操作系统课程实验

2016-03-28

课程计划

- 第5周：实验环境安装与搭建(part0)
- 第6~7周：线程的休眠与唤醒(part1)
- 第8~9周：优先级调度(part2)
- 第10~11周：线程锁、信号量与优先级继承(part3)
- 第12~14周：多级反馈队列调度(part4)

实验一.线程休眠与唤醒

需要解决：

（1）通过修改pintos的线程休眠函数来保证pintos不会在一个线程休眠时忙等待。

（2）通过修改pintos排队的方式来使得所有线程正确地被唤醒。

相关文件

pintos/src/devices目录：

timer.h, timer.c

pintos/src/threads目录：

thread.h, thread.c

Pintos/src/lib目录

需要阅读相关结构体以及函数。

Pintos的中断机制

出于安全性等考虑，每隔一段时间操作系统必须获得CPU时间，进行进程调度等工作。而操作系统是通过中断来获得CPU时间。pintos中操作系统中断频率为：

```
/* Number of timer interrupts per second. */
#define TIMER_FREQ 100
```

Timer中断产生时以下函数就会被调用。

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

中断相关

pintos/src/threads目录：

interrupt.c 和 interrupt.h

```
/* Interrupts on or off? */
enum intr_level
  {
    INTR_OFF,             /* Interrupts disabled. */
    INTR_ON               /* Interrupts enabled. */
  };
```

该枚举定义了中断是开还是关，在原子操作中必须保证中断是关的。

用函数interrupt_disable()来关闭中断,这个函数是有返回值的!

休眠

线程相关

Pintos中定义了一个thread的结构体用于存储线程的信息(**包括优先级，状态**等)，位于thread.h。

其中thread有四个状态：

```
/* States in a thread's life cycle. */
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. */
  };
```

这个结构体非常重要！

一个线程里有什么？

```
struct thread
  {
    /* Owned by thread.c. */
    tid_t tid;                          /* Thread identifier. */
    enum thread_status status;          /* Thread state. */
    char name[16];                      /* Name (for debugging purposes). */
    uint8_t *stack;                     /* Saved stack pointer. */
    int priority;                       /* Priority. */
    struct list_elem allelem;           /* List element for all threads list. */

    /* Shared between thread.c and synch.c. */
    struct list_elem elem;              /* List element. */

#ifdef USERPROG
    /* Owned by userprog/process.c. */
    uint32_t *pagedir;                  /* Page directory. */
#endif

    /* Owned by thread.c. */
    unsigned magic;                     /* Detects stack overflow. */
  };
```

实验相关：线程休眠

timer.c中实现了线程休眠的函数timer_sleep(是sleep，不是nsleep不是usleep也不是msleep)

```
/* Sleeps for approximately TICKS timer ticks.  Interrupts must
   be turned on. */
void
timer_sleep (int64_t ticks)
{
  int64_t start = timer_ticks ();
  //ASSERT->括号里不为真就退出
  ASSERT (intr_get_level () == INTR_ON);
  while (timer_elapsed (start) < ticks)
    thread_yield ();
}
```

这个函数作用是让线程休眠一定的时间。
线程休眠时必须保证中断是打开的。

线程休眠

timer_sleep的实现原理是通过不断轮询检查经过时间是否达到ticks，若还没达到则调用thread_yield函数，达到了ticks就会结束休眠。

Ticks: pintos 的计时单位

thread_yield

```
/* Yields the CPU.  The current thread is not put to sleep a
   may be scheduled again immediately at the scheduler's wh
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);
  cur->status = THREAD_READY;
  schedule ();
  intr_set_level (old_level);
}
```

该函数会把当前线程放进ready队列，并调度下一个线程，线程调度时要保证中断关闭。

schedule

专门负责线程切换的函数，执行了以后会把当前线程放进队列里并调度出下一个线程。

```
/* Schedules a new process.  At entry, interrupts must be off and
   the running process's state must have been changed from
   running to some other state.  This function finds another
   thread to run and switches to it.

   It's not safe to call printf() until thread_schedule_tail()
   has completed. */
static void
schedule (void)
{
  struct thread *cur = running_thread ();
  struct thread *next = next_thread_to_run ();
  struct thread *prev = NULL;

  ASSERT (intr_get_level () == INTR_OFF);
  ASSERT (cur->status != THREAD_RUNNING);
  ASSERT (is_thread (next));

  if (cur != next)
    prev = switch_threads (cur, next);
  thread_schedule_tail (prev);
}
```

线程休眠问题

Thread_yield函数只是把线程放进调度队列，然后切换线程，此时休眠线程状态是ready，然后running下一个线程，但此时timer_sleep函数的while循环还在进行，就导致线程依然不断在cpu的ready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。

目标

通过重新设计thread_sleep函数让休眠线程不再占用cpu时间，只在每次tick中断把时间交给操作系统时再检查睡眠时间，tick内则把cpu时间让给别的线程。

解决办法

1.重写timer_sleep函数，不是调用thread_yield而是调用thread_block函数把线程block了，这样在unblock之前该线程都不会被调度执行。

2.

（1）Thread结构体中新增成员变量ticks_blocked；记录线程剩余睡眠的时间(单位：tick)。

```
int ticks_blocked;
```

（2）初始化为0；

（3）在timer_sleep函数中把tick赋值为睡眠时间。

```
enum intr_level old_level = intr_disable ();
struct thread* curThread = thread_current();
curThread->ticks_blocked = ticks; //the ticks the thread should sleep
thread_block();
intr_set_level (old_level);
```

解决办法

3.thread.c中增加函数blocked_thread_check判断线程是否休眠完毕(ticks_blocked的值)。

```
void
thread_check_and_block(struct thread* t,void *aux UNUSED)
{
  if(t->status == THREAD_BLOCKED && t->ticks_blocked != 0)
  {
    t->ticks_blocked--;
    if(t->ticks_blocked == 0)
    {
        thread_unblock(t);
    }
  }
}
```

4.处理timer中断时(timer_interrupt)遍历所有线程(thread_for_each，详细见代码)并判断被block线程是否睡眠完毕，睡眠完毕则唤醒。

唤醒

唤醒机制

完成上述步骤以后，大部分的alarm测试都通过了。但是alarm_priority依然还没通过。接下来介绍解决alarm_priority的知识。

```
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
```

list

```
/* List element. */
struct list_elem
  {
    struct list_elem *prev;     /* Previous list element. */
    struct list_elem *next;     /* Next list element. */
  };

/* List. */
struct list
  {
    struct list_elem head;      /* List head. */
    struct list_elem tail;      /* List tail. */
  };
```

list.h: 位于pintos\src\lib\kernel目录下。list中存放的数据类型是list_elem，存放待执行线程靠的就是list。

Thread与list

因为线程很忙的，所以线程派一个小弟去帮它排队:这个小弟既要负责排队，还要负责同步。有一个list变量ready_list专门存放待执行的线程，由于list存放的是list_elem类型，因此thread中有一个成员变量专门用于在ready_list中“排队”。

```
/* Shared between thread.c and synch.c. */
struct list_elem elem;              /* List element. */
```

Thread与list

既然在ready_list中并不是thread而只是thread的一个成员，怎么通过list中的elem来访问对应线程的其他变量？List中已经提供了一个宏定义解决这个问题。

```
/* Converts pointer to list element LIST_ELEM into a pointer to
   the structure that LIST_ELEM is embedded inside.  Supply the
   name of the outer structure STRUCT and the member name MEMBER
   of the list element.  See the big comment at the top of the
   file for an example. */
#define list_entry(LIST_ELEM, STRUCT, MEMBER)           \
        ((STRUCT *) ((uint8_t *) &(LIST_ELEM)->next     \
                     - offsetof (STRUCT, MEMBER.next)))
```

解决办法

alarm-priority要求线程是根据优先级进行调度的，而pintos的基本实现是直接把线程放进ready_list尾部，是FIFO的调度方式。如果要通过这个测试就必须保证在ready_list中线程是有序的。

hints：使用list_insert_order函数可以保证有序插入。

注意事项

1.先阅读源代码和测试的代码。把握线程的整体结构，以后做实验也会轻松一点。

2.调用函数前要看看该函数是否要求关闭或者打开中断。

3.修改前最好备份一下。

此时需要make check：如果通过了所有的alarm测试，那么恭喜你，你可以开始写报告了，如果没有，你自己慢慢debug去吧。加油！

要求

1.回答问题：

(1) 之前为什么20/27? 为什么那几个test在什么都没有修改的时候还过了？

(2) 一道经典面试题：线程与进程的区别？

(3)另一道经典面试题：中断和轮询的区别？

2. 解说一下每个测试做了什么，并描述一下过程

3.关键代码截图，实验感想

报告注意事项

- 实验结果图要截到名字哦！
- 报告和源码分开交，具体格式参照上一次PPT。
- 超过零点就别交FTP啦，补交请交到 zhanxy5@mail2.sysu.edu.cn
- 补交格式：学号_姓名_labx_补交.zip，包含源码和PDF。
- 本次实验截止日期为4.10，中途经历一次放假，但是大家清明节也不要忘记写pintos哦！