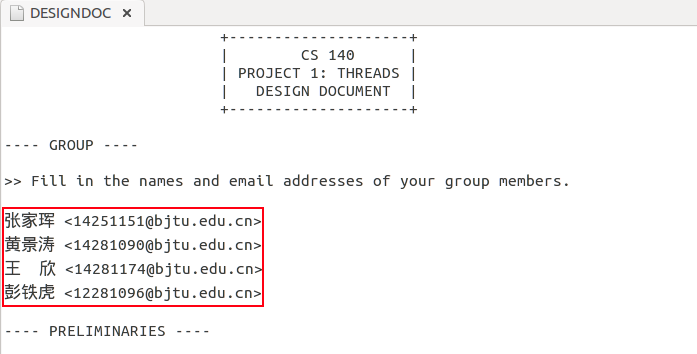


图 2 项目设计文档内容

3.3 闹钟

重新实现 ”devices/timer.c” 文件中的timer\_sleep()函数。

**3.3.1 源码解析**

1. /\* Sleeps for approximately TICKS timer ticks.  Interrupts must
2. be turned on. \*/
3. **void**
4. timer\_sleep (int64\_t ticks)
5. {
6. int64\_t start = timer\_ticks ();
7. ASSERT (intr\_get\_level () == INTR\_ON);
8. **while** (timer\_elapsed (start) < ticks)
9. thread\_yield();
10. }

这是原始代码中timer\_sleep()的实现。系统现在是通过使用busy wait来实现的，即线程不停地循环检查当前时间并调用thread\_yield()，直到过去足够长的时间。

1. int64\_t start = timer\_ticks ();

此处调用了timer\_ticks() 函数：

1. /\* Returns the number of timer ticks since the OS booted. \*/
2. int64\_t
3. timer\_ticks (**void**)
4. {
5. **enum** intr\_level old\_level = intr\_disable ();
6. int64\_t t = ticks;
7. intr\_set\_level (old\_level);
8. **return** t;
9. }

在以下两个语句包裹的内容的目的是保证这个过程不会被中断：

1. **enum** intr\_level old\_level = intr\_disable ();
2. ...
3. intr\_set\_level (old\_level);

ticks获取了当前的ticks值，从pintos被启动开始，ticks就一直在计时，代表着操作系统执行单位时间的前进计量。

start获取了进程起始的时间，然后断言必须可以被中断，不然会一直死循环下去，接下来的就是一个循环：

1. **while** (timer\_elapsed (start) < ticks)
2. thread\_yield();

这个循环的实质就是在ticks的时间内不断执行thread\_yield()。thread\_yield其实就是把当前线程扔到就绪队列里， 然后重新schedule， 注意这里如果ready队列为空的话当前线程会继续在cpu执行。

最后回溯到我们最顶层的函数逻辑： timer\_sleep就是在ticks时间内， 如果线程处于running状态就不断把他扔到就绪队列不让他执行。

这种实现方式的缺点是：线程依然不断在cpu就绪队列和running队列之间来回， 占用了cpu资源， 这并不是我们想要的， 我们希望用一种唤醒机制来实现这个函数。

**3.3.2 timer\_sleep()重新实现**

我们需要将timer\_sleep()重新实现，以避免忙碌等待。

实现的思路为：调用timer\_sleep的时候直接把线程阻塞掉，然后给线程结构体加一个成员ticks\_blocked来记录这个线程被sleep了多少时间， 然后利用操作系统自身的时钟中断（每个tick会执行一次）加入对线程状态的检测， 每次检测将ticks\_blocked减1, 如果减到0就唤醒这个线程。

1. /\* Sleeps for approximately TICKS timer ticks.  Interrupts must
2. be turned on. \*/
3. **void**
4. timer\_sleep (int64\_t ticks)
5. {
6. **if** (ticks <= 0)
7. {
8. **return**;
9. }
10. ASSERT (intr\_get\_level () == INTR\_ON);
11. **enum** intr\_level old\_level = intr\_disable ();
12. **struct** **thread** \*current\_thread = thread\_current ();
13. current\_thread->ticks\_blocked = ticks;
14. thread\_block ();
15. intr\_set\_level (old\_level);
16. }

以上是修改后的timer\_sleep()代码。

在代码中调用了thread\_block()函数：

1. /\* Puts the current thread to sleep.  It will not be scheduled
2. again until awoken by thread\_unblock(). \*/
3. **void**
4. thread\_block (**void**)
5. {
6. ASSERT (!intr\_context ());
7. ASSERT (intr\_get\_level () == INTR\_OFF);
9. thread\_current ()->status = THREAD\_BLOCKED;
10. schedule ();
11. }

给线程的结构体加入ticks\_blocked成员：

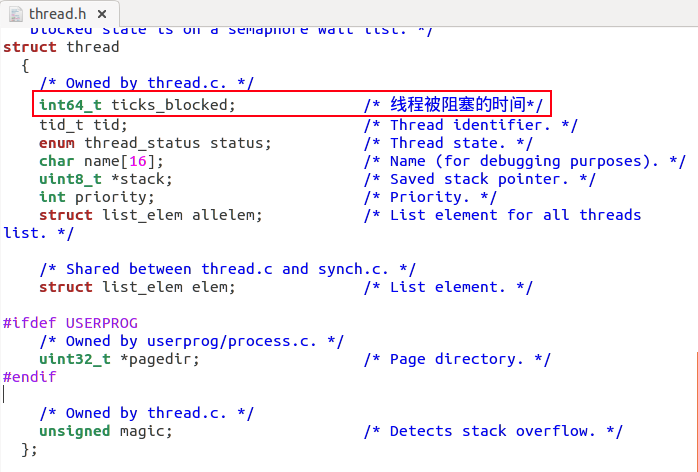


图 3 thread结构体

然后在线程被创建的时候初始化ticks\_blocked为0， 加在thread\_create函数内：

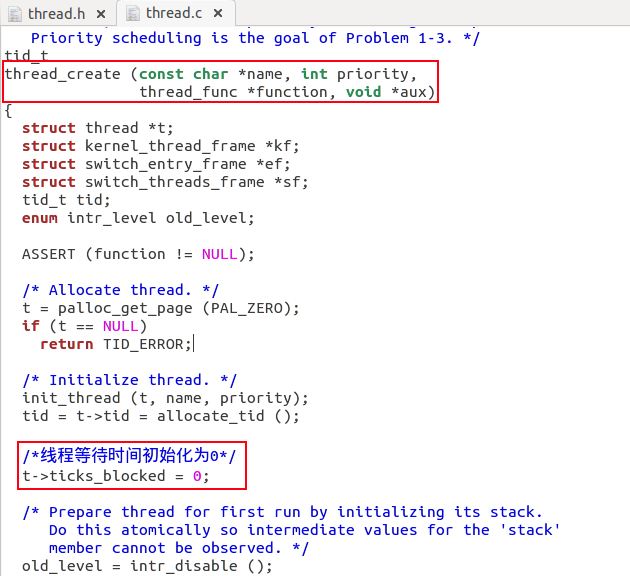


图 4 初始化等待时间

然后修改时钟中断处理函数， 加入线程sleep时间的检测， 加在timer\_interrupt内：

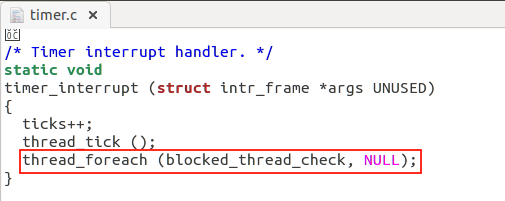


图 5 检测线程sleep时间

这里的thread\_foreach(blocked\_thread\_check, void aux\*)函数就是对每个线程都执行blocked\_thread\_check这个函数，aux为传给这个函数的参数。

然后在thread.h中声明blocked\_thread\_check方法：

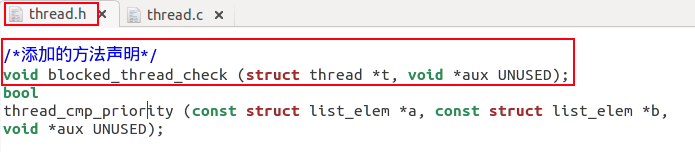


图 6 方法声明

在thread.c中写入方法实体：



图 7 方法实体

其中用到的thread\_unblock() 函数就是将线程放入就绪队列中继续运行。

1. /\* Transitions a blocked thread T to the ready-to-run state.
2. This is an error if T is not blocked.  (Use thread\_yield() to
3. make the running thread ready.)
5. This function does not preempt the running thread.  This can
6. be important: if the caller had disabled interrupts itself,
7. it may expect that it can atomically unblock a thread and
8. update other data. \*/
9. **void**
10. thread\_unblock (**struct** **thread** \*t)
11. {
12. **enum** intr\_level old\_level;
14. ASSERT (is\_thread (t));
16. old\_level = intr\_disable ();
17. ASSERT (t->status == THREAD\_BLOCKED);
18. list\_push\_back (&ready\_list, &t->elem);
19. t->status = THREAD\_READY;
20. intr\_set\_level (old\_level);
21. }

**3.3.3 timer\_sleep()唤醒机制测试结果**

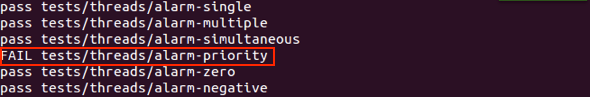


图 8 唤醒机制测试结果

4. 遇到的问题及解决方案