Android OS Lab – 进程调度

一、 Background

移动数据网络对于手机来说是非常费电的。现在的智能手机可以在不使用网络时自动关闭网络连接。但我们认为这样还不够。我们希望网络连接的请求能够被重新组织，将零散的网络请求分为小组，每次同时执行一组网络请求。这样就能减少网络连接的次数，达到省电的目的。

在本次 Lab 中，我们需要实现一个内核同步的机制”net lock”，用一个控制器来阻塞那些请求网络通信的进程，并且在合适的时机释放它们。我们还要实现一个新的进程调度算法，使得释放的进程可以按照我们设定的机制进行调度。

Part 1: net-lock 同步机制

我们将提供一个新的内核同步机制，叫做"net-lock"。在这个机制下，netlock 会有两种不同的状态：(1)被网络休眠控制器持有，处于 sleep(关闭)状态，阻塞网络连接请求；(2)被零至任意多个网络应用持有，处于 use(开启)状态，释放并允许网络连接请求。状态的切换由用户空间的网络休眠控制器与网络应用同时作用。当网络休眠控制器想要关闭网络连接时，它请求持有并独占 netlock。当它打开网络连接时，释放 netlock。当应用程序需要使用网络连接时，它请求持有 netlock 并可以与其他进程共享。由于此时 netLock 可能被网络休眠控制器持有而处于关闭状态，应用程序在请求开启时还要给出一个时限(单位：秒)，代表它最长可承受的等待时间。网络休眠控制器必须在时限之内释放 netlock，从而允许网络访问。在网络应用请求 netlock 却还未获得之时，应用处于阻塞状态。当持有开启状态的 netlock 时，应用可以访问网络。当每个应用使用完网络后，同样要释放 netlock。网络休眠控制器同样可能在任何时刻请求 netlock 从而关闭连接，但这个请求必须阻塞直至当前没有任何处于等待或正在使用网络的应用为止。

Step 1:用读者-写者问题建立模型

这个问题可以用读者-写者问题来建立模型。netlock 是共享资源，休眠控制器可看做写者，因为休眠控制器对资源的占有是排他的。网络应用可看做读者，多个网络应用可同时持有 netlock。分析题意，我们可以发现这是一个读者优先的读者-写者问题。

Step 2: 内核同步机制：spin_lock 与 wait_queue 的相关知识

(1) spinlock 自旋锁

自旋锁与互斥锁有点类似，只是自旋锁不会引起调用者睡眠，如果自旋锁已经被别的执行单元保持，调用者就一直循环在那里看是否该自旋锁的保持者已经释放了锁，"自旋"一词就是因此而得名。

由于自旋锁使用者一般保持锁时间非常短，因此选择自旋而不是睡眠是非常必要的，自旋锁的效率远高于互斥锁。

spin lock 常用的 API 有：

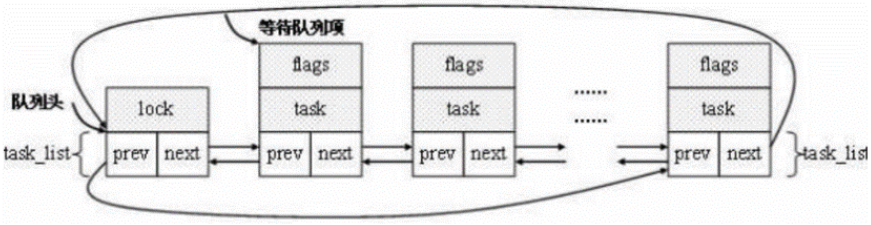
DEFINE_SPINLOCK(x)：该宏声明一个自旋锁 x 并初始化它。

spin_lock(lock)：该宏用于获得自旋锁 lock，如果能够立即获得锁，它就马上返回，否则，它将自旋在那里，直到该自旋锁的保持者释放，这时，它获得锁并返回。总之，只有它获得锁才返回。

spin_unlock(lock)：该宏释放自旋锁 lock

(2) wait queue 等待队列

等待队列用于使得进程等待某一特定事件的发生，无需频繁的轮询，进程在等待周期中睡眠，当时间发生后由内核唤醒。



Note:我们没有使用常用的同步机制 semaphore，这是因为 semaphore 一般需要在同一个进程上下文中操作，而我们的请求与释放不在同一个进程上下文中。

Part 2:

part2 的任务主要是结合 net_lock 编写相应的调度策略。对于 net_lock，我们需要添加一个 EDF 策略。EDF 策略是截止时间最早的进程被最早执行的进程调度策略。我们需要添加一个 EDF 调度的类（sched_edf_class）。这个调度策略需要和 linux 内核中的其它调度策略同时存在。为此我们需要增加一个新的调度策略 SCHED_EDF，我们将其编号为 6。只有调度策略为 SECHED_EDF 的进程才用 edf 策略进行调度。这个进程调度策略需要一个更高的优先级（高于默认的 CFS 策略，但是不高于 SCHED_RR 和 SCHED_FIFO 策略）。进程只有在等待 netlock 或者处在 netlock 的使用状态时才会被 EDF 策略调度。一旦进程从 netlock 状态被释放以后，进程应当被放回原来的进程调度队列中。另外，sleep netlock 的持有者不应该被这种进程调度影响。为了防止 EDF 策略的调度似的其它调度策略下的进程处于饥饿状态，在执行 schedule 函数的时候 EDF 策略下的进程应只有 80%的可能性被调度到。

P.S. Part2 的要求为简单起见，我们的调度算法仅仅要求在进程的 ddl 到来前，能够及时开启该进程的 netlock 即可。并且目前对于留有百分之 20 的时间给其余进程也不做要求。不考虑多核 CPU 的情况。

主要步骤：新建 EDF 调度类的数据结构

Hints：1. 参考 rt.c, fair.c 等文件（kernel/sched 路径下）

2. 正确理解调度类(调度器)与调度策略之间的关系(思考何处注册调度策略号 6)

3. 适当建立响应的数据结构（比如在 kernel/sched/sched.h 中、include/linux.sched.h 中，对比实时调度的算法的数据结构）

4. 别忘记修改 Makefile

Final. 编写测试程序（这一部分还未完成，因此上述内容无法保证完整的正确性）

进程调度基本概念讲述

每一个 CPU 都对应了一个 rq，也就是说一个 rq 就是一个 CPU 的抽象。

DEFINE_PER_CPU_SHARED_ALIGNED(struct rq, runqueues);

在 linux 启动的时候（启动时，即 init 的时候，不太确定）操作系统会为每一个 CPU 建立一个实体 rq，并放入 runqueues 之后，每当有新的进程建立时，操作系统就会调用 sched_class 中的各种函数，去完成将进程放入各种就绪队列中的操作。我们需要进行的主要操作就是建立 edf_sched_class，并完成其中的各个函数。

额外说一点有关 pick_next_task()函数的事情，操作系统会按照以下的进程调度策略的顺序去依次查找该使用哪个函数去完成现在所需要的任务。stop_sched_class->rt_sched_class->fair_sched_class->idle_sched_class

相关数据结构

1. rq, /kernel/sched/sched.h 每一个 CPU 都对应了一个 rq，也就是说一个 rq 就是一个 CPU 的抽象

2. rt_rq /kernel/sched/sched.h

3. sched_rt_entity 进程调度单元

4. rt_sched_class 进程调度类

Part2 参考文献：

1. **http://blog.csdn.net/janneoevans/article/details/8125106**

   本博客从整体上描述了调度策略，调度类的概念，并形象的描述出就绪队列的概念。在整体有个把握的情况下，下面的工作会容易很多。

2. **http://blog.csdn.net/sunnybeike/article/details/6950191**

   这里大概描述了/kernel/sched/rt.c 里面的函数，即 RT 的调度算法。

3. **http://www.oenhan.com/task-group-sched**

   这里讲述了 sched_rt_entity, rt_rq 等