**Project1**

周 仪 10132510114

小组成员——

董思远10132510115，

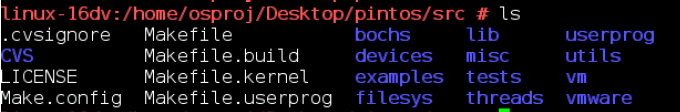
茅凯博10132510116，

胡诗璐10132510113，

**Part One**

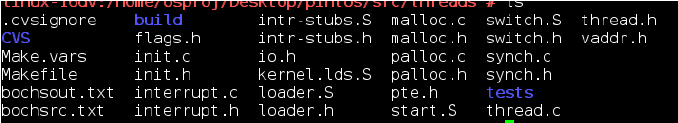
**该部分由小组成员一起讨论整理，主要是阅读源码，了解pintos，和初步尝试修改。**

了解pintos目录树结构



|  |  |
| --- | --- |
| threads/ | 基本内核的源代码 |
| userprog/ | 用户程序载入器的源代码 |
| vm/ | 是个空的目录，用于实现虚拟内存 |
| filesys/ | 基本文件目录的源代码 |
| devices/ | IO设备接口的源代码：键盘、定时器、磁盘等 |
| Lib/ | 实现了标准的C语言库，这个目录当中的代码Pintos kernel一起编译。 |
| Test | 对每个程序进行测试。如果你觉得对你有用的话，你可以修改这个代码。但是这个文件在官方测试之前会被它的原件所取代 |
| examples/ | 一些例子 |

Thread 目录树



|  |  |
| --- | --- |
| loader.s | 内核加载器。这个代码不用看 |
| loader.h | |
| kernel.lds.S | 连接脚本，用来连接内核，设置负载地址的内核。不用看 |

|  |  |  |
| --- | --- | --- |
| start.S | | 跳到主函数 |
| init.c | | 内核初始化，包括main()函数。至少要看看main()做了哪些初始化，也可以在里面添加你自己的初始化代码 |
| init.h | | |
| thread.c | | 基本线程支持。大多数工作在这些文件里面完成。在本项目当中需要修改这里面的内容。 |
| thread.h | | |
| switch.S | | 汇编语言常规的用于交换的线程 |
| switch.h | | |
| palloc.c | | 页分配器，将系统内存分配成多个4Kb的页面 |
| palloc.h | | |
| malloc.c | 关于内存分配和内核空间的一个简单实现 | |
| malloc.h | | |
| interrupt.c | 基本中断处理和开关中断的方法 | |
| interrupt.h | | |
| intr-stubs.S | 低级中断处理的汇编代码 | |
| intr-stubs.h | | |
| synch.c | 基本同步原语：信号量、锁、条件变量和optimization barriers。 | |
| synch.h | | |
| io.h | I/O端口访问的方法。这个主要用在devices目录下，不用动 | |
| vaddr.h | 为了用虚拟地址和页表条目的方法和宏 | |
| pte.h | | |
| flags.h | 定义了一些32位机器的“Flag”的宏 | |

**1、Data structure**

**一.阅读源码**

在pintos源码中找到最基本的概念：

本次实验需要阅读init.c、thread.h、thread.c、switch.h、synch.h、synch.c 、time.h、time.c这些文件。

pintos在thread.h中定义了一个结构体struct thread，这个结构体就存了有关进程的基本信息。

struct thread

{

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. \*/

};

队列结构示意图：

空队列：

+------+ +------+

<---| head |<--->| tail |--->

+------+ +------+

队列中有两个元素：

+------+ +-------+ +-------+ +------+

<---| head |<--->| 1 |<--->| 2 |<--->| tail |<--->

+------+ +-------+ +-------+ +------+

这个线程的几个基本信息。值得注意的是enum thread\_status这个枚举类型的变量，他的意思就是这个线程现在所处的状态。

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. \*/

};

还有一个最最重要的概念是中断。所谓中断其实分两种，一种是IO设备向CPU发出的中断的信息，另一种是CPU决定切换到另一个进程时（轮换时间片）发出的指令。我们现在处理第二种。pintos的中断在interrupt.h和interrupt.c之中。其中这个枚举类型intr\_lverl会在后面被反复提到：

enum intr\_level

{

INTR\_OFF, /\* Interrupts disabled. \*/

INTR\_ON /\* Interrupts enabled. \*/

};

其实这个intr\_level表达的意思更简单，就是有两个单词，intr\_off表示关中断，on表示开中断。大家都知道，执行原子级别操作的时候，中断必须是关着的。

最后还要说以下，pintos是以ticks作为基本时间单位的，每秒有TIMER\_FREQ个ticks：

/\* Number of timer interrupts per second. \*/

#define TIMER\_FREQ 100 //系统默认这个宏为100

还有一点，pintos默认每一个ticks调用一次时间中断。换句话说，每一个线程最多可以占据CPU一个ticks的时长，之后就必须放手。

**二.掌握thread的基本操作（函数）：**

以下函数在thread.c中都可以找到。

1.thread\_current()获取当前的线程的指针。

2.thread\_foreach(thread\_action\_func \*func, void \*aux) 遍历当前ready queue中的所有线程，并且对于每一个线程执行一次func操作。注意到这里的func是一个任意给定函数的指针，参数aux则是你想要传给这个函数的参数。实际上pintos没有多么高深，所有ready的线程被保存在一个链表中。这个函数做得不过是遍历了一遍链表而已。注意这个函数只能在中断关

闭的时候调用。

3.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。

4.timer\_interrupt (struct intr\_frame \*args UNUSED)这个函数在timer.c中，pintos在每次时间中断时（即每一个时间单位（ticks））调用一次这个函数。

5. intr\_disable () 这个函数在interrupt.c中，作用是返回关中断，然后返回中断关闭前的状态。（其实说白了状态不就是INTR\_OFF,INTR\_ON 这两种么。 ）

**三、代码分析与timer\_sleep()函数的重新设计(红色为修改部分)：**

timer\_sleep的作用是让此线程等待ticks单位时长，然后再执行。函数原型：

void

timer\_sleep (int64\_t ticks) //参数的意思是你想要等待的时间长度

{

int64\_t start = timer\_ticks (); //获取当前的系统时间

ASSERT (intr\_get\_level () == INTR\_ON);

while (timer\_elapsed (start) < ticks) //如果elapse>=ticks时就返回。否则将持续占用cpu

thread\_yield ();

}

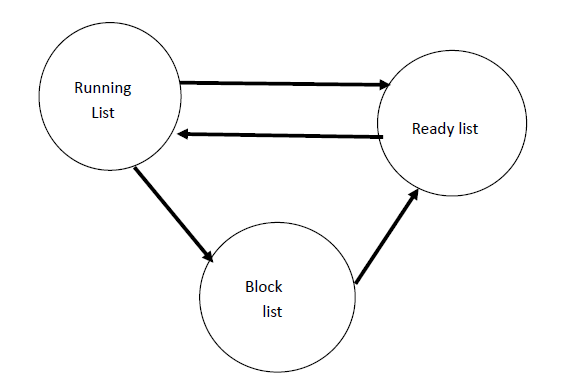
线程调度过程中的忙等待的原因主要产生在timer\_sleep()这个函数，函数使用的一个循环while (timer\_elapsed (start) < ticks)

thread\_yield ();

每次调用这个函数的时候，会对线程进行判断，判断线程需要睡眠的时间ticks是否已经到了。如果没有到就循环执行thread\_yield()这个函数，让出cpu。而且thread\_yield()函数只是把当前的thread压入ready队列，这样的话可能会因为优先级的问题，使得刚刚进入睡眠的线程可能会被立即调用。起不到使线程睡眠的作用。

原本的timer\_sleep函数是对的，只不过它不好。它使用的方法（忙等待）是利用一个while循环不断地请求CPU来判断是否经过了足够的时间长度。这样做得坏处是很显然的，通常cpu在一个ticks时间内可以处理10000次这样的循环，而timer\_elapsed()函数只会在ticks＋1时更新一次。所以我们可以做得就是改进这个函数，使得它每一个ticks才检查一次时间，而不是每一个ticks检查一万次。

由以上对需求的分析，可以设计出大致的实现思路。产生忙等待的原因是在timer\_sleep()中的循环判断时间。因此需要把时间的判断放在别的函数中。并且仅仅使用thread\_yield()无法使线程真正睡去。因此我们需要一个block队列，可以通过把线程移到block队列，从而起到一个阻塞的作用，使线程不能立即再被cpu调用。



我们的设计思路是：在timer\_sleep（）函数中让该进程暂时阻塞（调用thread\_block()），然后过了ticks个时间段后再把它加回到ready queue中。

至于因为每一次时间中断的时候恰好是ticks加一的时候，因此我们可以改进timer\_interrup()函数，使得系统每次调用他的时候都检查一下这个进程是否已经等

待了足够长得时间了。如果还没有够，则不管它，如果已经足够长了，则调用thread\_unblock()函数将它召唤回ready\_queue中。

这个时候又有一个问题出现了：怎么样才能得到一个线程被阻塞了多长时间呢？事实上pintos本身没有这样一个功能，需要我们对thread这个结构体添加，加入一个整形变量int block\_ticks就可以了。当这个线程被block的时候，将block\_ticks记录为需要等待的时间长度。之后每次中断的时候检查它一次，并且顺便使其自减。当它小到等于0的时候，把线程调到ready queue中。

**四.修改源码**

当然修改之前最好先备份一下源文件

1.将thread改为：

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. \*/

int block\_ticks\_time; /\* 标记flag 表示已经多久\*/

/\* 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. \*/

};

2.改动timer.c中的timer\_sleep函数

void

timer\_sleep (int64\_t ticks)

{

/\*

int64\_t start = timer\_ticks ();\*/

ASSERT (intr\_get\_level () == INTR\_ON);

/\*while (timer\_elapsed (start) < ticks)

thread\_yield ();\*/

enum intr\_level old\_level; //定义变量保存原先的中断状态

struct thread \*t;

t=thread\_current ();

t->block\_ticks = ticks; //将等待时长设置为需要等待的时间

old\_level = intr\_disable (); // thread\_block()方法需要关闭中断

thread\_block(); //阻塞进程

intr\_set\_level (old\_level); //恢复中断

}

3.为timer.c增加函数用来检查每个进程中的block\_ticks是否为零

void

block\_check(struct thread \*t, void \*aux UNUSED)

{

if (t->status == THREAD\_BLOCKED&&t->block\_ticks>0)//看这个线程是不是阻塞态

{

t->block\_ticks--; //将等待的时间自减

if (t->block\_ticks == 0)

{

thread\_unblock (t); //把t线程重新”解锁“

}

}

}

4.修改timer\_interrupt函数，使其每次中断都对所有线程执行block\_check函数

static void

timer\_interrupt (struct intr\_frame \*args UNUSED)//UNUSED是一个宏，表示这个参数没用

{

enum intr\_level old\_level; //和先前一样，记录原来的中断状态

old\_level=intr\_disable(); //forreach函数要求关中断。

ticks++;

thread\_foreach (block\_check, 0);

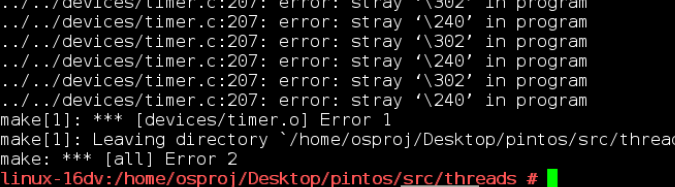
intr\_set\_level (old\_level); //恢复中断

thread\_tick (); //交给操作系统出发中断并且调度新的线程进驻cpu。

}

5.实现中断

修改完后,编译:



编译之后出现一堆错误、因为是编码的问题。我们是把timer.c 复制到Windows下用notepad++修改的源码的。

修改后编译，仍然出现错误。

测试test下的alarm\_multiple..运行和上次结构一样。

**Part Two**

没有修改之前pintos的状态：

在timer\_sleep函数中采用了一个while循环，如果当前的时间小于ticks,

则不断执行thread\_yield;会产生一种忙等的状态。

目标：

消除这种忙等状态，给thread一个等待时间的属性，在执行timer\_sleep的时候，获得此属性并将thread放入一个block的状态之中，再在适当的时间进行thread的唤醒。

I 新建一个block队列，将被阻塞的线程按照等待时间的大小按顺序放入这个队列中。在timer\_interrupt里面进行唤醒。

II 不需要新建block队列，直接将线程设置为block态，唤醒时采用foreach调用（默认遍历readylist）来依次判断thread的睡眠时间有没有到，如果到了就可以将其唤醒。

需要修改的部分

1、../devices/timer.c

i）timer\_sleep

ii）timer\_interrupt

2、../threads/thread.h

i) +thread\_sleep

ii) +thread\_wakeup

iii) +wakeup\_ticks

3、../threads/thread.c

i) thread\_sleep

ii) thread\_wakeup

iii) block\_list

timer\_sleep的作用：让此线程等待ticks单位时长，之后再执行

修改以前:

while (timer\_elapsed (start) < ticks)

//如果elapse（流逝）的时间>=ticks时就返回。否则将持续占用CPU

thread\_yield ();

//用于把current\_thread进入ready list，任意时刻可再次被调用。Thread\_yield中会调用一个schedule的调度函数，再进行调度

一般的CPU在1个ticks中可以执行很多个这样的循环，这会产生忙等, 改进：

在timer\_sleep函数中加入thread\_sleep(ticks)让当前线程阻塞进入block\_list（需要我们自己添加），经过ticks个时间单位后，通过调用thread\_wakeup函数再次被唤醒进入ready队列。

这是对目前pintos实现的改进

timer\_sleep 改后代码

void timer\_sleep(int64\_t ticks)

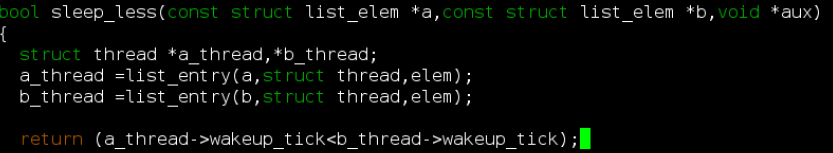
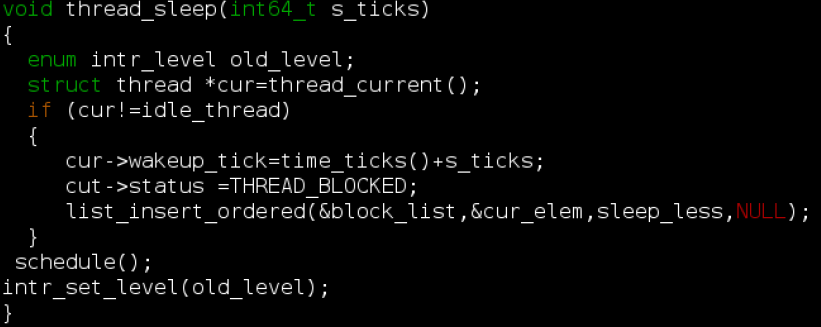
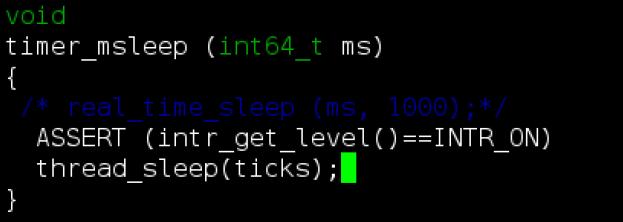
{

ASSERT (intr\_get\_level() == INTR\_ON);

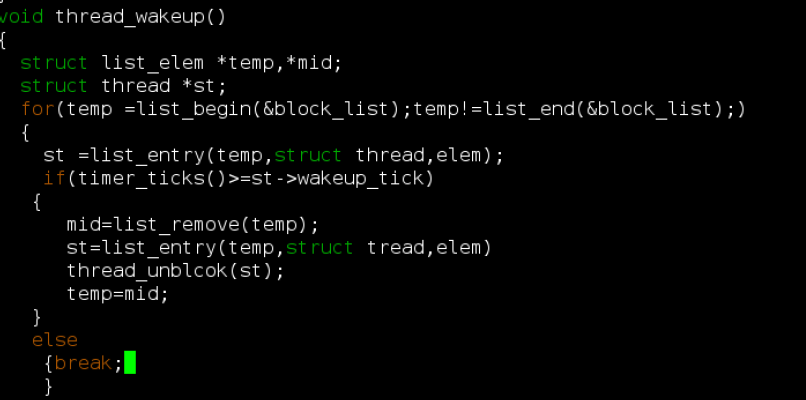
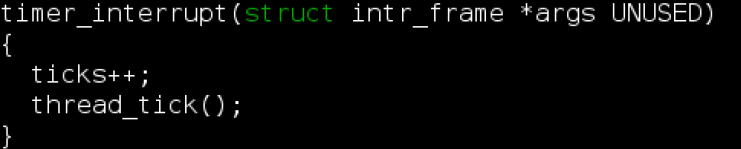
thread\_sleep(ticks);

//加上thread\_sleep函数

}



这个函数是帮助比较两个thread的wakeup\_ticks的大小



2.对schedule（void）函数的修改：

static void

schedule (void)

{

struct sleep\_elem \*ret;

struct list\_elem \*l;

struct thread \*t;

int64\_t now\_time;

for (l = list\_begin (&sleep\_list); l != list\_end (&sleep\_list); l = list\_next (l))

{

ret = list\_entry (l, struct sleep\_elem, elem);

if (ret->ticks > now\_time)

{

list\_remove(&ret->elem);

t = thread\_by\_tid(ret->tid);

thread\_unblock(t);

}

}

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);

}

这里用了系统本身实现好的双向链表list，遍历sleep\_list,找到到时的进程（et->ticks > now\_time）。将到时的进程remove出sleep\_list，并加入ready\_list，这样下面在调度时就可以选到该进程了。

优先级调度:

在线程进行调度的过程中，根据线程的优先级别来进行调度，每次从ready\_list

中移出到running\_list中的线程都应该保证是ready\_list中优先级最高的线

程，如果操作系统当前创建了一个新的线程，并且该线程的优先级大于正在运行的

线程的优先级，那么当前运行的线程要yield，交出CPU。

现在pintos的进程调度方法是FCFS，及先到先服务。我们需要把它改为优先级调度算法，要求可以抢占，可以重设优先级。同时官方文档要求当一个进程用CLOCK，SEMA，COND被阻塞时必须让出CPU，更重要的是同时要避免priority adversity。

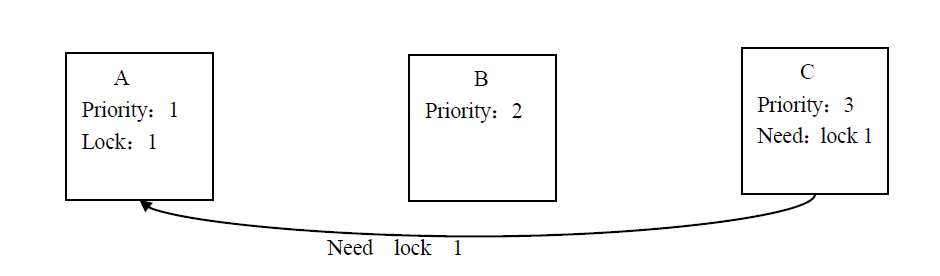
一个进程要想进入ready\_list就要调用thread\_unblock，在此函数中加入调度功能，就能完成抢占的要求。

若进程的优先级被改变，及调用thread\_set\_priority，只需同时再调用schedule（），就进一步完善了抢占的要求。

Priority adversity比较难解决，方法是在struct thread中新添加thread\* fath，指向被它阻塞的进程中优先级最高的那一个，在锁被打开时，将fath重置为NULL即可。

优先级捐赠：

优先级调度在锁得问题上会产生优先级反转的问题。例如：



此时lock 1 被A 线程持有，线程C 也需要lock 1，因此被阻塞掉。但是A

线程的优先级低于B线程，B线程会先于A线程执行。因此执行顺序会变成B，A，C

使优先级发生了翻转。

数据结构:

在thread.h中的线程的结构体中添加int型变量priority\_old 用来记录受到捐赠前得优先级,加入一个队列locks 记录这个线程所持有的锁。一个bool型变量记录是否发生过优先级的捐赠。一个指向锁得指针，用来记录这个线程被哪把锁所阻塞。

int priority; /\* Priority. \*/

int priority\_old; /\*Priority before donated\*/

struct list locks; //record the lock that this thread has

bool donated; //To identifer that if there is donation

struct lock \*blocked; // by which lock this thread is blocked

在synch.h中结构体lock中添加队列元素，用来记录持有这个锁的所有的线程。

其中添加了一个整形变量lock\_priority,锁是没有优先级的。这个变量是用来

记录等待该锁的线程的最高的优先级。

struct list\_elem holder\_elem; //The elem of lock list in struct thread

int lock\_priority; /\* The highest priority waiting for the lock \*/

在环境变量condition中添加holder指针记录持有这个环境变量的线程。

struct thread \*holder; // Thread holding condition

具体解决方法及关键代码

1.对thread\_unblock的修改：

void

thread\_unblock (struct thread \*t)

{

enum intr\_level old\_level;

ASSERT (is\_thread (t));

old\_level = intr\_disable ();

ASSERT (t->status == THREAD\_BLOCKED);

list\_push\_back (&ready\_list, &t->elem);

t->status = THREAD\_READY;

intr\_set\_level (old\_level);

if(thread\_mlfqs)

{

adv\_pri[t->priority]++;

schedule\_ad(thread\_current(),0);

}

else

schedule\_x();

}

也及在最后一行加了调度的函数。

2.修改调度函数

由于担心pintos在其他地方还要用schedule（），就重新建立了一个调度函数scedule\_x（）:

static void schedule\_x(void)

{

struct thread \*cur = running\_thread ();

struct thread \*next ,\*fath;

struct thread \*prev = NULL;

struct list\_elem \*e;

int priority;

if (list\_empty (&ready\_list))

return ;

else

{

e=list\_begin(&ready\_list);

struct thread \*now = list\_entry (e, struct thread, elem);

priority=f->priority;

for ( ; e != list\_end (&ready\_list); e = list\_next (e))

{

struct thread \*f = list\_entry (e, struct thread, elem);

for(fath=f; fath->fath != NULL ; fath=fath->fath);

if(now->priority < fath->priority)

now = f;

}

if(cur -> priority >= now -> priority)

return;

list\_remove(&now->elem);

next=now;

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);

}

}

不是通过调用next\_thread\_to\_run来取得下一个进程，而是通过遍历ready\_list，找到优先级最高的进程。

值得注意的是，查看一个进程是否优先级最高，并不是直接看起thread中priority的值，而是要通过fath，找到覆盖其优先级的进程（可能是多级的),看此进程的priority。

3.对lock的修改（由于另外两个十分相似，且lock要求解决优先级逆转问题，所以这里就不赘述sema，cond的实现方法）

当一个进程申请锁，但该锁已被占有时，会调用sema\_down，并在一个while循环里不停thread\_block，直到该锁解开。我们在这里做个判断，若申请者的优先级高于持锁人的等价优先级，就将持锁人的fath指向改进程：

while(f != NULL)

{

priority=f->priority;

f=f->fath;

}

ASSERT (loc->semaphore != NULL);

ASSERT (!intr\_context ());

old\_level = intr\_disable ();

while (loc->semaphore->value == 0)

{

list\_push\_back (&loc->semaphore->waiters, &thread\_current ()->elem);

if(t->priority > priority)

{

loc->holder->fath=t;

}

thread\_block ();

}

类似的，sema\_up函数也需要修改：

void

sema\_up\_x (struct lock \*loc)

{

enum intr\_level old\_level;

ASSERT (loc->semaphore != NULL);

old\_level = intr\_disable ();

if (!list\_empty (&loc->semaphore->waiters))

thread\_unblock (list\_entry (list\_pop\_front (&loc-.semaphore->waiters),

struct thread, elem));

loc->fath=NULL;

loc->semaphore->value++;

intr\_set\_level (old\_level);

}

及要将fath置为NULL。

调度算法:

调度算法要再一次修改，要从优先级调度升级为多级反馈队列优先级调度。其中设计到nice，recent\_cpu , load\_avg的计算，并在有规律的改变进程的优先级。调度时调度高优先级的进程，对于同优先级的进程，用RR电镀算法。

由于原先的调度算法维护了一个ready\_list，我在此也没有新建立list或将其排序，只是维护了一个adv\_pri[64]的数组，用来记录各优先级进程的数量，从而完成调度。

由于每个tick，系统都会产生中断，并调用函数thread\_tick，所以，就在此函数中判断

时间的流失，并相应更新有关的变量值。

修改如下

1. 对thrad\_tick函数的修改：

thread\_tick (void)

{

struct thread \*t = thread\_current ();

/\* Update statistics. \*/

if (t == idle\_thread)

idle\_ticks++;

#ifdef USERPROG

else if (t->pagedir != NULL)

user\_ticks++;

#endif

else

kernel\_ticks++;

if(thread\_mlfqs)

{

t->recent\_cpu++;

long long tick=idle\_ticks + user\_ticks + kernel\_ticks;

if(tick % 4 == 0)

{

struct thread \*t;

struct list\_elem \*l;

int ad;

for( ad = 0; ad <64 ;ad++)

adv\_pri[ad]=0;

for (l = list\_begin (&all\_list); l != list\_end (&all\_list); l = list\_next (l))

{

t = list\_entry (l, struct fd\_elem, elem);

t->priority=PRI\_MAX -(t->recent\_cpu / 4)-(t->nice \* 2)

adv\_pri[t->priority]++;

}

schedule\_ad(thread\_current(),0);

}

if(tick % 5 == 0 )

//时间片论转

{

schedule\_ad(thread\_current(),1);

}

if(tick % 100 == 0)

{

struct thread \*t;

struct list\_elem \*l;

for (l = list\_begin (&all\_list); l != list\_end (&all\_list); l = list\_next (l))

{

t = list\_entry (l, struct fd\_elem, elem);

t->recent\_cpu = (2\*load\_avg )/(2\*load\_avg + 1) \* t->recent\_cpu + t->nice

}

int read\_thread=0;

read\_thread=list\_size(&ready\_list)+1;

load\_avg=(59/60)\*load\_avg+(1/60)read\_thread;

}

}每4个ticks修改一次priority，并重新调度( schedule\_ad() );

每100个ticks（及1秒），更新一次每个进程的recent\_cpu，及load\_adv

每5个ticks,做一次RR轮转。

2.对调度的更改，由于调度分两种，一种是一定要换一个进程（及RR调度），另一种是检测现在运行的进程是否需要被抢占。所以重新维护一个调度函数，并增加参数thread\*和一个int，int是个模式位，用来区分两种调度。

我们考虑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。

BSD调度可以解决这样的问题：对于在内存里等待运行的线程，BSD调度能合理的分配I/O操作和CPU的使用，这样减少平均等待时间，提高系统的吞吐率。BSD调度用到的结构是多级反馈队列，即多个ready\_list，每个队列有不同的优先级，对于队列里存放的线程，要在struct thread里添加一个变量nice，其值为-20~20。Nice是一个抽象的变量，大体上可以理解为nice是一个对优先级有影响的变量：正的nice使thread的优先级降低，负的nice使thread的优先级提高。用nice计算线程的优先级的公式如下所示：

priority = PRI\_MAX -(recent\_cpu / 4) -(nice \* 2)

这个优先级首先在thread\_init( )中计算，然后每过4个tick计算一次（在timer.c中实现）。

公式中出现了一个变量recent\_cpu，这个变量是对线程使用CPU时间的估计（下取整）

recent\_cpu的计算公式为：

recent\_cpu = (2\*load\_avg)/(2\*load\_avg + 1) \* recent\_cpu+ nice

对于recent\_cpu，每一个中断到来就加1，每一秒recent\_cpu就会被重新计算一次。

在这个公式中有出现了一个变量load\_avg。这个变量表示ready\_list中thread的数量，计算load\_avg的公式如下：

load\_avg = (59/60)\*load\_avg + (1/60)\*ready\_threads

数据结构

在thread.h 中添加

int nice; 记录该线程的友好度。

int recent\_cpu;记录CPU的使用时间。

在timer.c 中添加

load\_avg 几率平均负载。

算法分析。

实现BSD调度的话就要避免进行优先级的捐赠。因此需要在相关优先级捐赠的函数中，进行判断，通过语句if(thread\_mlfqs)屏蔽掉相关代码。

修改：lock\_acquire ( ),lock\_realse ( ) 屏蔽相关功能。

添加或修改：

int thread\_get\_nice (void);

void thread\_set\_nice (int);

int thread\_get\_recent\_cpu (void);

int thread\_get\_load\_avg (void);

void thread\_caculate\_load\_avg(void);

void thread\_caculate\_recent\_cpu(struct thread \*cur);

void thread\_caculate\_recent\_cpu\_for\_all(void);

实现线程nice，rencent\_cpu ，系统load\_avg的计算和更新。

修改timer.c 中的timer\_sleep( )，timer\_interrupt( )函数，

最后在timer.c中实现动态计算优先级及各指标的功能。

测试结果：

