Lab1 线程机制 实验报告

内容一：总体概述

本实习主要内容是扩展线程机制和扩展调度算法。在阅读源码的基础上，我实现了用户ID、线程ID、最大线程数维护、TS功能、基于优先级的抢占式调度算法、时间片轮转算法（Challenge）。我认为比较重要的在于理解线程、调度器、中断器、时钟都在如何运行，并应该负责哪些功能，这决定着Exercise的实现方式。

内容二：任务完成情况

任务完成列表

Exercise1	Exercise2	Exercise3	Exercise4	Exercise5	Challenge
Y	Y	Y	Y	Y	Y

调研

调研Linux或Windows中进程控制块（PCB）的基本实现方式，理解与Nachos的异同。调研Linux或Windows中采用的进程/线程调度算法。

Linux中PCB实现

- 类型为`task_struct`，称为进程描述符/进程控制块的结构，存放在“任务队列”双向循环链表中。
- Linux内核为每个进程提供一个内核栈（在PCB中描述），在2.6以前各个进程的`task_struct`放在它们内核栈的尾端，2.6之后在内核栈的栈顶（对于向上增长的栈）创建新的结构`struct thread_info`，其中有一个指向PCB的指针。
- 进程id`pid`、文件打开表的指针`files`、栈指针`stack`及相关的虚拟内存页表信息、调度策略`policy`、实时优先级`rt_priority`、进程相应的程序和数据地址、资源清单、 同步机制的信号量、所在队列的PCB链接字（指向队列下一个PCB的首地址）、进程的创建时间、消耗时间等。
- 进程状态`state`：
    - TASK_RUNNING： 进程可执行，它要么正在执行，要么在运行队列中等待执行。
    - TASK_INTERRUPTIBLE： 可中断，进程正在睡眠（被阻塞），可被唤醒。
    - TASK_UNINTERRUPTIBLE： 不可中断，收到信号也不会被唤醒或准备运行。
    - TASK_STOPPED： 进程停止执行。
    - TASK_ZOMBIE： 退出状态，成为僵尸进程，不相应信号，无法被sigkill。

Nachos中PCB实现

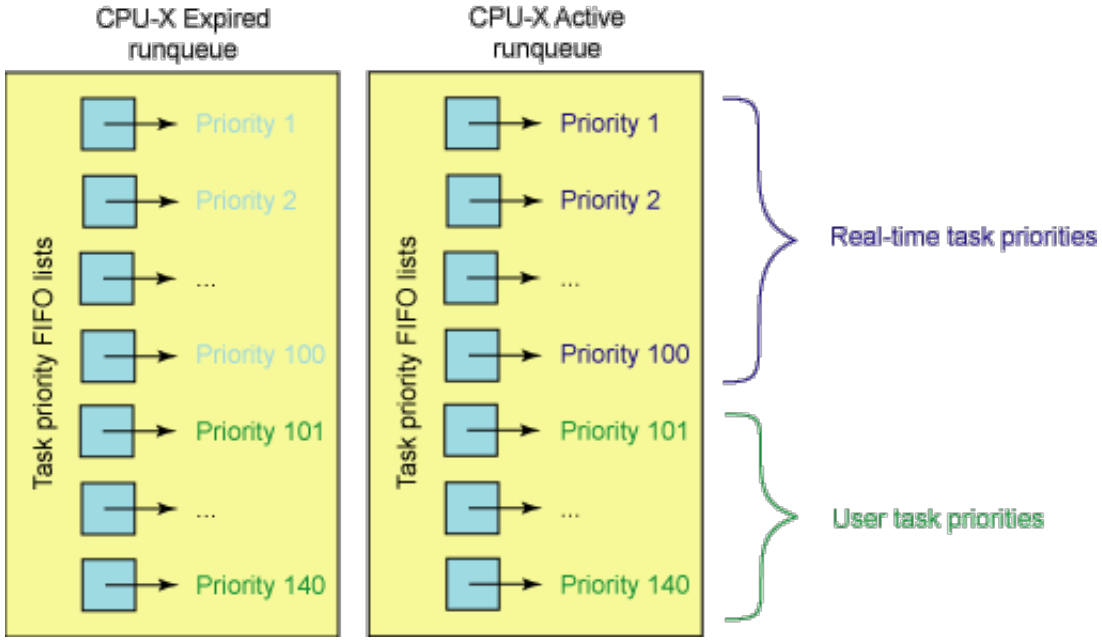
- `int* stackTop  //the current stack pointer`
- `int machineState[MachineStateSize]  //all registers except for stacktop`
- `int* stack  //Bottom of the stack`
- `ThreadStatus status`
  - `enum ThreadStatus { JUST_CREATED, RUNNING, READY, BLOCKED }`

Linux进程调度

Linux2.4 O(n)调度器

- 每个进程被创建时被赋予一个时间片，时钟中断递减当前运行进程的时间片。调度器保证只有所有RUNNING进程的时间片都被用完之后，才对所有进程重新分配时间片，此为一个epoch。
- pick next：对runqueue中所有进程的优先级依次进行比较，取最高优先级进程下一次被调度。
- 实时进程的优先级是静态的（内核不为其计算动态优先级），始终大于普通进程的优先级。
  - SCHED_FIFO：没有时间片概念，一直占有CPU直到更高优先级进程就绪或者主动放弃，不在乎同优先级进程。
  - SCHED_RR：相对FIFO增加时间片概念，使得同优先级进程可以轮流执行（若不同还是会使高优先级进程一直运行）。
- 普通进程中，调度器倾向于提高交互式进程的优先级。
- Linux2.4内核是非抢占的，当进程处于内核态时不会发生抢占。

Linux2.6 O(1)调度器

- 支持内核态抢占，更好地支持了实时进程。
- pick next：调度器维护两个进程队列数组：指向活动运行队列的active、指向过期运行队列的expire。数组中的元素是某一优先级的进程队列指针，所以两个数组大小都是140。O(1) 调度器直接从 active 中选择当前最高优先级队列中的第一个。
  - 

- 交互式进程和实时进程时间片用尽后，会重置时间片并插入 active 中，其他进程被插入 expire 中；当交互式进程和实时进程占用 CPU 时间达到某个阈值后，会被插入到 expire 中，以防expire中其他进程饥饿。当一个优先级的active队列变为空时，对应的expire队列转变为expire队列(只需要指针的交换）。
- 维护一个bitmap来在常数时间找到优先级最高的队列。

CFS调度器

“CFS在真实的硬件上模拟了完全理想的多任务处理器。”

- 设定一个调度周期`sched_latency_ns`，目标是让每个进程在这个周期内至少有机会运行一次。
- > cfs中的时间片是动态分配的，是按照比例分配的而不是按照优先级固定分配的，其精髓就是系统拥有一个可配置的系统调度周期，在该周期内运行完所有的进程，如果系统负载高了，那么每一个进程在该周期内被分配的时间片都会减少，将这些进程减少的部分累积正好就是新进程的时间片。
  >
  > 1秒钟内的进程切换次数就不和进程优先值有关，而是和调度周期和进程数量有关，理论上就是(进程数量)*(1秒/调度周期)次，当然加上抢占和新进程创建就不是这么理想了。
- 对每个CPU维护一个以时间为顺序的红黑树
- 每个进程维护一个虚拟的已运行时间`vruntime`，虚拟运行时间越小的进程越靠近整个红黑树的最左端。
- 内核中通过prio_to_weight数组进行nice值和权重的转换、vruntime计算：

$$\text{分配给进程的运行时间} = \text{调度周期} * \text{进程权重} / \text{进程总权重} \tag{1}$$

$$vruntime = \text{实际运行时间} * (\text{NICE\_0\_LOAD}) / \text{进程权重} \tag{2}$$

$$vruntime = (\text{调度周期} * \text{进程权重} / \text{进程总权重}) * \text{NICE\_0\_LOAD} / \text{进程权重} = \text{调度周期} * \text{NICE\_0\_LOAD} / \text{所有进程总权重} \tag{3}$$

可以看到所有进程的vruntime在宏观上是同时推进的，虽然实际运行时间不同，但虚拟运行时间的增长速度一致，代表着“获得了一样多的时间。”

Exercise 1 源代码阅读-理解现有的线程机制

thread.cc和thread.h

定义并实现了线程

```
int* stackTop;                                  //栈顶指针
int machineState[MachineStateSize];             //存放栈顶指针以外的寄存器
void Fork(VoidFunctionPtr func, int arg);    //线程设置自己得到cpu后要执行的函数，然后调度器将进程放入就绪队列
void Yield();           //线程放弃cpu，如果调度器调度下一个进程非空，则让当前进程进入就绪队列，让下一个进程运行
void Sleep();           //线程进入阻塞态，如果调度器调度下一个进程非空，则让下一个进程运行，否则cpu进入空闲态
void Finish();          //设置threadToBeDestroyed为自己，然后自己调用sleep
                            //释放过程在scheduler.cc的run()中，由下一个运行的进程来释放)
int* stack;             //栈底指针
void StackAllocate(VoidFunctionPtr func, int arg);     //给线程分配栈
```

main.cc

- 调用system.cc的Initialize分析命令行传入参数并初始化统计器、中断、调度器、时钟、创建main线程
- 检测到-q参数则进入线程模块的测试函数
- 最后调用Finish释放最后的线程

threadtest.cc

根据-q参数后连接的数字选择使用的测试函数，默认调用ThreadTest1

```
void ThreadTest1()
{
    Thread *t = new Thread("forked thread");  //新建一个名叫forked thread的线程
    t->Fork(SimpleThread, 1);    //新线程将自己放进就绪队列，若切换到新线程则调用参数为“1”的simpleThread函数
    SimpleThread(0);     //当前线程（即main线程）调用simpleThread
}
```

```
void
SimpleThread(int which)
{
    int num;
    for (num = 0; num < 5; num++) {
        printf("*** thread %d looped %d times\n", which, num);
        currentThread->Yield();    //每个线程在上面的打印信息后让出cpu，等再次切换上cpu后再往下走
    }                              //所以效果是两个线程轮流打印，各5次
}
```

Exercise 2 扩展线程的数据结构

> 增加“用户ID、线程ID”两个数据成员，并在Nachos现有的线程管理机制中增加对这两个数据成员的维护机制。

首先在thread.h中增加

```
/*  thread.h  class Thread  */
private:
  int userID;
  int tid;

public:
  void init(char* threadName);    //为了方便构造函数的重载，见后文
  int getUserID(){ return (userID); }   //用户id接口
  int getTid(){ return (tid); }     //线程id接口
  void setUserID(int userid){ userID = userid; }  //设置用户id
```

用户id暂时都使用默认值，而线程id应该向scheduler申请(Exercise 3中的线程数量限制一并实现)，于是在scheduler.h中增加

```
/*  thread.h  */
#define MAX_THREAD_NUMBER 128   //定义线程池容量

/*  class Scheduler  */
private:
    std::queue<int> tids;   //可用的tid队列

public:
    int acquireTid(Thread *t);  //用于给线程分配tid
    void releaseTid(Thread *t); //给线程释放tid
```

在thread构造函数中申请tid，所以增加

```
/*  thread.cc   Thread::init(char* threadName)
  在构造函数中调用此函数，减少重载构造函数的代码量
  */
this->tid = scheduler->acquireTid(this);    //申请一个tid
if(this->tid == -1)     //没有tid了，即线程池满了
    printf("No more space to create new thread!");
ASSERT(this->tid != -1);    //线程满了就退出程序
```

在scheduler构造函数中应该增加对可用tid队列的初始化

```
/*  Scheduler::Scheduler()  */
//初始化可用tid队列
for(int i = 0; i < MAX_THREAD_NUMBER; i++)
{
  this->tids.push(i);
}
```

scheduler分配tid与收回tid实现

```
/*  scheduler.cc  */
int Scheduler::acquireTid(Thread *t)
{
    if(tids.size() == 0)
        return -1;
    int tid = tids.front();   //将线程队列头分配给新线程
    tids.pop();
    return tid;
}

void Scheduler::releaseTid(Thread *t)   //线程的析构函数中调用
{
    tids.push(t->getTid());   //将线程的tid放回到可用tid队列的队尾
}
```

Exercise 3 增加全局线程管理机制

> 在Nachos中增加对线程数量的限制，使得Nachos中最多能够同时存在128个线程；
>
> 仿照Linux中PS命令，增加一个功能TS(Threads Status)，能够显示当前系统中所有线程的信息和状态。

在Exercise 2中，通过对`scheduler->tids`的初始化、`acquireTid()`的实现已经达到了限制线程数量的目的。

因为scheduler无法通过tid得到线程的其他信息，所以需要在Scheduler类中增加一个数据结构实现线程池，存储指向thread的指针：

```
/*  scheduler.h class Scheduler  */
private:
    std::vector<Thread *> threadPool;    //用vector实现线程池
```

thread构造函数与析构函数时需要对线程池进行增减，即修改`Scheduler::acquireTid(Thread *t)`与`Scheduler::releaseTid(Thread *t)`

```
//-----------------------------------------------------------------------
//用于给线程分配tid
//若可用tid队列为空则返回-1
//若不为空，则分配队首数值作为tid，并把线程指针放入线程池
//-----------------------------------------------------------------------
int Scheduler::acquireTid(Thread *t)
{
    if(tids.size() == 0)
        return -1;
    int tid = tids.front();
    tids.pop();
    threadPool.push_back(t);
    return tid;
}

//-----------------------------------------------------------------------
//用于给线程释放tid
//在线程池vector找到线程指针，调用erase方法删除
//-----------------------------------------------------------------------
void Scheduler::releaseTid(Thread *t)
{
    tids.push(t->getTid());
    for(std::vector<Thread *>::iterator t0 = threadPool.begin(); t0 <
threadPool.end(); t0++)
    {
        if(*t0 == t)
        {
            threadPool.erase(t0);
            break;
        }
    }
}
```

然后就可以实现TS功能了，只需遍历一遍`scheduler->threadPool`即可

```
//-----------------------------------------------------------------
//打印所有进程信息
//线程状态返回的原本是枚举类型变量，C语言中，枚举类型是被当做int或unsigned int类型来处理的
//于是这里写一个int到字符串的映射，用来输出线程状态
//-----------------------------------------------------------------
```

```
void Scheduler::TS()
{
    const char* TStoString[] = {"JUST_CREATED", "RUNNING", "READY", "BLOCKED"};
    printf("USERID\tTID\tNAME\tSTATUS\n");
    for(std::vector<Thread *>::iterator t0 = threadPool.begin(); t0 <
threadPool.end(); t0++)
    {
        printf("%d\t%d\t%s\t%s\n", (*t0)->getUserID(), (*t0)->getTid(), (*t0)-
>getName(), TStoString[(*t0)->getThreadStatus()]);
    }
}
```

测试TS功能

```
void Lab1Exercise3_2()
{
    Thread *t = NULL;
    for(int i = 0; i < 4; i++)
        Thread *t = new Thread("Test Thread");
    scheduler->TS();
}
```

```
vagrant@precise32:/vagrant/nachos/nachos-3.4/code/threads$ ./nachos -q 3
when: 134539339   stats->totalTicks: 10
USERID  TID     NAME    STATUS
0       0       main    RUNNING
0       1       Test Thread     JUST_CREATED
0       2       Test Thread     JUST_CREATED
0       3       Test Thread     JUST_CREATED
0       4       Test Thread     JUST_CREATED
```

Exercise 4 源代码阅读、理解现有线程调度方法

> 仔细阅读下列源代码，理解Nachos现有的线程调度算法
>
> - code/threads/scheduler.h和code/threads/scheduler.cc
> - code/threads/switch.s
> - code/machine/timer.h和code/machine/timer.cc

nachos实现的线程调度方法是

- FIFO调度，仅当线程主动在`yield`等函数中放弃cpu时，scheduler将`readyList`队首线程调度上cpu。
- 当传入参数-rs时，在system.cc中`randomYield = True`，并初始化一个Timer

```
if (randomYield)        // start the timer (if needed), 即存在rs参数
    timer = new Timer(TimerInterruptHandler, 0, randomYield);
```

Timer运行机制：`TimerHandler`与`TimerExpired`函数循环互相调用来不断向中断队列插入中断，在Timer的构造函数中第一次调用了`TimerHandler`插入第一个中断，使循环调用开始。`TimerExpired`最后调用system.cc中定义的`TimerInterruptHandler`设置`yieldOnReturn`标志为True，这使得每次`onetick`时当前线程都主动yield，即实现了不均匀的时间片轮转调度。需要明白的是nachos用软件模拟硬件（时钟），到目前的实验只有在开关中断setLevel函数中调用了onetick函数，即只有这时时间前进。

```
/*  timer.cc  */
static void TimerHandler(int arg)
{ Timer *p = (Timer *)arg; p->TimerExpired(); }

void Timer::TimerExpired()
{
    // schedule the next timer device interrupt
    // 这是一种循环调用，为了中断后能插入下一个中断
    interrupt->Schedule(TimerHandler, (int) this, TimeOfNextInterrupt(),
    TimerInt);

    // 调用TimerInterruptHandler设置yieldOnReturn标志为True
    (*handler)(arg);
}
```

Exercise 5 线程调度算法扩展

> 扩展线程调度算法，实现基于优先级的抢占式调度算法。

1. 在thread里加入`priority`属性、`getPriority`方法、`setPriority`方法、用于设置优先级的构造函数重载：

```
/*  thread.h  class Thread  */
private:
  int priority;

public:
  Thread(char* threadName, int prio);
  int getPriority(){ return priority; }
  void setPriority(int prio){ priority = prio; }
};
```

2. 原本fork时scheduler直接把线程指针加到就绪队列的队尾，现在则要与就绪队列中元素比较优先级后插入：

```
void Scheduler::ReadyToRun (Thread *thread)
{
    thread->setStatus(READY);
    switch(scheduleMethod)
    {
        case PRIORIY:
            readyList->SortedInsert((void *)thread, thread->getPriority());
//根据优先级插入
            break;
        default:
            readyList->Append((void *)thread);
    }
}
```

3. 抢占式的实现，即当优先级更高的线程“到达”时可以直接运行，在这里实现为新线程fork时进行抢占：

```
void Thread::Fork(VoidFunctionPtr func, int arg)
{
   + scheduler->preemptive_sched(this);       //新线程fork时让scheduler判断是否抢占
}

void Scheduler::preemptive_sched(Thread* t)
{
    if(scheduleMethod == PRIORIY)
        if(currentThread->getPriority() > t->getPriority())    //若新线程优先级更高
            currentThread->Yield();                            //当前线程主动yield
    return;
}
```

4. 测试

```
void Lab1Exercise5()
{
    Thread* t3 = new Thread("thread 3", 3);
    t3->Fork(Lab1Exercise3Thread, 3);   //第二个参数是传给Lab1Exercise3Thread的，要与线程名一致，而不是tid

    Thread* t2 = new Thread("thread 2", 2);
    t2->Fork(Lab1Exercise3Thread, 2);

    Thread* t1 = new Thread("thread 1", 1);
    t1->Fork(Lab1Exercise3Thread, 1);
}
```

```
void
Lab1Exercise3Thread(int which)
{
    int num;
    for (num = 0; num < 2; num++) {
  printf("*** thread %d (userID = %d, tid = %d) looped %d times\n",
which,currentThread->getUserID(),\
                              currentThread->getTid(),num);
        currentThread->Yield();
    }
}
```

优先级越小越优先，main线程优先级是3，所以下面的代码的线程运行顺序应该是main->t2（抢占）->t3->t2->main->t1（抢占）->t1->t3->main。

```
vagrant@precise32:/vagrant/nachos/nachos-3.4/code/threads$ ./nachos -q 4
*** thread 2 (userID = 0, tid = 2) looped 0 times
*** thread 3 (userID = 0, tid = 1) looped 0 times
*** thread 2 (userID = 0, tid = 2) looped 1 times
*** thread 1 (userID = 0, tid = 3) looped 0 times
*** thread 1 (userID = 0, tid = 3) looped 1 times
*** thread 3 (userID = 0, tid = 1) looped 1 times
No threads ready or runnable, and no pending interrupts.
```

符合预测，实现完成。

Challenge 1 线程调度算法拓展

> 可实现“时间片轮转算法”、“多级队列反馈调度算法”，或将Linux或Windows采用的调度算法应用到Nachos上。

这里我实现了时间片轮转算法。

1. 在scheduler中增加一个数组用来存储各个线程的时间片，以tid为索引。在线程的构造函数中调用`initTicks`即可。

```
/*  scheduler.h class Scheduler  */
private:
    int tid_Ticks[MAX_THREAD_NUMBER]; //一个tid对应一个ticks计数

  public:
    void initTicks(Thread *t){ tid_Ticks[t->getTid()] = 0; }
    int checkTicks(Thread *t){ return tid_Ticks[t->getTid()]; }   //返回线程已用时间数
    void addTicks(Thread *t, int ticks){ tid_Ticks[t->getTid()] += ticks; }
//onetick时调用
    void resetTicks(Thread *t){ tid_Ticks[t->getTid()] = 0; }
```

2. 在时间前进（onetick）时，scheduler为当前线程增加已使用时间

```
/*  interrupt.cc onetick()  */
if (status == SystemMode) {
    stats->totalTicks += SystemTick;
    stats->systemTicks += SystemTick;
    + scheduler->addTicks(currentThread, SystemTick);      //scheduler为当前
线程增加已使用时间
```

3. 在system.cc里新增一个handler（即改变中断处理函数），在创建时钟时传入。新的中断处理函数增加了对线程时间片是否用完的判断，若用完了才设置`YieldOnReturn`标志。这样就不会每次中断时（不是每次onetick）都切换进程了。

```
/*  system.cc  */
static void RRHandler(int dummy)
{
    if(scheduler->checkTicks(currentThread) >= TimerTicks){    //若线程已用时
间大于时间片
        if(interrupt->getStatus() != IdleMode){    //且系统非空闲态
            printf("CurrentThread used up TimerTicks\n");
            interrupt->YieldOnReturn();
            scheduler->resetTicks(currentThread);
        }
        else{
            printf("readylist is Empty\n");
        }
    }
    else{
        printf("\n");
    }
}

void Initialize(int argc, char **argv)
{
    + if(!strcmp(*argv, "-rr")) {      //检测到-rr，启用时间片轮转调度
    +       ASSERT(argc > 1);
    +       roundRobin = TRUE;
    +       argCount = 2;
    +   }
    + if(roundRobin)                   //开始RR时间计数器
    +    timer = new Timer(RRHandler, 0, FALSE);
}
```

4. 测试

   设定`timeticks`为30,便于观察结果

```
void Lab1Challenge1Thread(int which)
{
```

```
    for(int i = 0; i < 5; i++){
        printf("*** %s looped %d times with ticks: %d\n", currentThread-
>getName(), i + 1,
                                scheduler->checkTicks(currentThread));
        interrupt->OneTick();
    }
}

void Lab1Challenge1()
{
    DEBUG('t', "Entering Lab1Challenge1");
    Thread* t3 = new Thread("Thread 3");
    t3->Fork(Lab1Challenge1Thread, 3);

    Thread* t2 = new Thread("Thread 2");
    t2->Fork(Lab1Challenge1Thread, 2);

    Thread* t1 = new Thread("Thread 1");
    t1->Fork(Lab1Challenge1Thread, 1);
}
```

```
vagrant@precise32:/vagrant/nachos/nachos-3.4/code/threads$ ./nachos -rr -q
5
Handling....Next interrupt time: 30   Now: 30
CurrentThread used up TimerTicks
*** Thread 3 looped 1 times with ticks: 10
*** Thread 3 looped 2 times with ticks: 20
Handling....Next interrupt time: 60   Now: 60
CurrentThread used up TimerTicks
*** Thread 2 looped 1 times with ticks: 10
*** Thread 2 looped 2 times with ticks: 20
Handling....Next interrupt time: 90   Now: 90
CurrentThread used up TimerTicks
Handling....Next interrupt time: 120   Now: 120

*** Thread 3 looped 3 times with ticks: 10
*** Thread 3 looped 4 times with ticks: 20
*** Thread 3 looped 5 times with ticks: 30
Handling....Next interrupt time: 150   Now: 150
CurrentThread used up TimerTicks
*** Thread 2 looped 3 times with ticks: 10
*** Thread 2 looped 4 times with ticks: 20
Handling....Next interrupt time: 180   Now: 180
CurrentThread used up TimerTicks
*** Thread 1 looped 1 times with ticks: 10
*** Thread 1 looped 2 times with ticks: 20
Handling....Next interrupt time: 210   Now: 210
CurrentThread used up TimerTicks
```

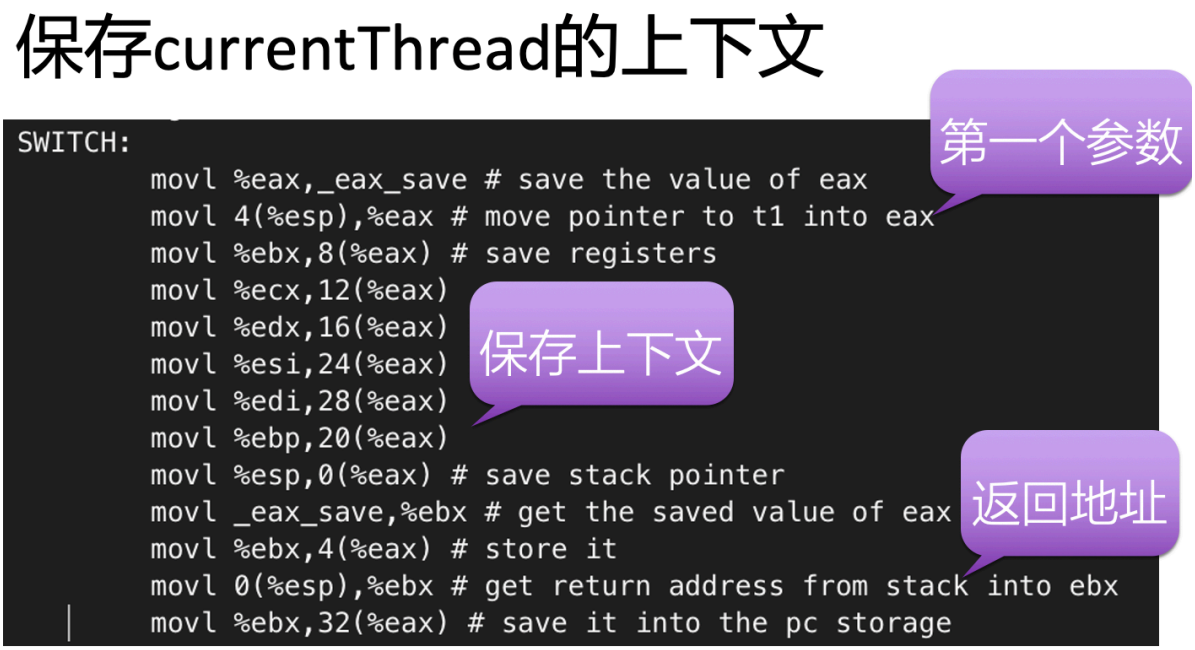
```
*** Thread 2 looped 5 times with ticks: 10
Handling....Next interrupt time: 240   Now: 240

*** Thread 1 looped 3 times with ticks: 10
*** Thread 1 looped 4 times with ticks: 20
Handling....Next interrupt time: 270   Now: 270
CurrentThread used up TimerTicks
*** Thread 1 looped 5 times with ticks: 10
No threads ready or runnable, and no pending interrupts.
```

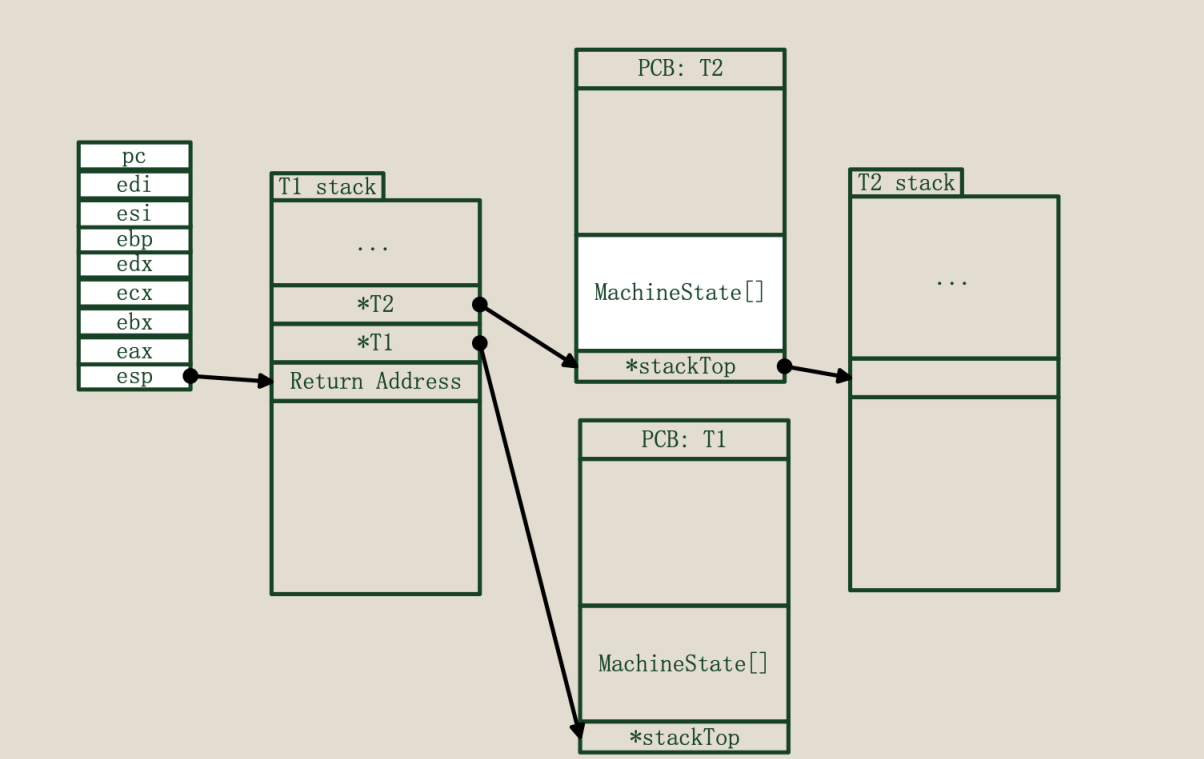
可见每个线程运行到时间片限制后切换，由于每次开关中断都调用`onetick`，所以有时候线程的实际时间片为20。

内容三 遇到的困难及解决方法

1. 线程是怎么结束的，通过向小组讨论找到`Thread::StackAllocate()`中将`currentThread->Finish()`放到了`machineState`中，在线程运行的最后自动调用函数来进入阻塞态并待释放。
2. fork中调用readytorun，本来把进程调度一起写在了里面，后来注意到进程调度里有yield函数，这样会使得fork中的关中断、开中断之间又有一次关开中断，所以放到了fork最后（开中断之后，调用scheduler来抢占）。
3. swtch.s看不懂的问题，在小组讨论时得到解答，参考链接：

   以及图片：





swtch.s T1 T2作为switch函数的参数，与switch函数在栈中连续。

4. 考虑：时间片应该作为进程类的一个变量，还是应该放在scheduler里？我认为应该放进程里/在scheduler里建映射，这样有利于之后拓展到阻塞后时间片没用完的情况。最后选择放在scheduler里，不过Linux里是放在进程里的。
5. threadToBeDestroyed bug：经同学提醒，发现`threadToBeDestroyed`是在system.cc开头定义的。如果当前仅有两个线程，其下一条语句都是`Finish()`，则：t1->Finish()->t1->Sleep()->scheduler->Run(t2)->t2->Finish()->t2->sleep()，由于t1没有把自己放到就绪队列中，两个线程都无法被delete。

内容四 收获及感想

动手实践毫无疑问大大提高了我对线程机制与调度的理解，与只是背记概念完全不同。非常感谢nachos的设计者以及软微的课程设置、陈老师的教导。

内容五 对课程的意见或建议

无，主要是一周只有一次课三小时，我觉得内容量已经很够讲了。

参考文献

《Linux内核设计与实现》

https://github.com/daviddwlee84/OperatingSystem

CFS中的虚拟运行时间vruntime

linux内核CFS进程调度策略

Linux下的进程控制块(PCB)

Linux进程调度-------O(1)调度和CFS调度器

Linux进程管理与调度总集

几个问题理解cfs