

中山大学本科生实验报告

(2018 学年春季学期)

课程名称: Operating System

任课教师: 饶洋辉

年级+班级	16 级+2 班	专业 (方向)	信息与计算科学
学号	16343065	姓名	桑娜

1. 实验目的

搭建 Linux 环境并完成 Pintos 的配置。

2. 实验过程

注: 因为各种原因, 没有按照实验建议使用 VMware, 而是使用 Win10 下的 Linux 子系统 (Windows Subsystem for Linux)。不知道是不是这个原因, 用 bochs 配置时候出现了一些错误。尝试多次都没有解决, 因此最后使用的是 qemu。

(1) 安装 Linux

① 开启开发人员模式

打开“设置”→“更新和安全”→“针对开发人员”, 在“使用开发人员功能”中选择“开发人员模式”。



针对开发人员

使用开发人员功能

这些设置只用于开发。

[了解更多信息](#)

☐ Windows 应用商店应用

仅安装 Windows 应用商店的应用。

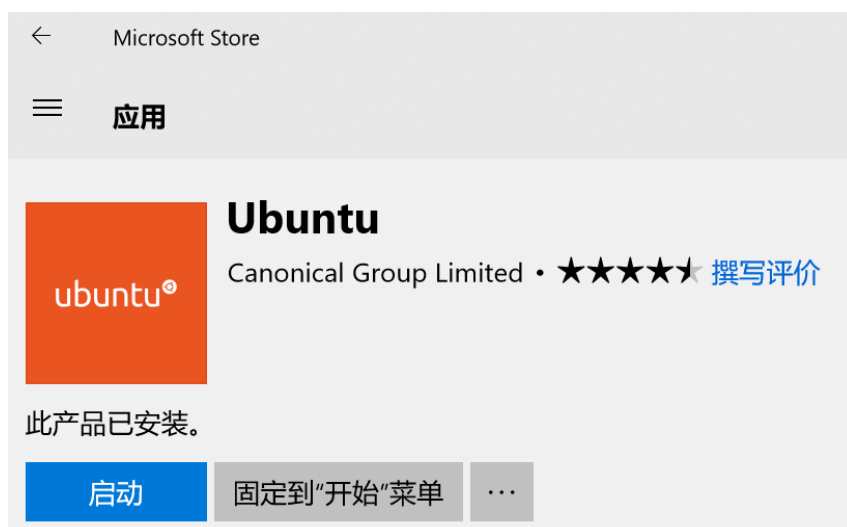
☐ 旁加载应用

从你信任的其他来源 (例如工作区) 安装应用。

☒ 开发人员模式

安装任何已签名的可信应用并使用高级开发功能。

② 打开 Microsoft Store，安装 Ubuntu。



③ 安装 Desktop

因为 WSL(Windows Subsystem for Linux)并没有 GUI，所以还需要另外安装配置 Desktop，但这个内容与本实验无关，故略去不写。

④ 更改 apt 源

因为默认的 apt 下载起来比较慢而且容易缺失，故将 apt 源更换为国内的镜像源，我采用的是中科大源。

```
$ cd etc/apt
$ sudo cp sources.list sources.list.bak
$ sudo gedit sources.list
```

将 sources.list 中原有内容替换为

```
deb https://mirrors.ustc.edu.cn/ubuntu/ xenial main restricted universe
multiverse

deb https://mirrors.ustc.edu.cn/ubuntu/ xenial-updates main restricted
universe multiverse

deb https://mirrors.ustc.edu.cn/ubuntu/ xenial-backports main restricted
universe multiverse

deb https://mirrors.ustc.edu.cn/ubuntu/ xenial-security main restricted
universe multiverse
```

保存，关闭即可。

(2) 安装 qemu

```
$ sudo apt-get install qemu
```

(3) 安装 Pintos

① 下载

地址: <http://www.stanford.edu/class/cs140/projects/pintos/pintos.tar.gz>

② 提取

打开/home/sangna/文件夹, 右键 pintos.tar.gz, 选择 “Extract Here”。

③ 设置环境变量

```
$ sudo gedit ~/.bashrc
```

在打开的文件中, 在最后一行添加

```
export PATH="$PATH:/home/sangna/pintos/src/utils"
```

```
$ source ~/.bashrc
```

④ 修改相关文件

```
$ gedit pintos/src/threads/Make.vars
```

Line 7: SIMULATOR = --qemu

```
$ gedit pintos/src/utils/pintos
```

Line 103: 将 bochs 换为 qemu.

Line 259: 将 kernel.bin 替换为 /home/sangna/pintos/src/threads/build/kernel.bin

```
$ gedit pintos/src/utils/Pintos.pm
```

Line 362: \$KERNEL_ROOT/src/threads/build/loader.bin.

```
$ gedit pintos/src/utils/pintos-gdb
```

Line 4: GDBMACROS=\$PINTOS_ROOT/src/misc/gdb-macros

```
$ gedit pintos/src/utils/Makefile
```

Line 5: LDLIBS = -lm

⑤ 修复错误

```
$ gedit pintos/src/devices/shutdown.c
```

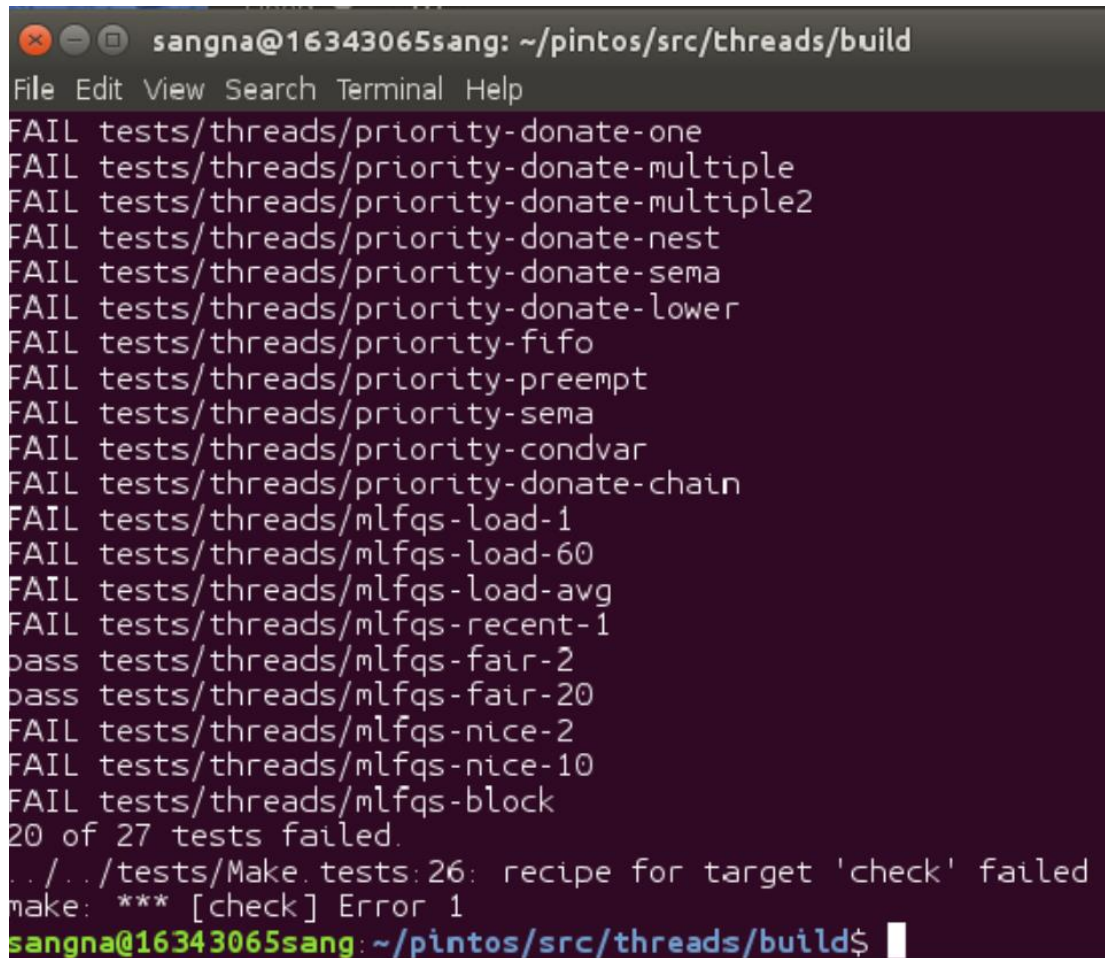
```
printf ("Powering off...\n");
```

```
serial_flush ();  
  
//增加这一行  
  
outw( 0x604, 0x0 | 0x2000 );
```

(4) 编译及测试

```
$ cd pintos/src/utils  
$ make  
$ cd src/threads  
$ make  
$ cd build  
$ make check
```

3. 配置结果



A terminal window titled 'sangna@16343065sang: ~/pintos/src/threads/build' showing the output of a 'make check' command. The terminal lists 27 tests, most of which failed. The tests are: priority-donate-one, priority-donate-multiple, priority-donate-multiple2, priority-donate-nest, priority-donate-sema, priority-donate-lower, priority-fifo, priority-preempt, priority-sema, priority-condvar, priority-donate-chain, mlfqs-load-1, mlfqs-load-60, mlfqs-load-avg, mlfqs-recent-1, mlfqs-fair-2, mlfqs-fair-20, mlfqs-nice-2, mlfqs-nice-10, and mlfqs-block. The first 20 tests failed, and the last 7 passed. The terminal output is as follows:

```
sangna@16343065sang: ~/pintos/src/threads/build  
File Edit View Search Terminal Help  
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.  
./././tests/Make.tests:26: recipe for target 'check' failed  
make: *** [check] Error 1  
sangna@16343065sang: ~/pintos/src/threads/build$
```

4. 实验感想

本次实验按理来说应该比较简单，但是却花费了不少时间，原因在于我坚持使用 WSL 而不是 VMware。并不是因为固执，而是有其他特别的原因。凭记忆列举以下实验中曾遇到过的一些问题。

1. Make check

① 错误提示：Run didn't start up properly: **no "pintos booting" message**

结果就是 27/27 failed。

按照 installation_document 中的错误样例重新配置，依然解决不了。

最后放弃 Bochs，下载 qemu。（希望以后的实验中不会因为使用了 qemu 而造成不必要的错误...）

换成 qemu 后，

② 错误提示：run: **TIMEOUT** after 61 seconds of wall-clock time - load average: 0.10, 0.34, 0.2

结果又是 27/27 failed

在 StackOverflow 上找到了解决办法

Apparently, QEMU no longer supports the power off sequence on the port 0x8900. Here is a fix that made it work for me (found in [chaOs](#)): in the file pintos/src/devices/shutdown.c patch shutdown_power_off as follows:

```
void
shutdown_power_off (void)
{
    // ...
    printf ("Powering off...\n");
    serial_flush ();
    outw (0xB004, 0x2000); // <-- Add this line
    // ...
}
```

2. Apt 下载的时候有些包缺失

解决办法：更改 apt 源为中科大源。具体步骤已在实验过程中写出。

5. 源码分析

（1）thread.h

线程的结构定义在 thread.h 中，下面的代码中，绿色的中文注释是我结合 CS140 课程主页上的介绍和源代码中的注释添加的自己的理解。

```
enum thread_status      /* 线程的状态 */
{
    THREAD_RUNNING,      /* 正在运行的线程，每个特定的时间只有一个线程处于运行状态，
thread_current() 返回正在运行的线程 */
    THREAD_READY,        /* 就绪状态，处于这个状态的线程被存放在名为 ready_list 的双向链表中
*/
    THREAD_BLOCKED,      /* 阻塞状态，等待某一事件的触发，通过 thread_unblock() 可以进入就绪
状态，等待被安排运行 */
    THREAD_DYING         /* 将要死亡，当 CPU 转向下一个进程的时候，这个线程将会被毁灭 */
};
```

```
struct thread
{
    /* Owned by thread.c. */
    tid_t tid;                /* 线程的序号，唯一标记一个线程，默认从 1 开始 */
    enum thread_status status; /* 线程的状态：运行/就绪/阻塞/死亡 */
    char name[16];            /* 线程的名字 */
    uint8_t *stack;           /* 线程的栈指针，当 CPU 转向其他进程的时候，用于保存
当前的状态 */
    int priority;             /* 优先级，从低到高为 0~63 */
    struct list_elem allelem; /* 全部线程列表，线程被创建时插入列表，线程退出时被
移出列表 */

    /* Shared between thread.c and synch.c. */
    struct list_elem elem;    /* 某类(ready / waiting)线程表。 */
    // 这里的理解不太确定，没有看懂

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

    /* Owned by thread.c. */
    unsigned magic;            /* 检查是否栈溢出 */
};
```

/*每个线程结构都存储在它自己的 4 kB 页内。线程结构本身从下往上(从 0 开始)，剩余的空间预留给内核栈，内核栈是从上往下存储的(从 4 kB 开始)。

```
4 kB +-----+
      |          kernel stack          |
      |          |                      |
      |          |                      |
      |          |                      |
```

```

|           V           |
|       grows downward |
|                       |
|                       |
|                       |
|                       |
|                       |
|                       |
|                       |
|-----+-----+
|           magic       |
|           :           |
|           :           |
|           name        |
|           status      |
|-----+-----+
0 kB

```

这样，无论是线程结构或者内核栈都不会被允许增长得太大

(2) thread.c

The screenshot shows a code editor with two panes. The left pane displays the source code of `thread.c`, and the right pane shows a 'Function Table' (函数功能表) for the same file.

Source Code (thread.c):

```

34  /* Initial thread, the t
35  static struct thread *in
36
37  /* Lock used by allocate
38  static struct lock tid_l
39
40  /* Stack frame for kerne
41  struct kernel_thread_fra
42  {
43      void *eip;
44      thread_func *functio
45      void *aux;
46  };
47
48  /* Statistics. */
49  static long long idle_ti
50  static long long kernel_
51  static long long user_ti
52
53  /* Scheduling. */
54  #define TIME_SLICE 4
55  static unsigned thread_t

```

Function Table (线程功能表):

- thread_init (void)
- thread_start (void)
- thread_tick (void)
- thread_print_stats (void)
- thread_create (const char *name, int priority, void *aux)
- thread_block (void)
- thread_unblock (struct thread *t)
- thread_name (void)
- thread_current (void)
- thread_tid (void)
- thread_exit (void)
- thread_yield (void)
- thread_foreach (thread_action_func *func, void *aux)
- thread_set_priority (int new_priority)
- thread_get_priority (void)
- thread_set_nice (int nice UNUSED)
- thread_get_nice (void)
- thread_get_load_avg (void)
- thread_get_recent_cpu (void)
- idle (void *idle_started UNUSED)
- kernel_thread (thread_func *function, void *aux)
- running_thread (void)
- is_thread (struct thread *t)
- init_thread (struct thread *t, const char *name, int pri)

可以看到 `thread.c` 包含了相当多的 `tread` 操作函数。

下面只分析其中部分函数

void thread_tick (void) 函数

```
void
thread_tick (void)
{
    struct thread *t = thread_current (); /* thread_current () 返回正在运行的线程 */

    /* 更新计时*/
    if (t == idle_thread) /*Idle thread. 当没有其他线程处于就绪运行的时候执行*/
        idle_ticks++;
#ifdef USERPROG
    else if (t->pagedir != NULL)
        user_ticks++;
#endif
    else
        kernel_ticks++;

    /* 到达一定时间，实施抢占 */
    if (++thread_ticks >= TIME_SLICE)
        intr_yield_on_return ();
}
```

void thread_unblock (struct thread *t) 函数

```
/* 这个函数是使处于阻塞状态的线程进入就绪状态，如果对一个不是处于阻塞状态的进程使用这个函数就会
出错（可以用 thread_yield() 将线程从运行转为就绪） */
void
thread_unblock (struct thread *t) {
    enum intr_level old_level; // 记录是否中断状态
    /*
    这个定义在"interrupt.h"中，源代码如下
    enum intr_level
    {
        INTR_OFF,          /* Interrupts disabled. */
        INTR_ON            /* Interrupts enabled. */
    };*/
    ASSERT (is_thread (t)); // 确认是线程
    old_level = intr_disable (); // 关闭中断
    ASSERT (t->status == THREAD_BLOCKED); // 确认当前线程是处于阻塞状态
    list_push_back (&ready_list, &t->elem); // 将这个线程放入 ready_list 的末尾
    t->status = THREAD_READY; // 更新线程的状态为就绪
    intr_set_level (old_level); // 恢复之前中断状态
}
```