Pintos简介
Project 1

目标：完成Project1

Pintos简介

1. 目录结构
2. 编译、运行、测试

目标：完成Project1

Pintos目录结构

1. *threads:为基核准备的源代码，在实验一中我们会进行修改。*
2. *userprog:为装载用户程序的源代码，在实验二我们会进行修改。*
3. *vm:基本上空的目录，在实验三我们实现虚拟内存。*
4. *filesys:基本文件系统的源代码，从实验二开始使用，实验四开始修改。*
5. *devices:键盘、定时器、硬盘等等IO设备的接口源代码，在实验一中我们会修改定时器。其他情况我们不会进行修改。*
6. *lib:标准C库的子集实现。这个目录中的代码被编译到内核中，并且从实验二开始，用户程序也会在其下运行。在内核代码和用户程序中，我们可以使用`#include<...>`的方式引用这个目录中的`header`文件。我们基本上不用修改本目录中的源代码。*

Pintos目录结构

7. lib/kernal:这个目录中的代码仅被内核使用。还包含在内核代码中可以自由使用的一些数据类型的实现:位图、双向链表和哈希表。在内核代码中，我们可以使用 `#include<...>` *的方式引用这个目录中的* `header` *文件。*

8. lib/user:这个目录中的代码仅被用户程序使用。在用户程序中，我们可以使用 `#include<...>` *的方式引用这个目录中的* `header` *文件。*

9. test:每个项目的测试。如果它可以帮助您测试提交用例，可以自行修改。

10. examples:从实验二开始使用的示例用户程序。

*11. misc & utils:如果尝试在自己的计算机上运行Pintos才会用到这些文件。**请在当前用户目录下的*** `.zshrc` ***或*** `.bashrc` ***文件中添加*** `utils` ***文件夹的环境变量。***

开始操作：第一步，编译

到src文件夹下

到threads文件夹下

执行make命令（如果已经有build文件夹了，则先删除它）

build文件夹下的内容

1. Makefile：pintos/src/Makefile.build的复制。它描述了如何构建内核。

2. Kernel.o 内核对象（目标文件）

3. Kernel.bin 内核映像（二进制文件）

4. Loader.bin 内核加载器的内存映像

5. 构建的文件夹中同样包含子文件夹，子文件夹中的内容由.o 与.d 文件组成。

接着操作：第二步，运行与调试

Step 1. 启动终端，进入**threads/build**目录,执行命令

> pintos --gdb -s -- run alarm-multiple（当中有个空格）

Step 2. 启动另一个终端，进入**threads/build**目录，输入命令

> gdb kernel.o

Step 3. 连接pintos虚拟机

> target remote localhost:1234

Step 3. 看结果

make check

make check的结果

```
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.
make[1]: *** [check] 错误 1
make[1]:正在离开目录 `/home/virtual/pintos/src/threads/build'
make: *** [check] 错误 2
```

Q&A

1. 在调试时，容易出现报错：

Warning: can’t find squish-pty，so terminal input will fail

解决方案：

在pintos/src/utils目录中执行make命令，使用sudo ln squish-pty /usr/local/bin/ 解决。

Pintos Projects

Project的内容

Project1-----进程（线程）管理问题

- 忙等待
- CPU调度问题（优先级调度、多级反馈队列调度）

Project2-----用户应用程序的管理

- 解决参数传递（argument passing）与系统调用（system call）等用户程序的应用问题

Project 1

任务1、忙等待问题

任务2、优先级调度

任务3、多级反馈队列调度（BSD调度）

忙等待相关文件(thread)

(1)loader.S loader.h ——内核调度器loader ， PC BIOS 调入内存，找到内核，调到内存，然后执行 start.S中start() 函数.不需要修改

(2)init.c init.h——内核初始化，包含main()----内核主程序，你可以加入初始化代码

(3)**thread.c thread.h——基本内核程序，thread.h定义结构体:thread，在项目中可以修改**

(4)switch.S switch.h——switch.S 汇编语言的程序

(5)palloc.c palloc.h——页面调度器，在内存分配时使用，在页面调度时使用

忙等待相关文件

(6)malloc.c malloc.h——在内核中malloc() 和 free()实现

(7) interrupt.c interrupt.h——基本的中断处理函数，设置终端on 和off.

(8)intr-stubs.S intr-stubs.h——汇编程序，处理低级终端

(9)synch.c synch.h——基本同步原语:信号量，锁，条件变量，在四个项目中都要使用这些同步。

(10)io.h ——I/O端口访问，被devices目录中文件用，你可以不修改 .

(11)vaddr.h pte.h——在虚地址和页面表中的函数及宏定 义，在后继项目中使用

(12)flags.h ——80x86标志寄存器中使用的宏定 义，你可以不修改

Project 1

需要阅读

（/threads/）

Thread.h, thread.c,

（/device/）

interrup.h, time.c这四个文件

修改完善timer.c中的timer_sleep函数

任务一:忙等待, timer.c

什么是忙等待呢？

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

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

任务一:忙等待, thread.h

```
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. */
  };
```

任务一:忙等待, thread.h

```
/* 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. */
  };
```

任务一：忙等待，thread.c

- **thread_current()** 获取当前当前的线程的指针
- **thread_foreach(thread_action_func *func, void *aux)**遍历所有线程，并且对每一个线程t都执行aux函数，且保证中断关闭。
- **thread_block()和thread_unblock(thread *t)**这是一对函数，区别在于第一个函数的作用是把当前占用cpu的线程阻塞掉（就是放到waiting里面），而第二个函数作用是将已经被阻塞掉的进程t唤醒到ready队列中。

任务1：解决忙等待问题

timer_sleep()通过循环条件询问OS当前时间是否大于或者等于ticks时间刻度，不满足会调用thread_yield()，会将当前线程扔到就绪队列里，重新调度。这就是问题所在，CPU在处理该进程时，不断将其在就绪态与运行态之间切换，会浪费时间。

目标：

修改文件代码，避免切换进程状态。

任务2:优先级调度问题

目的: 实现pintos中对线程按优先级调度

进程管理中priority参数没有充分利用

我们需要利用priority参数，修改并完善优先级调度功能

现有的实现机制是:FIFO

修改整体调度机制

在等待队列中按优先级大小进行排列，实现按优先顺序进行调度

Semaphore

可以通过semaphore信号量参数来实现对资源的控制

value---资源锁，我们对其进行追踪

当value减小(sema_down)时，对使用此资源的进程进行阻塞。反之(sema_up)加入待执行队列(优先级调度应按优先级大小排序)

Semaphore

```
/* A counting semaphore. */
struct semaphore
  {
    unsigned value;             /* Current valu
    struct list waiters;        /* List of wait
  };

void sema_init (struct semaphore *, unsigned va
void sema_down (struct semaphore *);
bool sema_try_down (struct semaphore *);
void sema_up (struct semaphore *);
void sema_self_test (void);
```

Semaphore

在threads/thread.c中还要修改如下两个函数以保证进程可以修改或获得优先级

void thread_set_priority (int new_priority)

设置进程的优先级

int thread_get_priority (void)

返回进程的优先级

Advanced Scheduler（可选）

- 实现多级反馈队列调度（BSD调度）以使运行中的平均反应时间更小
- 通常需要多种调度方式适用于不同的调度需求，因此对于不同的设备（比如I/O，需要不断申请IO，但是对CPU需求很少）应该采取不同的调度方式。

我们期待的结果

```
pass tests/threads/mlfqs-load-60
pass tests/threads/mlfqs-load-avg
pass tests/threads/mlfqs-recent-1
pass tests/threads/mlfqs-fair-2
pass tests/threads/mlfqs-fair-20
pass tests/threads/mlfqs-nice-2
pass tests/threads/mlfqs-nice-10
pass tests/threads/mlfqs-block
All 27 tests passed.
```