Lab2 线程调度 实验报告

Exercise1

- 对于linux中的普通进程来说，调度算法采取完全公平调度算法(CFS)，其根本思想是让每一个进程能够分配的到相同时间的CPU使用权，如果某个进程为IO密集型进程，其CPU使用时间较短，则调度时则会考虑让其有更高的优先级上CPU，以弥补其在CPU使用时间上的劣势。
- linux对于每个进程维护一个`nice`值和一个`vruntime`值，`nice`值代表进程的优先级，`nice`取值`-20`到`+19`，值越小代表进程的优先级越高。但是`nice`值确不会直接影响进程调度，是通过影响`vruntime`来影响进程调度，`vruntime`是进程在CPU的虚拟使用时间，其计算方式为`实际运行时间×NICE_0_LOAD/权重`，每个进程的`vruntime`会累加，在调度时选取`vruntime`值最小的进程上CPU。而`NICE_0_LOAD`代表nice值为0的权重，`权重`变量则依照不同的nice值对应至不同的至，进而按照程序的优先级对其占用CPU的时间进行对应的缩放变成`vruntime`，产生不同的优先级调度的效果。
- 对于实时进程，linux采用了两种调度方式，SCHED_FIFO和SCHED_RR，即简单的先进先出和时间片轮转算法。但是每个进程的优先级都高于普通进程。
- Windows 实现了一个优先驱动的，抢先式的调度系统——具有最高优先级的可运行线程总是运行，而该线程可能仅限于在允许它运行的处理器上运行，这种现象称为处理器亲和性，在默认的情况下，线程可以在任何一个空闲的处理器上运行，但是，你可以使用windows 调度函数，或者在映像头部设置一个亲和性掩码来改变处理器亲和性。也就是在一定程度上将进程和CPU绑定
- 参考链接：https://blog.csdn.net/deyili/article/details/6420034；https://blog.csdn.net/lenomirei/article/details/79274073；https://www.cnblogs.com/lenomirei/p/5516872.html

Exercise2

Nachos系统中，没有设置复杂的线程调度算法。其调度算法可以分为两种情况。

- 在命令行不添加`-rs 2`的情况下，其初始化全局变量时不会初始化时钟中断模拟器。

```
// system.cc
 if (randomYield)                // start the timer (if needed)
        timer = new Timer(TimerInterruptHandler, 0, randomYield);
```

这种情况下，在线程运行的过程中不会产生时钟中断，因此当系统中没有其他中断的情况下，cpu的使用权只能由线程自行放弃，无法被抢占和调度。

- 在命令行添加`-rs 2`的情况下，时钟模拟器会初始化，并且定期向待处理的中断列表中发送时钟中断。Nachos在每条指令执行前都会检查中断列表，当需要处理中断时，则调用`timer.cc:TimerHandler()`函数，该函数会向中断列表插入下一个时钟中断，然后调用注册的中断处理函数`system.cc:TimeInteruptHandler()`，该函数检查当前是否还有进程等待，如果有则调用`Thread.cc:Yield()`函数，该函数将当前进程加入等待队列并选取下一个进程进行调度，选取原则是先来先服务。其本质上也是一种时间片轮转算法

Exercise3 可抢占的优先级调度算法：

- 调度原理：对于每个进程来说，都具有相应的优先级，当优先级更高的进程就绪后，应该调度其上CPU而暂停当前进程的工作。
- 调度时机：当前的Nachos系统`interupter.cc:OneTick()`函数代表了一次指令执行，函数内会检查当前是否有待处理中断，通常的操作系统中断处理都意味着一次进程调度，当中断处理结束之后，会设置标志位`YieldOnReturn=true`，之后检查这个标志位来决定是否需要进行进程调度。对这里进行更改，增加需要进行进程调度的原因：如果有优先级更高的进程在等待的话。代码变更为如下：

```
/*检查是否有优先级更高的进程等待，有的话则发生调度*/
/*   interrupt.cc:  172   */
if (yieldOnReturn || scheduler->checkPriority(currentThread))
    { // if the timer device handler asked
        // for a context switch, ok to do it now
        yieldOnReturn = FALSE;
        status = SystemMode; // yield is a kernel routine
        currentThread->Yield();
        status = old;
    }
/*  scheduler.cc:checkPriority(Thread* curT)*/
/*检查是否存在优先级更高的进程等待 */
bool Scheduler::checkPriority(Thread *curT)
{
```

```
    if(this->scheduleMethod != PRIORITY){
        return false;
    }
    ListElement *first = readyList->getHead();
    ListElement *ptr;

    for (ptr = first; ptr != NULL; ptr = ptr->next)
    {
        if (curT->getPriority() > ((Thread *)ptr->item)-
>getPriority())
        {
            return true;
        }
    }
}
```

- 调度算法： 在`scheduler.cc : findNextToRun()`函数中，是真正的进程调度方法。期作用是选取下一个上CPU运行的进程，因此对该函数进行变更，针对不同的调度方式进行判定，采取不同的调度算法，变更为如下代码：

```
Thread *
Scheduler::FindNextToRun()
{
    switch(this->scheduleMethod){
        case PRIORITY:   // 优先级调度
        return priority();
        case RR:    //   时间片轮转调度
        return runtimeRound();
        case MULTIQUEUE:  //多级反馈队列调度
        return multiPriorityQueue();
    }

}
```

- 运行测试：在`TreadTest.cc`中编写了简单的线程测试函数，依次优先级从低到高创建线程，由于main函数使用默认的3（最低优先级，0为最高）为优先级，因此可以预见到，在创建完成优先级为2的进程后，main函数会被抢占，知道再次获得CPU使用权才能创建优先级为1的进程，代码如下：

```
void ThreadTest3(){
    Thread* t3 = new Thread("thread 3", 0, 3);  //优先级3
    t3->Fork(SimpleThread2, (void*)1);
    for(int i = 0; i > 30; i ++);

    Thread* t2 = new Thread("thread 2", 0, 2);// 优先级2
    t2->Fork(SimpleThread2, (void*)1);
    for(int i = 0; i > 30; i ++);

    Thread* t1 = new Thread("thread 1", 0, 1); //优先级1
    t1->Fork(SimpleThread2, (void*)1);
    for(int i = 0; i > 30; i ++);
}
```

- 测试结果：部分测试结果如下，可以看到main函数被抢占，之后在调度的过程中总是调度优先级高的线程，即使线程2主动放弃CPU，调度线程3上CPU，但是在运行时还是会切换回到进程2

```
Forking thread "thread 3" with func = 0x56561653, arg = 1
Putting thread thread 3 on ready list.
Forking thread "thread 2" with func = 0x56561653, arg = 1
Putting thread thread 2 on ready list.
Yielding thread "main"
Putting thread main on ready list.
Switching from thread "main" to thread "thread 2"
我的名字是 : thread 2, 我的用户是 : 0, 我的tid是:  2
...
...
Yielding thread "thread 2"
Putting thread thread 2 on ready list.
Switching from thread "thread 2" to thread "thread 3"
Now in thread "thread 3"
Yielding thread "thread 3"
Putting thread thread 3 on ready list.
Switching from thread "thread 3" to thread "thread 2"
```

Exercise 4 时间片轮转算法

- 实现原理：为每个线程设置时间计数，当进程使用完指定数量的时间时，就就强迫进程进入等待，调度下一个进程。
- 调度时机：这里依旧在上面提到的`interrupt.cc:  OneTick()`函数中增加线程调度的条件：当前线程时间用完，同时在函数开始时需要为当前的进

程时间加一：`curThread->addTick()`代码如下：

```
/*   interrupt.cc:  172   */
 if (yieldOnReturn || scheduler->checkPriority(currentThread) ||
scheduler->checkRunTime(currentThread))
    { // if the timer device handler asked
        // for a context switch, ok to do it now
        yieldOnReturn = FALSE;
        status = SystemMode; // yield is a kernel routine
        currentThread->clearTicks();  //清除运行时间
        currentThread->Yield();
        status = old;
    }
/*  scheduler.cc:  checkRunTime(Thread* curT)*/
/*检查是否运行完成时间片*/
bool
Scheduler::checkRunTime(Thread* curT){
    if(curT->getTicks() < this->RunTicks){  //RunTicks为调度器指定的
整数，
        return false;
    }
    return true;
}
```

- 调度算法：在上面已经看到，在`scheduler::findNextToRun()`函数中进行调度算法判定，之后跳转至具体的调度算法，时间片轮转则是FCFS算法，代码如下：

```
Thread*
Scheduler::runtimeRound(){
    return (Thread *)readyList->Remove();
}
```

Exercise5 多级反馈队列

- 算法原理：将等待队列分为多个级别，优先级越高的队列时间片越短，当进程使用完当前时间片时，将其加入下一级等待队列，如果没有使用完当前时间片，则将其加入下一个优先级的队列中，当优先级高的队列不为空时，应该抢占当前优先级低的进程。
- 调度时机：调度时机分别为当前时间片用完，出现中断处理行为，以及此算法特有的，更高级的队列上出现新的进程时，因此在上面优先级抢占算

法中进行逻辑调整，当当前策略为多级反馈队列时，应该检查优先级更高的队列是否为空。代码如下：

```
/*
    检查是否存在优先级更高的进程等待
*/
bool Scheduler::checkPriority(Thread *curT)
{
    if (this->scheduleMethod == MULTIQUEUE)
    {
        int l = threadListLeval[curT->getTid()];
        switch (l)
        {
        case 0:
            return false;
        case 2:
            return !readyList->IsEmpty();
        case 3:
            return (!readyList->IsEmpty()) && (!readyList2->IsEmpty());
        }
    }
    else
    {
        if (this->scheduleMethod != PRIORITY)
        {
            return false;
        }
        ListElement *first = readyList->getHead();
        ListElement *ptr;

        for (ptr = first; ptr != NULL; ptr = ptr->next)
        {
            if (curT->getPriority() > ((Thread *)ptr->item)->getPriority())
            {
                return true;
            }
        }
    }
}
```

- 实现细节：在算法实现上，主要需要主义在加入准备队列时，应该判断当前的进程是在哪个优先级队列中的，为此维护了一个列表`threadListLeval[]`，大小为最大进程数量，按照线程TID作为索引，标识其之前所在的队列，并且在加入准备队列时进行判断，代码如下：

```
void Scheduler::ReadyToRun(Thread *thread)
{
    DEBUG('t', "Putting thread %s on ready list.\n", thread->getName());

    thread->setStatus(READY);
    if (this->scheduleMethod == MULTIQUEUE)
    { //多级反馈队列
        if (checkRunTime(thread))
        {
            switch (this->threadListLeval[thread->getTid()])
            {
            case 0:
                readyList2->Append(thread);
                threadListLeval[thread->getTid()] = 2;
                break;
            case 2:
            case 3:
                threadListLeval[thread->getTid()] = 3;
                readyList3->Append(thread);
                break;
            }
        }
        else
        {
            switch (this->threadListLeval[thread->getTid()])
            {
            case 0:
                readyList->Append(thread);
                threadListLeval[thread->getTid()] = 0;
                break;
            case 2:
                readyList->Append(thread);
                threadListLeval[thread->getTid()] = 2;
                break;
            case 3:
                threadListLeval[thread->getTid()] = 3;
                readyList2->Append(thread);
```

```
                break;
            }
        }
    }
    else
    {
        readyList->Append((void *)thread);
    }
}
```

- 同时，由于多级反馈队列没个队列的时间片大小不同，因此检查是否使用完时间片的函数也需要变更，代码如下：对每个队列级别的线程分别按照不同的时间片大小进行检查

```
/*
    检查是否运行完成时间片
*/
bool Scheduler::checkRunTime(Thread *curT)
{
    if (scheduleMethod == MULTIQUEUE)
    {
        switch (threadListLeval[curT->getTid()])
        {
        case 0:
            return curT->getTicks() > this->RunTicks;
        case 2:
            return curT->getTicks() > 2 * this->RunTicks;
        case 3:
            return curT->getTicks() > 4 * this->RunTicks;
        }
    }
    else
    {
        return curT->getTicks() > this->RunTicks;
    }
}
```