《openEuler内核编程》

课程讲稿

第四章 第7讲

调度

软件所制

第四章 第7讲 调度

**学时：**2学时

**教学目的：**系统学习调度

**PPT讲稿：**



今天的内容是第7讲，操作系统调度。



这一节分6个部分，我们将主要介绍时钟定时器、抢占调度的概念，基本的OS调度机制，经典的进程调度算法，多核与多处理器调度，以及实时调度。



时钟中断是OS的心跳，周期性触发事件和工作的执行

大量的OS机制是时间驱动的，例如更新系统在线时间:uptime ；更新日历时间 ，SMP系统中对运行队列进行负载均衡 ，运行已到期的定时器 ，更新资源和进程统计信息

系统中断的频率:HZ.在编译时静态确定，早期系统为100，现在为250或1000。每秒产生HZ次时钟中断

HZ对系统的影响。HZ越大:计时精度更高，时钟中断处理开销越大

例如，进程只剩余2ms的时间片，若HZ为100,则可能10ms后 才退出执行;若HZ为1000，则可精确在2ms后暂停执行。

jiffies:Linux中的全局变量 ，标识系统启动至今的tick数量(时钟中断) ，每秒jiffies增加HZ次 ，系统启动至今的时间(秒数)=(jiffies-预设偏移)/HZ

这是jiffies相关的宏，用来判断时序的先后关系

时钟中断的处理例程，do\_timer更新系统时间等状态，update\_process\_times更新进程状态

Jiffies更新系统jiffies值

更新系统全局负载统计信息。并判断是否有到期的定时器需要运行，在scheduler\_tick中判断当前进程时间片是否用完，决定是否设置调度标志

调用schedule\_timeout函数前，讲当前进程状态设为不可中断。

设置该函数后，进入睡眠。

简单的monte carlo算法。

每次时钟中断发生时，判断当前状态在用户态还是内核态

如果在内核态，根据程序PC，判断当前的函数位置

时钟相关的系统调用包括获取时钟，设置和摧毁时钟。

好，讲完了时钟，我们介绍抢占管理。

抢占方式有两种，一种是被更高优先级进程抢占，另一种是时间片耗尽被抢占。

Linux 2.6以前的内核，抢占只能发生在用户态，即进程在内核态时无法被抢占

2.6及以后内核，进程运行在内核态时也可以被抢占 ，内核态可抢占可以降低用户态进程的分派延时(dispatch latency)



抢占进程需要判断当前进程的优先级，并设置重调度标记。

抢占的时机发生在中断异常返回点，当发现NEED\_RESCHED置位，进程调用preempt\_schedule\_irq进行调度



thread\_info中的preempt\_count大于0时，禁止抢占

禁止抢占的场景包括内核执行中断服务器例程(ISR)，执行软中断或tasklet时，其它设置了preempt\_count大于0的操作



抢占处理相关的内核操作包括开启、禁止等

19:

抢占控制函数，通过增加屏障原语，并进行计数标示

20:

若在禁止抢占期间出现了需要抢占的事件，则在抢占恢复后进行判断，并进行相应调度

21:

22:

OS调度的三个核心要素：抢占机制，调度顺序和运行时间片

23:

调度器的三个主要工作步骤:

保存当前进程/线程状态(PCB/TCB)

选择下一个待运行的进程/线程

分派(加载并跳转到相应PCB/TCB)

24:

对于调度过程中的资源分为可抢占资源和不可抢占资源。

不同类型的资源需要采用不同管理机制

25:

调度时机包括：进程/线程创建；进程/线程退出；I/O阻塞、同步；I/O中断；时钟中断

非抢占系统：当前进程主动放弃CPU时

可抢占系统：中断请求被服务例程响应完成时

当前进程被抢占： 进程时间片用完，进程从等待切换到就绪

26:

调度准则包括：最小等待时间、最大CPU利用率、最大吞吐率、最小响应时间、公平性

27:

相对于服务器，个人计算机的设计目标是什么?

追求平均响应时间 vs. 吞吐率

还是追求平均响应时间 vs. 公平性

响应时间是操作系统的计算延迟。吞吐率是操作系统的计算带宽

28:

接下来介绍经典进程调度算法。

29:

调度算法包括先来先服务算法、短进程优先算法、彩票调度算法、最高响应比优先算法、时间片轮转算法、虚拟轮转算法、多级反馈队列算法、公平共享调度算法

30:

先来先服务算法（FCFS，First Come First Served）依据进程进入就绪状态的先后顺序执行，一直运行到结束（早期），或者一直运行到阻塞或主动放弃CPU，主要用于非抢占式调度

如下面的例子，三个进程计算时间已知，不同的到达顺序，其平均响应时间不同

31:

优点是实现简单

缺点是平均响应时间波动较大，且I/O资源和CPU资源的利用率较低

CPU密集型应用会导致I/O设备空闲，I/O资源利用率低

32:

短进程优先算法 包括最短时间优先，和最短剩余时间有限，分别用于非抢占和抢占式调度

最短时间优先的例子，我们可以看到相较于先来先服务算法，短进程优先平均响应时间更短。

33:

选择就绪队列中剩余时间最短进程占用CPU进入 运行状态 。就绪队列按剩余时间来排序

34:

优点是平均响应时间短，但可能会造成饥饿 ，连续的短进程流会使长进程无法获得CPU资源

并且需要预知未来

操作系统如何预估下一个CPU计算的持续时间?

简单的解决办法:询问用户

用户不知道怎么办？

35:

一个通用的方法，用历史执行时间来预估未来的执行时间

36:

彩票调度算法的设计动机是因为SRTCF可以保证平均响应延迟，但是不公平，

给每个作业一定数量的彩票 ，随机抽取一张中奖彩票(winning ticket)；为了近似SRTCF，给短作业更多的彩票；为了避免“饥饿”，给每个作业至少一张彩票；相互合作的进程可以交换彩票

37:

最高响应比优先算法在短进程优先算法的基础上改进，不可抢占，关注进程的等待时间，防止无限期等待

选择就绪队列中响应比R值最高的进程

38:

时间片轮转算法和FCFS算法类似，但是增加了时间片。时间片结束时，调度器按FCFS算法切换到下一个就绪进程

轮转调度是抢占式调度

39:

如这个例子，时间片为20的时候，我们可以计算他们每个的等待时间，从而计算平均等待时间。

40:

因此，RR算法时间片长度选择很重要。

大时间片等待时间过长，极端情况下退化为FCFS

小时间片，响应时间快 ，产生大量上下文切换，影响系统吞吐

经验规则:选择一个合适的时间片，使上下文切换开销处于1%以内

41:

我们可以看到，同样是上面的例子，相同执行顺序，不同时间陪以及FCFS的调度后，其平均等待时间的差异。

42:

虚拟RR算法引入了人辅助队列。

I/O密集型进程会进入复制队列而不是就绪队列。

引入了优先级，使得辅助队列比就绪队列有更高优先级。

43:

这两种最基本的调度算法

FCFS先来先服务(First-Come,First-Served) ，早期OS中，待进程执行完毕再进行下一次调度；现在OS中，待进程阻塞或睡眠再触发下一次调度

问题是后来的短进程容易被前面的长进程长时间阻塞

RR轮转(Round Robin) 每个进程公平得到CPU时间配额q，分得1/n的CPU时间 ，进程的等待时间确定:(n-1)q

问题是

q过大,等同于FCFS

q过小，产生交织，带来过多的进程切换开销

典型进程的时间片:10ms – 100ms，典型的进程切换开销:0.1ms – 1ms，因进程切换带来1%的时间损耗

44:

进一步改进调度，引入多级反馈队列。

设计出发点:难以实现一种算法适用所有的情况，实际的场景需求远比两种基础算法复杂

多级队列:原就绪队列被划分区为多个队列。曾在Linux2.4版内核中采用，每个队列有自己的调度算法，交互式进程队列采用RR，后台进程采用FCFS；队列间采用RR,每个队列有其特定的优先级和时间片

进程自动在队列间循环移动，等待过久则提升至高优先级队列 ，独占CPU过久则降低至低优先级队列

45:

队列之间，每个队列分配一个优先级与相应时间片 ，队列间按照时间片调度

46:

进程可在不同队列间移动的多级队列算法。

时间片大小随优先级级别增加而增加 。进程在当前的时间片没有完成，则降到下一个优先级 。CPU密集型进程的优先级下降很快，I/O密集型进程停 留在高优先级

47:

公平共享调度（FSS，Fair Share Scheduling）控制用户对系统资源的访问

一些用户组比其他用户组更重要 ，保证不重要的组无法垄断资源，未使用的资源按比例分配，没有达到资源使用率目标的组获得更高的优先级

48:

接下来，我们介绍在多处理器下的操作系统调度

49:

现在我们的计算机已经从单核处理器，发展到多核处理器，甚至是众核处理器。因此，操作系统需要充分利用多核特性，优化调度算法。

50:

多核调度的设计问题 主要是进程/线程到处理器分配

一种是协同调度(co-scheduling) ，即一个进程的多个线程共同运行 ，或一个应用的多个进程共同运行

专用的处理器分配，则是线程会在一个专用的处理器上运行直到完成

51:

单队列多处理器调度，缺乏可扩展性，缓存亲和性弱

尽可能让进程在同一CPU上运行，保证缓存亲和性，可能会牺牲其他进程的亲和度来实现负载均衡

多队列多处理器调度时，每个CPU调度之间相互独立，避免单队列中数据共享及同步带来的问题。可扩展性强，具有良好的缓存亲和度（进程可以绑定在特定CPU上）

多队列多处理器调度程序应该如何处理 负载不均问题，从而更好地实现预期的调 度目标?

52:

负载不均衡的解决方式：进程迁移

关键问题:系统如何决定发起这样 的迁移?

一个基本的方法是采用一种技术，名为工 作窃取 (work stealing)

53:

Linux调度器历史演进

Linux V1.2采用循环队列,通过轮转(Round-robin) ，可以高效添加和删除进程,快速且简单

Linux V2.2引入了调度类(scheduling Class)概念 ，实现了三类调度策略 ，Real-time，Non-preemptilbe，Non-real time task

Linux V2.4将时间片切分成段epoch在每个epoch内，每个进程允许执行完时间片。优点是简单，采用类似权重的goodness函数为依据，选择下一个执行进程

缺点是调度选择复杂度为O(N)，效率不高，扩展性差不适用于实时系统

54:

V2.6.22: O(1)调度器

将调度算法的复杂度降低到了O(1) 。优点:扩展性好，适应多处理器系统

问题是对交互式应用响应差 ，实现复杂，公平性不够

V2.6.23z之后出现了CFS调度器

55:

2.6版本的O(1)调度器由Ingo Molnar设计实现

设计动机：为唤醒、上下文切换、定时器中断开销建立O(1)的调度器

进程有140种优先级，可用长度为140的array去记录优先级

56:

基于nice值的固定时间片轮转调度算法

改善交互性能 ,在高负载情况下仍能够快速响应 ,自动判断出交互进程并动态提高优先级，避免进程饿死

扩展性良好，O(1)的调度时间复杂度；负载均衡:当有负载时不会有CPU空闲；CPU亲和性:不会出现进程随机在CPU间切换

57:

静态优先级优先级取值范围[1,139]，值越低优先级越高；[100,139]保留给普通进程:SCHED\_NORMAL；[1,99]留给实时进程:SCHED\_FIFO、SCHED\_RR

静态优先级与时间片是直接对应关系(ms)

58:

动态优先级用于进程抢占的判断，高优先级总是比低优先级先执行

动态优先级在调度器选择新进程时计算，取值最大为100，因此其不可能超越实时进程的优先级。动态优先级值由静态优先级换算

其换算关系为：

Dynamic priority = max(100， min(static priority –bonus + 5, 139))

59:

两个优先级队列数组：活跃(Active)与过期( expired) 。一共140个优先级 [1,140] 。数字越小，优先级越高

问题是会造成进程饿死

解决办法是O(1)中的Active与Expired队列 。时间片用完后，从active转到Expired队列，让低优先级 进程有机会运行

60:

如图所示，包括两个队列，数字小的为优先级高的实时任务，而数字大的为一般的用户任务。

61:

传统调度进程算法是O(N)复杂度

在调度时，每个进程都需要被检查，耗时长

62:

O(1)调度器调度过程：从活跃队列数组中找到最高优先级的非空队列，从选择的队列中找到第一个进程 ，调整该进程的优先级，切换到该进程执行；当进程用完时间片，则将该进程插入过期队列数组中 的对应队列，并调用schedule进行下一次调度

优势是交换活跃数组和过期数组，给低优先级进程运行机会 ，防止饿死，时间复杂度低: O(1) ，只在开始运行和结束运行时涉及进程相关数据的访问

63:

在队列数组中查找最高优先级队列时间复杂度低:为O(1)，只依赖与优先级的数量，与系统中的进程数量无关

通过bitmap实现，用于快速查找，140个queue只需要5个integer进行表示，从中查找第一个非零位

硬件指令可以完成这种查找，如x86上的bsfl

64:

时间片即为task\_struct 数据结构中的time\_slice字段

高优先级的进程总是得到更大的时间片

task\_timeslice() in sched.c文件中，定义了计算时间片的方式。这个表给出了优先级对应的时间片

65:

时钟中断驱动调度器

O(1) 调度器支持多核/SMP ，在一定时间间隔后，进行 load balance 分析

计算发生在在每个时钟中断之后的服务例程

时钟中断中与调度相关的函数是Scheduler\_tick()

如果是SCHED\_FIFO,不做任何操作；如果是SCHED\_RR，则更新时间片；如果是SCHED\_NORMAL，则更新时间片

66:

O(1)轮转调度器的问题不易区分CPU受限和I/O受限，很难确定合适的时间片，将时间片直接对应优先级存在问题，优先级是相对的，但时间片却是绝对的

67:

2.6.23后内核的调度，新增了几个概念调度类(schedular class)、调度策略(scheduling policy)、优先级(sched\_priority)

调度类中再细分调度策略，实时类分为FIFO和RR

68：

这是调度类的结构体

69:

按类的优先进行调度，高优先级类优先调度。

70:

调度策略中，实时类包括两种FIFO和RR，普通类采用CFS调度算法，包括默认调度和批处理调度

设置进程调度策略的系统调用是sched\_setattr

71:

SCHED\_FIFO调度策略是抢占策略会抢占任何采用SCHED\_OTHER、SCHED\_BATCH等非实时策略的进程

优先级(sched\_priority)取值范围[1,99]，值越大优先级越高

SCHED\_FIFO进程没有时间片的概念，只会被更高优先级的进程抢占

72:

SCHED\_RR是对SCHED\_FIFO的一个简单升级

所有SCHED\_FIFO的机制都适用于SCHED\_RR，除了每个SCHED\_RR进程只能运行有限的时间片，时间片一般为100ms

73:

非实时调度策略中，SCHED\_OTHER是缺省的Linux分时调度策略 ，适用于所有不要求实时机制的进程

SCHED\_OTHER进程的优先级不参与优先调度决策，只影响衰减因子，后续会介绍

SCHED\_BATCH:批处理 类似于SCHED\_OTHER

假定进程是CPU密集型的，调度优先级略低于SCHED\_OTHER

74:

ps命令:可用于查看进程的调度策略

chrt命令:可查改修改实时调度策略与优先级

75:

完全公平调度器CFS在内核2.6.23中引入，在优先级的基础上，维护公平性(fair) ，为每个进程提供公平的处理器时间

建立一个理想的处理多任务CPU模型，N个进程，每个进程执行在1/N速度的CPU上

76:

其核心思想是动态调整时间片和执行顺序

不对每个进程采用固定的时间片，相反，在所有进程间采用固定时间片

不采用固定优先级选择下一个任务，将已运行时间作为调度选择的依据，选择最少运行时间的进程，等同于动态优先级

77:

CFS 定义了一个调度延时(sched\_latency ) ，又称为目标延时(targeted latency) ，近似于所有进程的总时间片:，获取和设置命令:为$sysctl kernel.sched\_latency\_ns

保证在调度延时内，每个进程至少可以得到一次执行。进程最大等待时间不超过调度延时，缺省调度延时为6ms

时间片的计算是根据进程权重占调度延时的比例确定

最小时间片粒度(sched\_min\_granularity)，缺省为0.75ms

当进程数过多时，若无法在缺省调度延时内为每个进程分配最小粒度时间，则调度延时调整

78:

这是调度延时和最小时间片力度初始化的代码，

可以看到，其默认调度延时为6ms，最小时间陪粒度为0.75ms

79:

每个进程都有自己的Nice值，取值范围[-20,+19]，值越小，优先级越高，缺省值为0

进程的Nice值越小，可获得的CPU时间越多

普通用户可以为进程分配大于0的nice值，Root用户可以分配小于0的nice值

任务获得的CPU时间由其Nice值权重的相对占比确定，不再像对应O(1)调度器中对应绝对时间 。高优先级任务不再长时间阻塞低优先级

80:

这是nice值和权重值对应数组，我们可以看到，每次nice level改变，对应进程就会减少10%的CPU时间。

81:

CFS时间片计算示例。

假设调度延时为20ms，其每个进程的份额为0.5，因此每个进程10ms时间片

82:

弱两个进程nice值不同，通过nice值对应权重，计算其份额，结合对应时间片

83:

CFS选择要运行进程的策略是选择当前最小加权运行时间的进程

我们引入虚拟运行时间vruntime，是由nice值进行加权的实际运行时间，因为加权，其实并不与实际运行时间一致。

84:

Vruntime的增加额等于当次实际cpu时间乘以衰减因子。每次进程退出执行时，对进程的vruntime变量加上相应增加额。衰减因子越小，实际的vruntime时间越少

85:

这是衰减因子的计算示例。

86:

CFS调度器为每个进程分配一个vruntime，用于表示进程已经运行的时间

87:

采用CFS调度器后，会出现进程饿死现象吗？

不会，长时间得不到执行的进程vruntime值将不再增加

88:

问如何有效找出最小的vruntime值?

调度需要高效实现这个查找 ，简单遍历所有进程将是低效的O(N)复杂度操作

89:

因此，通过红黑树对进程vruntime值进行管理。

以vruntime作为键值进行排序 ，插入、删除、更新的时间复杂度是O(lgN)；查找最小之值的复杂度是O(1)；min\_vruntime 用来缓存最小vruntime

更新vruntime和min\_vruntime是在每次新任务加入或删除时，以及每个时钟周期和每次进程切换

90:

Vruntime不再需要专门的优先级队列，直接通过RB树对进程进行管理

91:

新进程被唤醒时，调用该函数检查是否可以抢占当前进程

92:

弱可抢占，则调用resched\_task进行重新调度

93:

判断当前运行进程的vruntime，若大，则可以抢占

94:

我们总结下CFS调度器，其在系统负载极重时，也能保证确定的响应时间和运行时间 ，低优先级任务也能得到关注和执行。可以自动识别I/O密集和CPU密集，I/O进程被唤醒时自动优先得到调度

CFS设计简洁，性能够用Olog(N)

95:

BFS是一种时间片轮转算法的变种，在多处理器情况的单就绪队列（双向链表）选择，增加了队列互斥访问的开销，但减少了负载均衡算法开销

96:

BFS就绪队列的线程优先级一共103个

其中100个静态的实时优先级，3个普通优先级 SCHEDISO（isochronous）、SCHEDNORMAL和SCHEDIDLEPRIO（idle priority scheduling）

所有CPU共享一个双向链表结构的单就绪队列，所有线程按优先级排队，相同优先级的每个线程有一个时间片长度和虚拟截止时间

97:

虚拟截止时间是一个就绪队列中线程等待CPU最长时间的排序，不是真实的截止时间；线程时间片用完，重新计算VD，插入就绪队列；事件等待结束时，VD保持不变，以抢先低优先级线程或插入就绪队列

不同CPU对线程的VD加一个权重——从而让线程在上次运行的CPU上执行，保证缓存亲和度

98:

Linux多处理器调度每个CPU一个runqueue。为防止出现负载不均衡，需要对运行队列进行再平衡

内核线程Migration thread负责将进程从一个队列移动至另一个队列，为防止死锁内核按相同的顺序对runqueues进行加锁

当出现某个队列没有可运行的任务时，会主动调用 load\_balance

Load\_balance 也可以通过定时器周期性被调用

当系统空闲或者每隔100ms时，每个tick都会被调动

99：

每个进程都有一个bitmask位表示其能够运行的CPU ，缺省情况下是所有CPU

进程可以改变bitmask ，子进程从父进程继承该掩码，父子进程运行在同样的CPU 上

负载均衡并不改变亲和性不能将进程调度至掩码以外的CPU上

我们可以尝试使用taskset个numactl命令

100:

好，我们最后介绍一下实时调度

101:

实时调度分为两种类型的实时，硬实时和软实时。

102:

系统延时会影响实时保证性，包括中断时延和派发时延。

中断时延:硬件中断触发到开始执行中断服务例程的时间

派发时延:选择并切换至对应进程执行的时延

我们需要保证优先级高的任务得到优先执行，同时保证硬实时进程能在deadline时间前完成

103:

每个进程会对应一个优先级，总是选择最高优先级的就绪进程(可能会阻塞)，在相同最高优先级的进程间进行轮转，可能会抢占或被抢占

动态优先级由操作系统调度。如果在就绪队列中等待过久则提升优先级

104:

高优先级的进程可能会依赖低优先级进程

中优先级的任务进程出现，会发生什么?

105:

解决方案：优先级继承（Priority Inheritance），继承其等待进程中的最高优先级，对各个进程的优先级进行链表管理。依赖关系解除时，进程可以恢复到原始状态 。占用资源的低优先级进程继承申请资源的高优先级进程的优先级。只在占有资源的低优先级进程被阻塞时,才提高占有资源进程的优先级

106:

占用资源进程的优先级和所有可能申请该资源的进程的最高优先级相同

不管是否发生等待,都提升占用资源进程的优先级

优先级高于系统中所有被锁定的资源的优先级上限，任务执行临界区时就不会被阻塞

这就是优先级天花板协议

107:

硬实时:必须满足截止完成期限

进程特性是其周期性需要得到执行

108:

单调速率调度适用于周期性进程，基于进程周期静态设定优先级。优先级按照周期倒数进行分配， 周期越短则优先级越高

109:

最早截止时间优先(EDF)

优先级根据截止时间确定 ：越早的截止时间，越高的优先级

进程在不同时刻可以有高低不同的优先级

并不需要进程有周期性，也不需要预先知晓所需要的CPU处理时间

是最优的动态调度算法，但需要知晓截止时间

110:

完