OS 实践项目 1 设计与实现文档

4p 小组
10 年 11 月 11 日

内容目录

1 小组人员组成与分工

1.1 小组成员

4p 小组由 4 名 08cs 成员组成，列表如下

陈歌，C，<olivia200705@126.com>

李梦阳，L，<mayli.he@gmail.com>

王盛，W，<wstnap@gmail.com>

肖天骏，X，<xiaotj1990327@gmail.com>

开发代号是每人姓氏的首字母

1.2 任务分工

任务分工由一系列表格组成，并且有较强的松散自由度。

日期	内容	人员	结果
11.3	实验环境搭建	**CLWX**	0
11.5	阅读源码，理解调用关系	**CLWX**	0
11.7	写完第一题代码	**CW**	延期 11.9
11.9	部署版本控制	**L**	完成
11.10	完成代码提交和检查	**WL**	完成
11.13	完成第二题代码	**L**	延期 11.15
11.16	完成第三题代码	**X**	
11.17	合并代码，提交版本	**LWX**	
11.18	文档清理上传	**CLWX**	
11.19	文档合并	**CL**	

1.3 版本控制

由于多人同时贡献代码，所以决定在项目中采用版本控制。

代码托管于 googlecode,采用 SVN 进行控制，可以在以下页面找到项目信息:

http://pintosof4p.googlecode.com

2 额外的参考资料

在作业完成的过程中除了压缩包中的资料，还参考了
pintos 官方文档： http://www.scs.stanford.edu/10wics140/pintos
minix3 源码 ：http://www.minix3.org/source.html

3 睡眠和唤醒

这一部分主要由 LXW 构思，W 实现，L 做出测试提出修改意见。

3.1 需求分析

既有的线程挂起机制，通过在 timer_sleep()中执行 while()循环实现，当前时间若不满足挂起的时间要求，则调用 thread_yield()函数继续循环，如果满足则直接压紧就绪队列。实际上只存在两个态，Running 和 Ready,并没有真正意义上的睡眠与唤醒。如图：

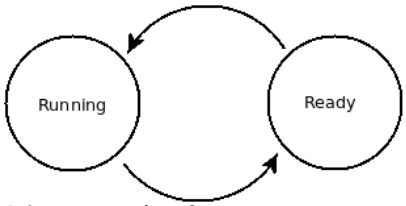


插图 *1:* 睡眠与唤醒

由于在 while()循环中，不断的进行 thread_yield()操作。通过查看源代码中 thread_yield()函数的注释：

```
/* Yields the CPU.  The current thread is not put to sleep and
 may be scheduled again immediately at the scheduler's whim. */
```

这也验证上面所说的线程并没有真正进入睡眠，且该线程有可能又一次立即被调度，这样的结果就是产生了忙等待。

我们需要实现的目标为，在 timer_sleep()函数中使线程进入 Block 态。系统运行一段时间后，睡眠时间到，再对该线程进行唤醒，从 Block 态转入 Ready 态。改造后的机制为如图：

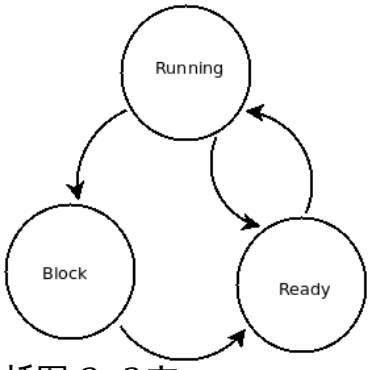


插图 *2: 3* 态

3.2 数据结构

虽然在 thread.h 文件中的 **enum** thread_statues{}已经提供了 thread_block 态 ，而且已经存在 ready_list 的链表,但中在源码中，并不存在一个叫做 block_list 的 list 结构。

故添加了数据结构：

```
static struct list block_list;
```

用于存储处于 block 态的 thread。

有了 block_list 就可以用来存储处于 Block 态的线程。而睡眠是有时间限制的，当睡眠时间到，之后则应进行唤醒操作。但在 struct thread{}结构中，没有相关的变量来记录睡眠的时间。
故在 struct thread{}进行改造，添加：
int64_t wakeup_ticks;
用以记录执行唤醒操作的时间，当前时间大于某个线程的 wakeup_ticks 时,应对线程唤醒。

3.3 算法设计

现有的 timer_sleep()函数中，在 while(){}循环中直接调用了 thread_yield()，要避免忙等待，需使线程进入睡眠，睡眠之后紧跟的问题便是，如何来唤醒 thread,更进一步的是何时来唤醒。

仔细分析知：

进入睡眠应该是在 timer_sleep()中实现，而进行唤醒的时间乍看之下似乎存在着不止一种选择
1.在每个 tick 之后，检查 block_list,当前时间大于 wakeup_ticks 则进行唤醒
2.在有某个线程要释放 cpu 时进行唤醒。

若在每个 tick 之后都进行唤醒检查 block_list，有可能出现所有的处于 Block 态的线程睡眠都没有结束的情况，即使某些线程睡眠结束，进行了唤醒，若不进行调度，唤醒后的线程也没有机会转入 Running 态，这样的效率过低。所以我们选择在进行线程调度之前检查 block_list,进行唤醒操作。

在具体实现中，thread.c 中
thread_sleep()函数使线程进入 Block 态，
thread_wakeup()函数检查 block_list，进行唤醒

归纳如下：

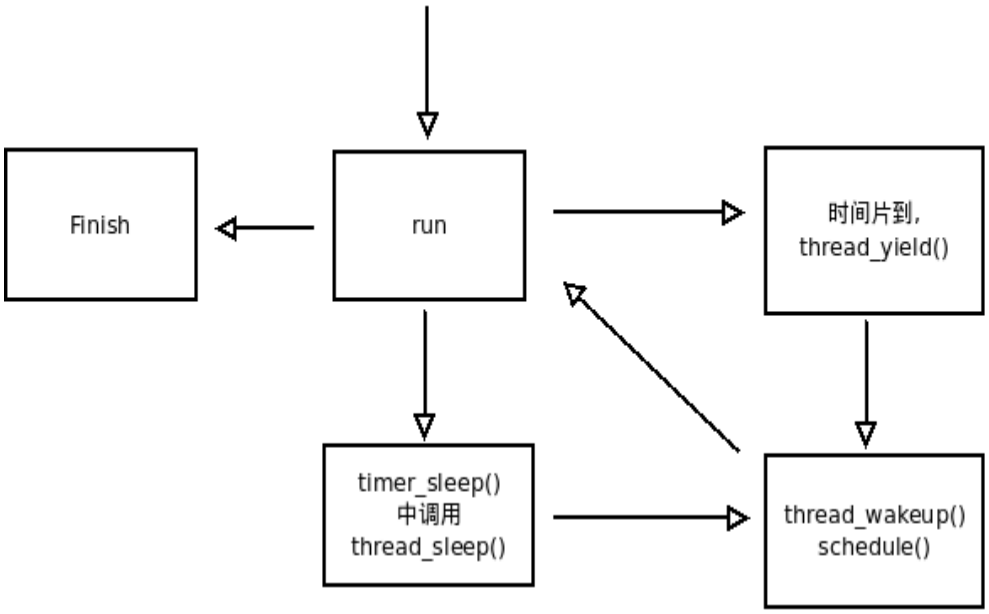
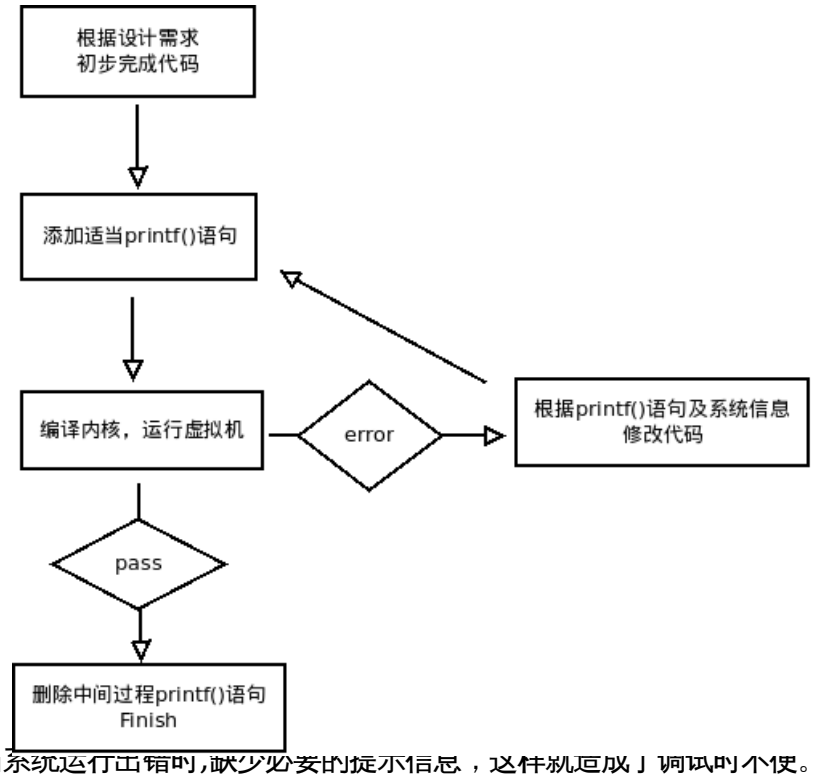


插图 *3:* 流程

3.4 调试



在代码执行的过程中，当系统运行出错时，缺少必要的提示信息，这样就造成了调试的不便。为了进行调试同时也为了弄清代码的执行过程，在调试阶段我们加入相当多的 printf()语句。整个过程的具体步骤如图所示：

```
------------------Sleep for approximately----------------------------------
[the sleep ticks is}:105
[now time_ticks() returns :}89

[DUMP:the block_list]
        thread 0's block_ticks is:105
[DUMP:the ready_list]
        thread 3's block_ticks is:0
        main's block_ticks is:0
        thread 4's block_ticks is:0
        thread 1's block_ticks is:0
        thread 2's block_ticks is:0

------------------Sleep for approximately----------------------------------
```

插图 *5: printf1*

```
------------------Sleep for approximately----------------------------------
[the sleep ticks is}:123
[now time_ticks() returns :}113

[DUMP:the block_list]
        thread 0's block_ticks is:105
        thread 3's block_ticks is:123
[DUMP:the ready_list]
        thread 1's block_ticks is:0
        thread 2's block_ticks is:0
        main's block_ticks is:0
        thread 4's block_ticks is:0

------------------Sleep for approximately----------------------------------
```

调试中遇到

插图 *6: printf2*

的问题

在 thread_sleep()中实现对线程的阻塞后，继续进行检查 block_list，然后调用 schedule()进行线程调度。而忽略了一点，并不是只有在 thread_sleep()才调用 schedule()进行调度。在 thread_yield()中调用了 schedule()函数，而我们没有在 thread_yield()函数中后检查 block_list，进行唤醒。问题的解决就是在 thread_yield()中添加检查 block_list,进行唤醒的代码。

对 **idle_thread** 的理解

```
/*  The idle thread is initially put on the ready list by
thread_start().  It will be scheduled once initially, at which
point it initializes idle_thread, "up"s the semaphore passed
to it to enable thread_start() to continue, and immediately
blocks.  After that, the idle thread never appears in the
ready list.  It is returned by next_thread_to_run() as a
special case when the ready list is empty. */
```

idle thread 只是在最初进入一次 ready_list,以后 idle thread 并不会出现在 ready_list 中。起初把它当作普通线程对待。在 thread.c 文件中 thread_tick()中，如果当前线程为 idle thread,执行的操作仅仅为 idle_ticks++,错误的对它进行改写为：idle_ticks>wakeup_ticks 时，执行唤醒。事实上 idle thread 在 thread_start()之后根本不会进入 ready list，自然也不会有睡眠与唤醒这样的操作。

3.5 优化

从最初正确可运行的代码到最终成型的代码，共进行了两点优化

起初采用了 thread_block_start 与 thread_block_ticks 分别记录开始进入 block 态的时间，和 thread 处于 block 的时间，当时间运行到大于等于 thread_block_start+thread_block_ticks 即应该进行唤醒。所以此问题，可以直接简化为用 wakeup_tick 解决，减少一个变量。

起初在每次 thread 从 Running 态进入 Block 态的时候，直接用执行入队列操作，插入到 block_list 的最后位置，而当唤醒的时候，每次又都要遍历 block_list，这样操作时间开销要大很多。若 block_list 中的是按 wakeup_ticks()进行排列的，那么在唤醒操作的时候，则不需要每次都进行遍历，当检查到某一线程的 wakeup_tick 不满足唤醒条件时，则不继续对后续 Block 态的线程进行检查，进行从而达到减少时间开销的目的。

3.6 题目完成程度

pass tests/threads/mlfqs-block

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

*这里的 FAIL 是因为这里代码只有睡眠唤醒机制，带有优先级的唤醒机制将在第二题里实现。

4 优先级调度

这一部分主要由 L 构思，L 实现，X 测试并提出修改意见。

4.1 需求分析

优先级调度要求实现一个基于优先级的调度策略。包括解决优先级授予和其他授予时的相关问题。

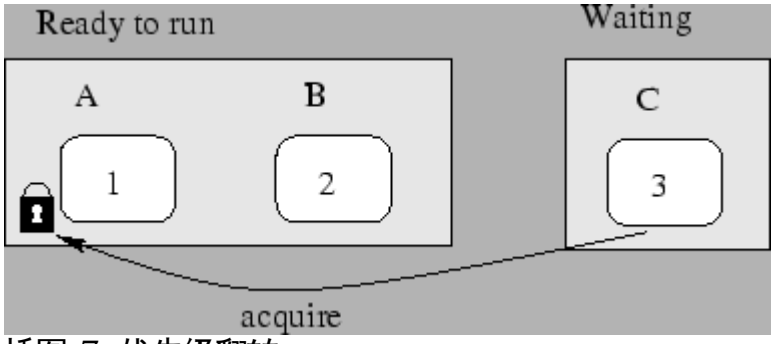


插图 7: 优先级翻转

优先级翻转描述如下：

线程 A,B,C 分别具有 1,2,3 的优先级(数字越大说明优先级越高),线程 A,B 目前在就绪队列中等待调度,线程 A 对一个互斥资源拥有线程锁。而此时,高优先级的线程 C 也想要访问这个互斥资源,线程 C 只好在这个资源上等待,不能进入就绪队列。当调度器开始调度时,它只能从 A 和 B 中进行选择,根据优先级调度原理,线程 B 将会首先运行。

这时就产生了一个问题,即本来线程 C 的优先级比线程 B 高,但是线程 B 却先运行了,从而产生了优先级翻转问题。

4.2 数据结构

Synch.h 中添加/修改

```
/* 锁本身，在原来锁的基础上添加了一个 id，这个值用来区别各个锁，类似于 tid */
struct lock
  {
    /* L: add an lid to identifer the lock */
    int lid;
    struct thread *holder;      /* Thread holding lock (for debugging). */
    struct semaphore semaphore; /* Binary semaphore controlling access. */
  };

/* 锁链表的一个元素，这个链表保存着所有的可能引发 donate 的锁 */
struct lock_elem
  {
    /* L:lock id */
    int lid;
    struct lock *lock;          /* This lock */
    struct list_elem elem;      /* List element */
```

```
  };

/* 锁列表的变量 */
struct list lock_list;
```

threah.h 中添加/修改

```
struct thread
  {
    ......
    int priority;                   /* Priority. */
    int priority_old;                 /* L: old priority stores the pri before donated */
    struct list_elem allelem;           /* List element for all threads list. */
    ......
    /* Owned by thread.c. */
    unsigned magic;                   /* Detects stack overflow. */
  };
```

4.3 算法

上面的数据结构依赖于以下的函数实现，这里没有给出他们的代码，不过后面会给出他们的流程，具体的细节请参考源码：

thread.c

```
/* 这两个辅助函数用来辅助线程列表中按优先级顺序的操作 */
list_less_func priority_higher;
list_less_func priority_lower;
/* 这俩辅助函数用来辅助 synch 里 lock_list 和 cond 的 list 的优先级顺序排序 */
list_less_func lock_lower;
list_less_func cond_lower;
/* 这个函数保持了某线程按优先级有序进入 ready_list ，alarm-priority 相关 */
void thread_unblock (struct thread *t) ;
void thread_yield (void) ;
/* 这个函数只修改 priority_old，priority-change 相关 */
void thread_set_priority (int new_priority);
/* 这个函数检查 lock 是否有更高优先级的线程在等待 */
bool donate_check (void);
/* 这个函数把 waiters 的最高优先级给锁的 holder，即 donate，priority-donate-*相关 */
void donate_do (void);
/* 会在每次安排调度时调用上面两个函数 */
static void schedule (void);
```

synch.c

```
/* 取锁函数，并且如果存在可以 donate 的情形，优先级授予，然后被阻塞，进入等待列表 */
lock_acquire (struct lock *lock);
/* 放锁函数，并且恢复优先级，这个并不会再次授予 */
lock_release (struct lock *lock) ;
/* 如果无法 down，就把线程按优先级插入到等待者列表，高优先级在链表尾 */
sema_down (struct semaphore *sema) ;
/* 升起信号量，并且 unblock 等待者最大优先级的线程 */
sema_up (struct semaphore *sema);
/* cond 的唤醒，这里唤醒等待列表中优先级最高的，priority-condvar 相关 */
cond_signal (struct condition *cond, struct lock *lock UNUSED);
```

4.4 解法描述

简介：

这里只使用了一个比较简单的数据结构(变量 lock_list)来实现优先级的各个操作，这个实现的解法思路比较简单，用一种方法解决所有的 test case，实现起来不复杂，缺陷也是比较明显的，它没有逐个解决 test_case 的方法的效率高。尤其是在存在许多锁却没发生优先级授予的时候，一定量的计算能力被消耗掉了。不过这个解法依然是有效的健壮的。采用这个实现方法的原因在于它的确很好的符合了 KISS 原则。

整体实现描述：

所有 priority 相关的要求分为以下三组：

睡眠唤醒的 priority:alarm-priority

优先级授予的:priority-donate-*

优先级调度的公平合理性:priority-[sema|fifo|condvar]

第一组的实现，只需要控制一个有序的 ready_list 队列即可。这里采用 wake_up 和 thread_yield 对进队列进行控制，通过 next_thread_to_run 对出队列进行控制。

第二组的实现，需要一整套锁系统来解决。其中包括的功能性要求有优先级授予、优先级收回、被授予的优先级的线程试图改变自己优先级的情形、复杂情形下优先级 multiple/chain 式授予与收回。

这组的实现以一个以锁表为核心的数据结构/算法来提供。任何用户申请的锁都会被在这里记下，当另一个线程申请这个锁时，检测它的优先级，判断是否进行授予，然后放入 waiters 队列被阻塞。优先级是和调度相关的，所以在调度时进行 donate 检查与授予，这里对所有发生优先级授予的锁都产生一个有高到底的优先级授予工作，这保证了 chain 式授予的正确性，同时 multiple 的情形在这里也被隐式的解决了。donate-check 和 donate-do 是这个解法的核心，优先级授予操作也都是由这俩函数实现的。

第三组的实现，只需要对 sema/ready_list/cond 的操作时严格的高优先级和先进先出即可，实现上并不复杂，这里不再赘述，具体请参考以下函数 sema_up/thread_unblock/cond_signal。

主要函数的工作流程

一个典型的 lock_acquire()和 sema_down()过程

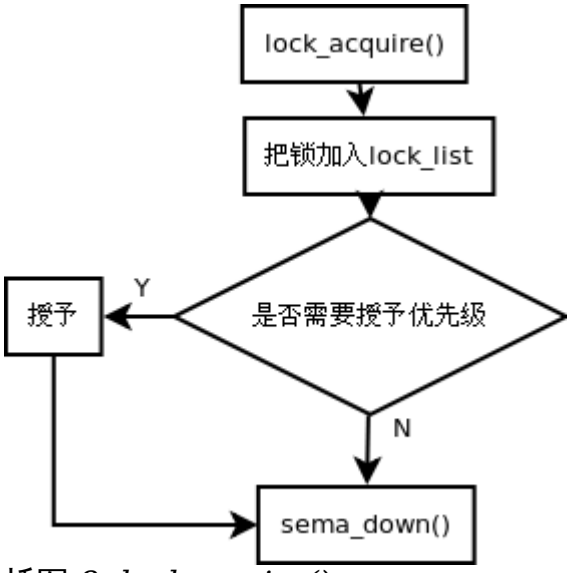


插图 *8: lock_qcuire()*

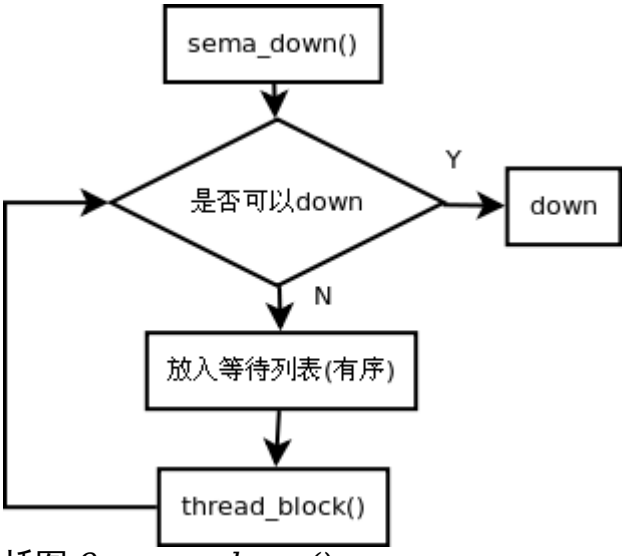


插图 *9: sema_down()*

一个典型的 lock_release()和 sema_up()过程

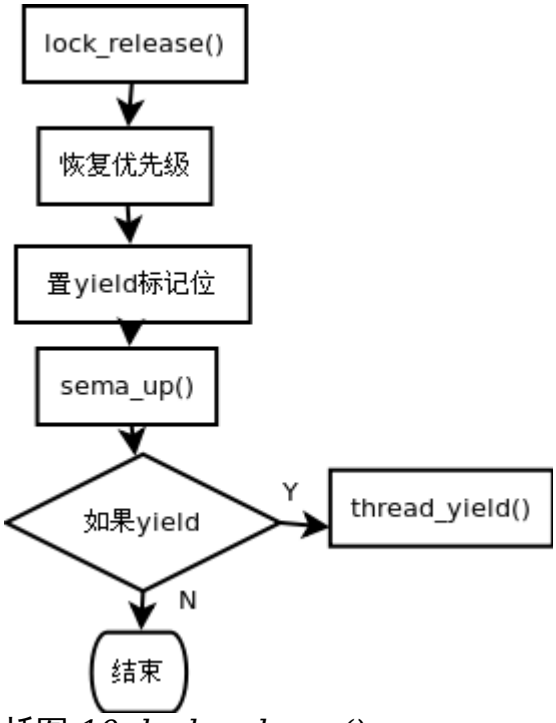


插图 *10: lock_release()*

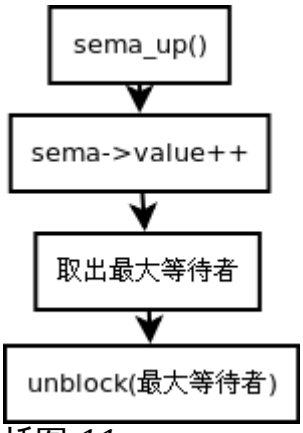


插图 *11: sema_up()*

处理 donate 的两个函数

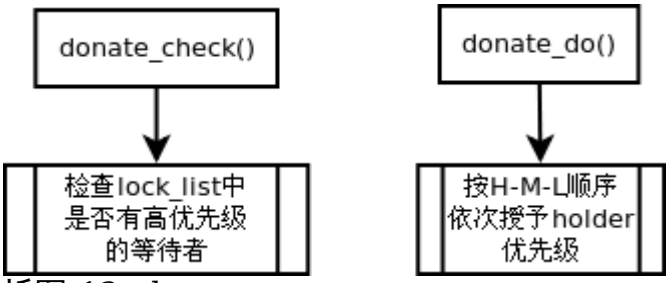


插图 *12: donate*

一个典型的 **testcase**：**priority_donate_nest**

下面的表格展示了从 main 取得第一个锁到程序结束的流程

	Main	*medium*	*high*
1	取锁 a		
2	创建线程 medium		
3		取锁 b	
4		尝试取锁 a，被阻塞，main->32	
5	让		
6	创建线程 high		
7			尝试取锁 b，被阻塞， medium->33，main->33
8	让		
9	释放 a，还原到 31		
10		释放 a，让，释放 b，还原到 32	
11			获得 b，释放 b，退出
12		退出	
13	退出		

需要明确的几个问题：

如何保证等待某个锁的最高优先级线程最先被唤醒？

我们使用一个有序的 waiters 列表保存所有被阻塞的线程，并且在唤醒时始终唤醒最高优先级的。

当 lock_acquire 被调用时发生了什么？

如果锁被锁住，放到 waiters 列表中，该线程就会被阻塞，在 thread_block 中有相关的阻塞后的处理。

当一个高优先级等的的锁被 lock_release 时发生了什么？

发生的操作是先解锁，然后把 waiters 列表中最高优先级的 thread_unblock，并且判断是否需要 thread_yield 来让 cpu 给更高的线程。

另外一种解决方法：

另一种优化的解法是只在 lock_acquire 和 lock_release 时进行优先级授予和收回操作，这需要在 thread 结构里提供一个锁链表，抛弃了共用一个锁表的数据结构，有一定的效率提升。而且当某个线程崩溃时，等待该锁的所有线程将丢失。

不过这样做的坏处是使得流程上有了两套控制体系，一是 thread 的 schedule，二是 synch 的 lock 系统，我认为在一个系统里如果有两套控制体系，应该不是一个好的基础设计，所以实际使用的解法是在 schedule 里实现几乎所有操作，而尽量少的去在 synch 里实现他们，使 schedule 成为系统的核心，同时也很符合 KISS 原则。

4.5 调试过程

由于本组测试条目较多，这里只对典型的测试样例说明

priority-change
priority-preempt
这组测试在调试中没有太大问题，第一个引发了必须添加 priority_old 的思考。

priority-fifo
priority-sema
priority-condvar
这组测试都是体现唤醒时的顺序：1 严格优先级 2 轮转，在调试时同样没有出现严重问题。

priority-donate-one
priority-donate-multiple
priority-donate-multiple2
priority-donate-nest
priority-donate-chain
priority-donate-sema
priority-donate-lower
这组测试都是 donate 相关的测试，其中比较典型的是 nest、multiple2，由于解法的思路比较清晰，所以实际代码编写中没有太多技术性内容，只是有几个疏忽导致 multiple2 发生了奇怪的现象。在 X 的帮助下修改了代码，并且对一些细节做了微调。

4.6 题目完成程度

pass tests/threads/priority-change

pass tests/threads/priority-donate-one

pass tests/threads/priority-donate-multiple

pass tests/threads/priority-donate-multiple2

pass tests/threads/priority-donate-nest

pass tests/threads/priority-donate-sema

pass tests/threads/priority-donate-lower

pass tests/threads/priority-fifo

pass tests/threads/priority-preempt

pass tests/threads/priority-sema

pass tests/threads/priority-condvar

pass tests/threads/priority-donate-chain

注：在默认模式下有时候会出现 no "Powering off" message 的错误，我们确信这是由于 Bochs 在这种模式下的问题，在 make check 测试时可能出现这种情况，在“pintos -v – -q run priority-”的输出结果和测试脚本验证一致。

5 高级调度

这部分由 X 构思，X 实现，C 测试。

5.1 需求分析

普通线程的调度策略是基于时间片轮转的优先级调度，总体来说，实现简单方便，体现了严格按优先级运行的严谨。

这种调度策略有一定的优点，不过，在某些情况下，这种策略存在一定弊端。这种调度策略极易造成线程的饥饿，当有高优先级且所需运行时间片较长的线程被创建，其余的线程只有在它行完之后才能重新获得 CPU，这种调度策略没有考虑到系统的吞吐量，无法减少线程的平均等待时间。这种弊端在大用户量的交互系统中会被放大，造成用户长时间的等待会降低用户对于该系统的评价。
因此，我们需要一种新的调度策略，该策略在考虑线程优先级的基础之上，能够按照线程运行情况来动态地考虑修改线程的优先级，于是根据执行情况反馈信息而对线程队列进行组织并调度线程的调度器多级反馈队列调度器应运而生。

5.2 设计思想

多就绪队列：
根据优先级个数声明 64 个队列，CPU 在调度时总是从最高优先级队列找起，发现第一个不为空的队列，便根据先进先出的原则，从队列的出口处取出线程，分给时间片进行运转，如图所示。

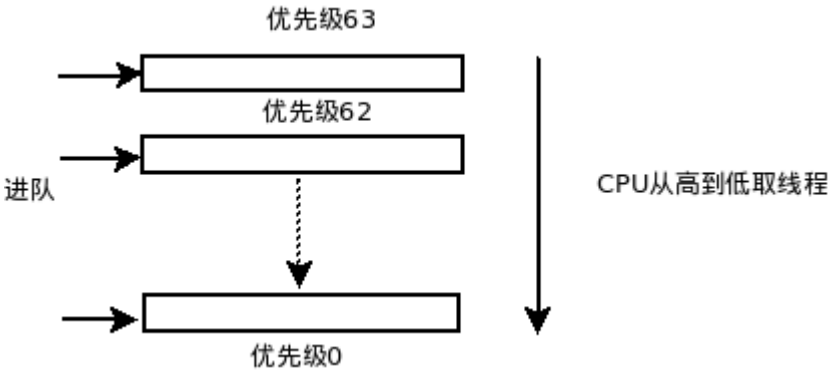


插图 *13:* 调度

记录线程运行情况：
我们设计相应的数据结构，来记录一个线程的近期占用 CPU 状况，该线程对于其他线程的友好程度，

近期 CPU 的负载等运行情况，以此作为依据，调整线程的优先级。

通过记录线程运行的信息修改线程优先级
修改线程优先级的宗旨是防止线程过度饥饿，适当降低近期多占 CPU 的线程的优先级，适当提高近期较少占用 CPU 的线程的优先级，达到一种动态平衡。

5.3 数据结构

1.nice：
基本信息：thread 结构体的成员变量，记录该线程对于其它线程的友好程度。通俗的描述就是表示该线程是否愿意通过设置较高的 nice 值降低自己的优先级，供其它线程运行更多的时间片。
数据类型：整形
数值范围：-20～20，该值为正表示愿意降低优先级，为负表示希望增加优先级。
缺省值：0
涉及的函数：
```
int thread_get_nice (void)
void thread_set_nice (int new_nice)
void renew_priority(struct thread*t)
```

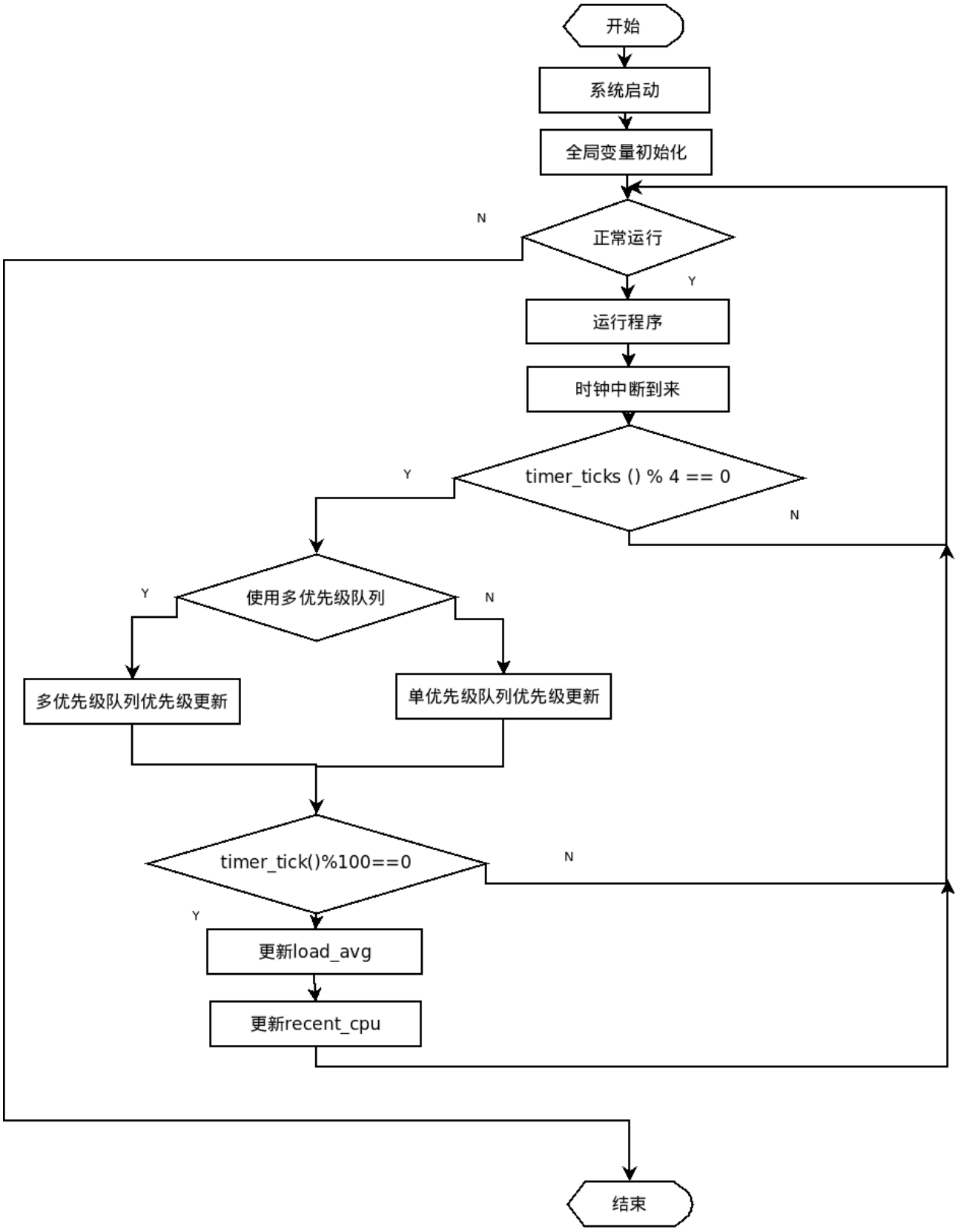
2.recent_cpu:
基本信息：thread 结构体的成员变量，记录该线程近期占用 CPU 情况，近期占用 CPU 较多则应该在计算优先级时考虑降低该线程的优先级。
数据类型：定点型实数
数值范围：-(2^31-1)/2^14~(2^31-1)/2^14
缺省值：0
涉及的函数
```
int thread_get_recent_cpu (void)
void renew_recent_cpu(struct thread* t)
```

3.load_avg
基本信息：静态全局变量，用于记录系统平均负载，估计过去一分钟系统平均就绪线程数
数据类型：定点型实数
数值范围：-(2^31-1)/2^14~(2^31-1)/2^14
缺省值：0
涉及的函数：
```
int thread_get_load_avg (void)
void renew_load_avg(void)
void renew_recent_cpu(struct thread* t)
```

4.advanced_ready_list[64]
基本信息：全局变量，多级反馈队列，其下标对应于线程的优先级；
数据类型：struct list
说明：在反馈调度队列的实现过程中，我认为只需单就绪队列也能完成同样的功能，我参考了 pintos 项目的官方文档（http://www.scs.stanford.edu/10wics140/pintos），上面说明，只要运行表现相同，则可以使用单队列进行处理，我在程序编写的过程中对单队列和多对列都有尝试，关于单队列和多队列实现的比较我在后文中有所说明，最后保留在提交代码中的是单队列实现，再此特作说明。

5.4 算法设计与函数实现

整体算法流程：



具体程序实现：
1.int64_t get_ready_threads()

实现功能：得到当前运行态和就绪态的线程数。
方法简介：调用 list_size(&ready_list)，得到当前处于就绪态的线程数，再加上当前运行的一个线程，返回结果。
调用关系：被 renew_load_avg()函数调用。
中断状态：应该在关中断条件下被调用。

2.void renew_load_avg(void)
实现功能：更新 load_avg 的值。
方法简介：更新函数
load_avg= (59/60)*load_avg+(1/60)*ready_threads,
该式的意义在于利用原值乘上系数与当前值相加，能够保留上一秒钟的结果，并通过当前 ready_threads 进行更新。
调用关系：在 thread_tick()函数中被调用。
调用条件：timer_ticks () % TIMER_FREQ == 0
中断状态：应该在关中断条件下被调用。

3.void renew_recent_cpu(struct thread* t)
实现功能：更新线程 t 的 recent_cpu。
方法简介：更新函数
recent_cpu = (2*load_avg)/(2*load_avg + 1) * recent_cpu + nice
该算式为一个迭代式，使用了指数加权移动平均方法，通过迭代更新可以避免存储 cpu 近期运行线程表的空间开销，以及查询 cpu 近期运行线程表的时间开销，是一个优化的算法。
调用关系：被 thread_all_renew(void)调用。
中断状态：应该在关中断条件下被调用。

4.thread_all_renew(void)
实现功能：更新所有线程的 recent_cpu,包括正在运行的，就绪的，被阻塞的线程。
方法简介：遍历 all_list，寻访所有线程，调用 renew_recent_cpu 完成更新。
调用关系：在 thread_tick()里被调用。
调用条件：timer_ticks () % 4== 0
中断状态：应该在关中断条件下被调用。

5.void renew_priority(struct thread* t)
实现功能：更新线程 t 的 priority 值。
方法简介：更新函数
priority = PRI_MAX - (recent_cpu / 4) - (nice * 2)
通过检测近期该线程的 cpu 使用状况来更改优先级，如果近期用 cpu 较多，则下一阶段应该让前一阶段用 cpu 较少的线程运行，具体操作就是减去 recent_cpu 乘以系数，而线程本身有 nice 值，nice 值较高说明该线程愿意让出 cpu 让别的线程运行，具体做法依然是用 nice 值来改 priority 的值。
调用关系:在 renew_all_priority 函数中被调用。
中断状态：应该在关中断条件下被调用。

6.void renew_all_priority()
实现功能：更新所有线程的 priority，包括正在运行的，就绪的，被阻塞的线程。
方法简介：遍历 all_list，寻访所有线程，调用 priority 完成更新。接下来的操作，根据是否使用多优先级队列会产生不同，如果使用多优先级队列，则只需将线程放入更新后优先级对应的队列，如果使用单队列，则需要对 ready _list 进行以此排序，才能保证表头为优先级最大的线程。
调用关系：在 thread_tick()里被调用。
调用条件：timer_ticks () % 4== 0

中断状态：应该在关中断条件下被调用。

7.thread_tick()
功能实现:这一个函数中添加的代码对应于整体流程图中的时钟中断之后的部分，当时钟中断到来时，进行判断是否需要进行优先级，recent_cpu 与 load _avg 更新。

8.fixed-point.h
功能实现：定点实数运算宏定义。由于 pintos 本身没有浮点数类型，而 load_avg,recent_cpu 并不是整形，所以我们需要用整数来表示实数，采用的办法就是一个 int_32 型整数的低十四位默认为小数点后的位数，由于转换较为简单，采用宏可以较好地完成这个任务，其中涉及的运算有实数之间的加减乘除，实数与整数之间的互相转换，整数与实数之间的加减乘除，能够满足参数计算的需要。

5.5 调试过程

1.mlfqs-load-1
测试目的：测试程序测试的是 renew_load_avg()函数是否运行正常。
测试过程：直接 pass。

2.mlfqs-load-60
测试目的：测试程序测试的是 renew_load_avg()函数是否运行正常。
测试过程：pass，但是与标准结果有误差。
分析原因：系统正常启动之后计时便开始，此时测试线程还没有被创建，不过主线程已经在运行，所以在测试线程创建时 load_avg 已经不是 0.

3.mlfqs-load-avg
测试目的：测试复杂环境下 load_avg 计算是否正确
测试过程：多次 FAIL 后 pass.
分析原因：
1.后面的 priority 计算有错，导致程序并未按照原有预期运行；
2.bochs 在默认条件下运行时并没有严格按照准确时间运行，导致重开中断之后当前线程还没有运行完又被打断。
解决办法：改正错误，优化冗余代码。

4.mlfqs-fair-2/mlfqs-fair-20
测试目的：同优先级的线程在一段时间内被分得的时间片应该大致相同；
测试过程：pass

5.6 思考与优化

单优先级队列与多优先级队列实现反馈调度的比较。

单队列与多对列在实现时不同的操作体现在更新优先级以及调度这两个部分。

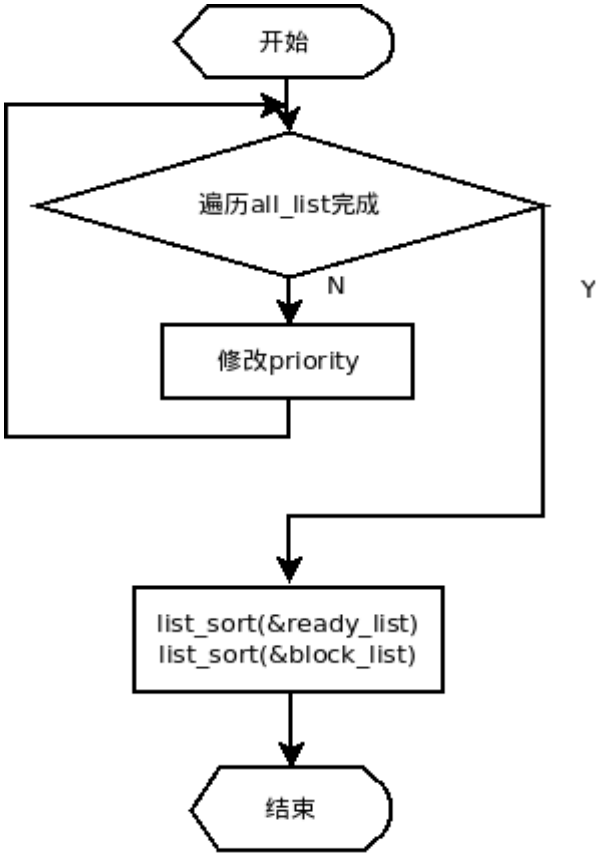


插图 *15:* 单队列更新

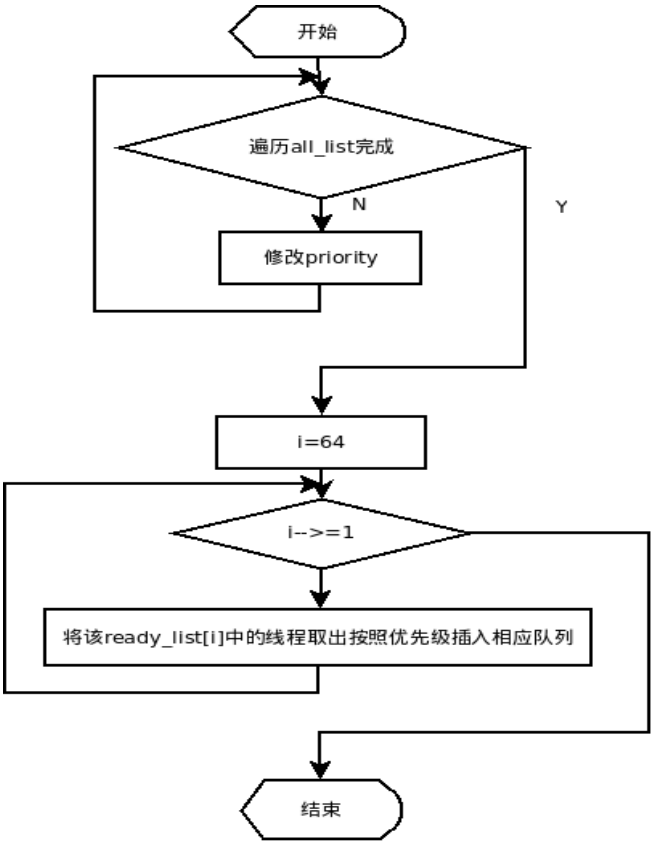


插图 *16:* 多对列更新

说明：priority 的更新都通过遍历 all_list 进行，单队列实现在更新完之后，为了保证队列保持单调，以保证每次访问 ready_list 与 block_list 时，出口都是优先级最大的线程，这一步操作的复杂度为 O(nlogn),而多队列实现时需要将 ready_list 中所有线程处理一遍，表面上看复杂度为 O(n),但是考虑到当线程数少于 64 时，其仍然要处理 64 次，其平均效率并不会提高多少。

在调度时，要取出下一个运行的线程，单队列时由于 ready_list 根据优先级排好序，只需弹出表头，复杂度为 O(1),而多对列实现，需要从优先级为 63 的队列找起，找到第一个不为空的队列取出队首元素，复杂度为 O(n),而插入时，单队列需要 O(n),多队列需要 O(1).

综上所述，在小型操作系统上(嵌入式操作系统，个人电脑操作系统)，两种实现方法表现应该差异不大，当运行在服务器上时，ready_list 里的线程可能达到上千的数量级，这时，用多队列来实现反馈调度机制较为优化。

5.7 题目完成程度

pass tests/threads/mlfqs-load-1
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
注：全部通过。

6 系统强健性测试

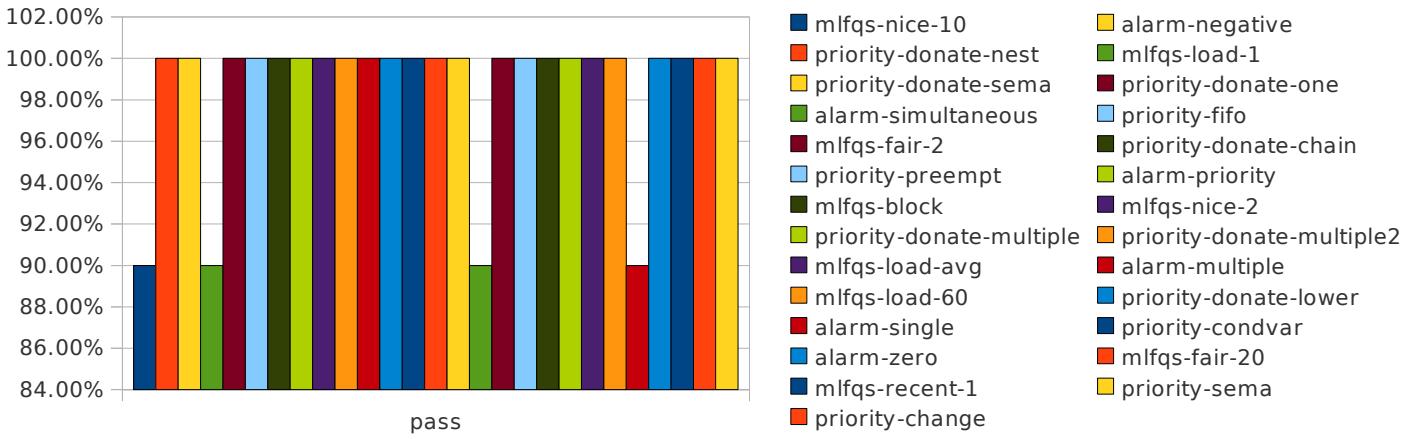
6.1 简介

由于操作系统的复杂性，我们不能保证它在任何情况下解决所有问题，所以我们进行了一些强健性测试，包括默认模式的 make check(reproducible)，使用随机长度时间中断(-j SEED)的 bochs，使用 realtiming 模式的 bochs(-r)，和一款常见的虚拟机 qemu 进行了分组测试，每个测试重复 10 次，并且统计了各个测试的 pass 的概率*。

*注：理论上在所有情况下应该都是正确的，不过由于暂时未知的原因我们遇到了”No PoweringOff Message”错误，并在所有的测试的 FAIL case 中均是这个错误，而不是其他的错误。

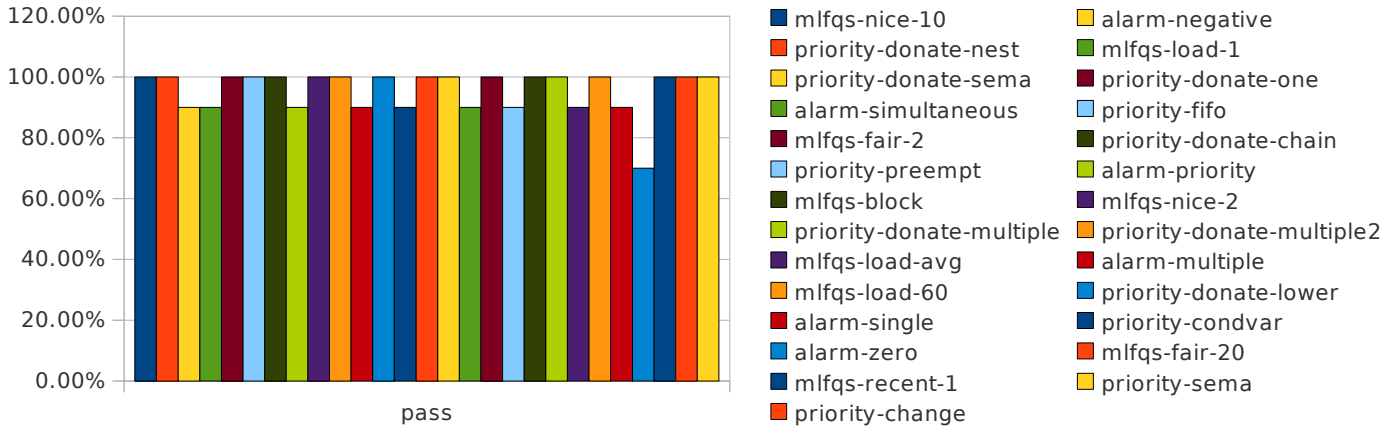
6.2 默认模式

使用'make check'进行了十次测试得到了如下的 pass 率



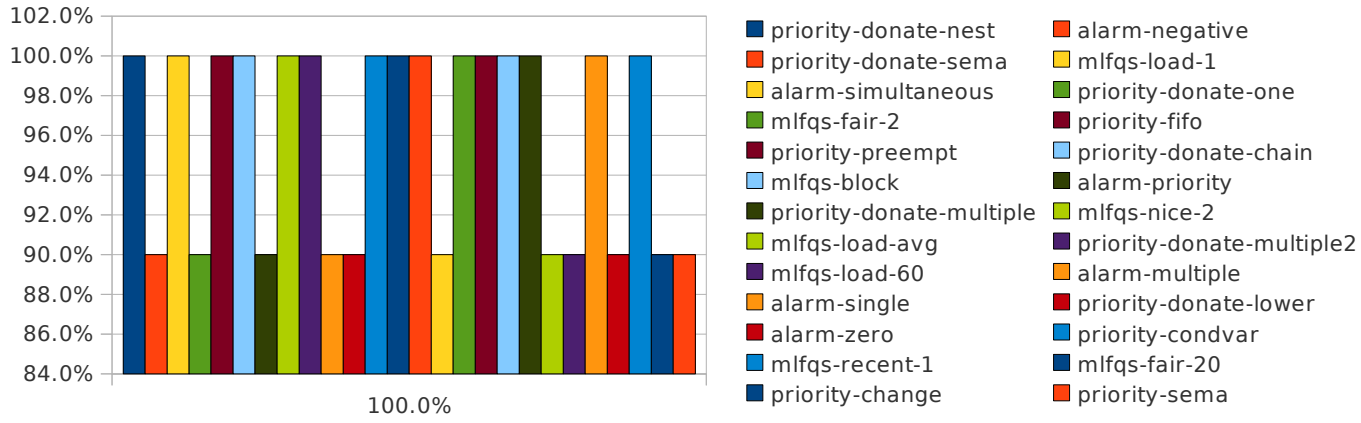
6.3 随机中断长度

使用'make check PINTOSOPIS=-j %d'%random 得到如下的 pass 率



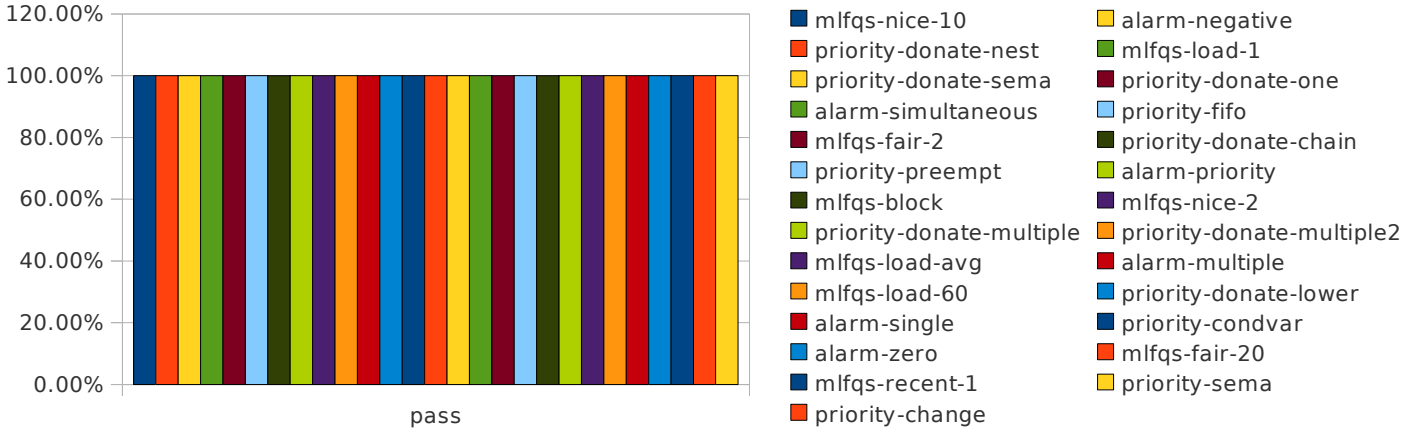
6.4 实时模式

使用'make check PINTOSOPIS=-r'得到的 pass 率



6.5 使用 *qemu*

使用'make check SIMULATOR=--qemu'得到的结果



6.6 小结

由以上图表得出我们的 pintos 在 bochs 上还不是可以 100%的 pass 每个 test-case，我们得到的所有 FAIL 全部都是 NoPoweringOff Message，而不是其他错误。我们认为解题思路是正确的，只不过还是存在有些我们没有解决的问题。不过对于一个比较成熟的虚拟机 qemu 的测试验证了我们的想法，我们通过了所有(10 次)的测试，由此得出我们的 pintos 实践项目 1 是比较成功的。