第 5 章 进程调度

OCR by savage@smth.org

5.1 简介

像内存和终端一样，CPU 也是一个共享的资源，系统中的许多进程都争用 CPU。操作系统必须决定如何在所有的进程之间分配 CPU 资源。调度器作为操作系统的一个组成部分，它决定在任一给定时刻哪个进程去运行，以及这个进程能运行多长时间。从本质上讲，UNIX 是一个分时系统。也就是说，UNIX 允许多个进程并发执行。从某种程度上讲，这只是一个假象(至少对于单处理器来说是这样)。因为在任一给定的时刻在一个处理器上只能有一个进程在运行。UNIX 操作系统通过在分时的基础上交替执行进程来模拟并发。调度器把 CPU 分配给每一个进程一小段时间，接着就切换到另一个进程上。这个小的时间段就叫做时间段或时间片。

对 UNIX 操作系统调度器的描述必须集中在两个方面。第一个就是策略，也就是说决定哪个进程运行及什么时候切换到另一个进程的规则；第二个是实现，就是实现这种策略的数据结构和算法。调度的策略必须满足几个目标——对交互式应用程序的快速响应时间，对后台作业的高的吞吐率，避免进程的饿死等等。这些目标经常互相冲突，因此调度器必须平衡协调这些目标，同时必须高效地用最小的花费实现它的策略。

在最底层，调度器安排处理器从一个进程切换到另一个进程，这个过程叫做上下文切换。内核把当前进程执行的有关硬件上下文保存在进程的进程控制块(processcontrolblock： PCB)中。传统的 PCB 是进程 u 区的一个部分，保存的上下文有进程的通用寄存器值、内存管理寄存器和其他特殊寄存器值。接下来，内核把另一个要运行的进程的上下文加载到硬件寄存器中(上下文的内容可以从进程的 PCB 中获得)。这样 CPU 就可以从已经保存的上下文开始执行这个进程了。调度器主要负责决定什么时候执行上下文切换和让哪一个进程去运行。

上下文切换花销很大。除了保存进程寄存器值的一份拷贝外，内核必须执行许多和系统体系结构相关的任务。一些系统中，内核必须刷新数据高速缓存、指令高速缓存和地址转换高速缓存来避免新的进程的不正确的内存访问(见 15．9—15．13 小节)。新的进程开始执行时，将发生多次内存访问。因为访问内存的速度明显地比访问高速缓存的速度要慢，这样就降低了进程的性能。最后在精简指令集合计算机(RISC：ReducedlnstructionSetComputers)那样的有流水线(pipeline)的体系结构中，内核必须在上下文切换前刷新指令流水线。这些因素可能不但影响调度器的实现，而且要影响调度的策略。

本章首先描述时钟中断和基于定时器的任务的处理。因为当一个执行进程的时间片用完时调度器总要抢占它的 CPU，所以时钟对于调度器的操作非常关键。本章的余下部分将介绍不同的调度器的设计以及它们如何影响系统的行为特性。

5.2 时钟中断处理

每一个 UNIX 机器都有一个硬件时钟，它在固定的时间间隔发生一次中断。一些机器中，时钟中断发生后需要操作系统来整理时钟。而在另一些机器中，时钟能够重新整理自身。两次连续的时钟中断之间的时间段叫做一个 CPU 时标(CPU tick)或时钟时标(clock tick)，或简单地就叫做一个时标。大多数的机器支持许多不同的时标值。典型地 UNIX 把时标值设置为 10 毫秒(1)。大部分的 UNIX 实现中，把时钟频率或每秒的时标数保存在一个叫做 HZ 的常数中，这个常数在文件 param．h 中定义。一个时标为 lOms 的系统，HZ 的值为 100。内核函数通常以时标数来测量时间，而不是用秒或毫秒。

中断处理是和系统有很大关系的。本节讲述在许多传统的 UNIX 系统中中断处理的普通实现。时钟中断处理程序响应硬件时钟中断。硬件时钟中断的优先级仅次于电源失效

(power・failure)的中断优先级。因此，时钟中断处理程序必须尽可能的快，并且处理的任务要尽可能的最少。它执行下列任务：

。在需要的时候重新整理硬件时钟。

。对当前运行的进程更新 CPU 的使用统计。

。执行有关调度的函数，如优先级重新计算和时间片超时处理。

。如果当前进程超过了分配给它的 CPU 时限，就向该进程发送一个 SIGXCPU 信号。

。更新日(time-of-day)时钟和其他有关的时钟。比如，SVR4 维护一个叫做 lbolt 的变量来保存自从系统启动以来已经过的时标数。

。处理调出队列(calloutqueue)(见 5.2.l 小节)。

。在恰当的时候调用系统调用。如对换进程(swapper)和页面管理进程(pagedaemon)。

。处理报警(alarms)(见 5.2.2 小节)。

这些任务中没有必要在每一个时钟时标都进行处理。大多数的 UNIX 系统定义一个主时标(major tick)。它是 n 个时标，这里 n 和指定的 UNIX 系统有关。对于一些任务，调度器只是在主时钟时标时执行它们。例如，4.3BSD 在每四个时标时执行优先级的重新计算，SVR4 中每秒 1 次处理报警和唤起系统进程。

5．2．1　调出链表

调出(callout)函数记录那些内核在晚一些时候要执行的函数。比如，在 SVR4 中，任何内核子系统都可以通过调用 timeout 来注册调出(callout)函数。

Int to_ID = timeout(void(* fn)().caddr_t arg，long delta);

其中，fn()就是要执行的内核函数，arg 是要传递给 fn()的一个参数，delta 是以 CPU 时标计算的时间间隔，过了 delta 时间后，函数 fn()就要给引发执行。内核是在系统上下文中调用调出函数。返回值 to_ID 可以用来取消调出函数的执行，使用：

① 这个值并不统一，依不同的 UNIX 变体而不同，它还依赖于系统硬件时钟的分辨率。

void untimeout(int to_ID);

调出函数可以用于处理不同时间间隔的任务，如：

・重新发送网络包。

・某些调度器和内存管理函数。

・监测外部设备防止中断的丢失。

・轮询那些不支持中断的设备。

调出函数被认为是正常的内核操作，并且不能按中断优先级进行执行。因此，时钟中断处理程序不直接调用调出队列中的函数。每一个时标，时钟处理函数检查是否有一个调出函数到时。如果有这样的函数到时，时钟处理函数对它设置标志位，表示这个调出函数必须要执行。系统返回到基本中断优先级时检查这个标志位，如果它已经被置位，系统将执行这个调出处理函数。调出处理函数将唤起执行所有到时的调出函数。因此，一经到时，调出函数将会尽可能快的执行，条件就是所有投递出挂起中断都已经被处理完(2)。

内核维护一个调出函数的列表。因为系统中有几个挂起调出函数，因此，调出列表的组织将影响系统的性能。因为调出列表在每个 CPU 时标以一个高优先级被检查，算法必须使用尽可能少的检查时间。把一个新的调出函数插入到调出列表中所需要的时间并不是很关键的，因为插入动作发生的频率远低于一个时标，而且是在低优先级下执行。

调出列表的实现有几种方法。在 4．3BSD[Leff 89]中使用的一种方法就是按照函数“触发”时间(time-to-fire)来排序。列表中每一项都记录了该函数要发生的时间与前一个调出函

数发生时间的差值。对每个时钟时标，内核对第一个调出函数的时间值进行递减。如果时间值为零，内核就要执行这个调出函数。其他到时的调出函数也要被执行。这个过程在图 5—1 中描述。

图 5・1　BSDUNIX 中的 Callout 实现

另一种方法是使用一个相似的有序列表，但是每一项需要保存绝对时间。使用这种方法时，每一个时标，内核比较当前绝对时间和列表中第一项的绝对时间。如果两者的值相等就

②　当主处理函数结束而同时又没有中断需要处理时，许多实现都对这一情况进行了优化。此时，时钟处理函数直接降低中断优先级并调用调出处理函数。

会执行调出函数。

两种方法都要维护一个有序的列表，如果这个列表很大，那么花销也将很大。一个可以替换的方法是使用一个时间轮(timing wheel)，它是一个固定大小的调出函数的环形数组。每一个时标，时钟中断处理程序把当前时间指针前进到数组中的下一个元素，时间指针走到数组的尾部后将返回到数组的头部。如果在队列中有调出函数存在，时钟中断处理函数检查每个调出函数的到时时间。新的调出函数插在队列中从当前开始的 N 个元素后，N 是以时标计算的函数的触发时间。

实际上，时间轮在到时时间的基础上散列(hash)调出函数。每一个队列中，调出函数可以是有序也可以是无序的。调出函数进行排序有利于降低处理非空调出队列的时间，但是增加了插入时间。[Varg 87]中描述了优化的方法，使用多个分层时间轮来最优化时钟的性能，

5.2.2　报警

进程可以要求内核在指定的时间量后向其发送一个信号，这和报警时钟很相似。有二种类型的报警——实时、统计(profiling)和虚拟时间。实时报警和实际的时间有关，通过信号 SIGALRM 来通知进程。统计报警测量进程执行的时间，用信号 SIGPROF 来通知进程。虚拟时间报警只监测进程在用户态下运行所花费的时间，用信号 SIGVTALRM 来通知进程。

BSD UNIX 中，通过系统调用 setitimer 允许进程请求任何一种类型的报警以及指定以微秒为单位的报警时间。在内部，因为 CPU 时标是内核所能提供的最高精度的时间表示方法，内核把这个以微秒为单位的报警时间转换为正确的 CPU 时标数。System 5 中，系统调用 alarm 要求是实时类型报警，所指定的时间必须为整秒数。SVR4 中增加了系统调用 hrtsys，它提供了一个高精度的时钟接口，允许以微秒为单位指定报警时间。这样，可以通过把 setitimer(还有 getitmer，gettimeofday 及 settimeofday)作为库例程实现来和 BSD 兼容。同样地，BSD 提供了库例程 alarm。

高精确度的实时报警并不表示高的准确度。假设用户需要一个实时报警在 60 微秒后发出叫声，时间到达后，内核立即向调用进程发送一个信号 SIGALRM。但是，这个进程直到被调度为运行进程时它才能看见这个信号和处理这个信号。这将带来一个非常严重的延迟，它和信号接受者的调度优先级以及系统中的活动进程的数目有很大关系。对于很少有调度延迟的高优先级的进程来说，高精确度的时钟是很有帮助的。当前进程在内核态下运行并且还没有到达一个可抢占点时，这些进程被暂时阻塞。这些概念将在 5．5．4 小节中进一步描述。

统计和虚拟时间类型的报警不会出现类似的问题。因为它们和实际的时间没有关系，它们的准确程度受另一个因素影响。时钟中断处理程序计算当前进程的整个时标，尽管也许多次使用部分时钟。因为这些报警所测量的时间反应该进程运行时发生的时钟中断的数目，从长远角度看这种方法很好，且是进程使用的好的时间指示器。然而对于任何单独的报警，

这些方法都不够准确。

5.3 调度器的目标

调度器必须公平地把 CPU 时间分配给系统中的所有进程。实际上，系统的负载加大时，每一个进程得到的 CPU 时间就变得少了，因此它们将要比在负载轻的系统中执行的慢。调度器必须保证只要全部负载在期望的范围内，每一个应用程序的执行都能达到可接受的性能要求。

典型的系统中，并发地运行几个应用程序，这些应用程序根据它们的调度需求和性能期望被大致的分为以下几类：

交互式　外壳程序编辑器等应用程序以及具有图形用户界面的程序需要不断地和它们的用户进行文互操作。这类应用程序使用大量的时间在等待键盘输入和鼠标操作等用户的输入。接收到输入时，必须快速的处理，否则用户就会认为这个系统是不响应的。系统需要减少平均反应时间和用户动作同应用程序完全反应的变化时间。这样用户就不能轻易地觉察到延迟。对于键盘输入和鼠标运动，可接受的延迟一般是 50—150 毫秒。

批处理式　软件编译和科学计算等活动，不需用户交互操作，常常被提交为后台运行作业。对于此类任务，衡量调度效率的就是在有其他活动存在的时候，这个任务的完成时间同在一个不活动系统中所需时间的比较。

实时　这是所有有临界时间的应用程序的集合。尽管有许多类型的实时应用程序，而且每一个都有自己的需求集合，但是它们有许多相同的特征。它们通常需要那种能保证响应时间边界的可预测地调度行为。比如，一个图像应用程序可能需要每秒显示固定数目的图像帧(fps)，那么这个应用程序关心较多的是每秒帧(fps)的变化，而不是只想获得更多的 CPU 时间。用户更希望一个稳定的 15fps 的恒定速度，而不是那种在 10fps，和 30fps 之间显著波动的平均为 20fps 的速度。

典型的工作站也许要同时执行许多不同类型的应用程序。调度器必须尽力平衡每一个应用程序的需求，它也必须保证那些内核函数如换页(paging)、中断处理和进程管理等当需要时能立即被执行。

一个运转良好的系统中，所有的应用程序必须持续地向前运行，任何一个应用程序不应该阻止其他应用程序的运行发展，除非是用户显式地允许它。进一步的讲，系统应该总是能够接收和处理交互式地用户输入，否则，用户将无法控制系统。

调度策略的选择对于系统满足不同类型应用程序的需求的能力有着深远地影响。下节我们回顾一下传统的(SVR3／4．3BSD 中)的调度器，它们只支持交互式作业和批处理作业。本章的其余部分将详细介绍在现代 UNIX 中的调度器，它们都对实时应用程序有一定的支持。

5.4 传统的 UNIX 调度

我们首先描述一下用在 SVR3 和 4.3BSD UNIX 等传统 UNIX 中的调度算法。这些系统的主要目标是在分时的和交互式的环境里，几个用户可以同时执行多个批处理作业和前台进程。调度策略的目的就在于保证低优先级、后台作业不饿死的情况下，提高交互式用户的响应时间。

传统的 UNIX 中的调度是基于优先级的。每一个进程有一个调度优先级，它随着时间变化。调度器总是选择有最高优先级的处于可运行状态的进程去执行。对于相同优先级的

进程，调度器使用可抢占的时间片去调度它们，同时根据这些进程的 CPU 使用模式来动态地改变它们的优先级。当一个高优先级的进程准备就绪可以去运行的时候，调度器抢占当前

进程的 CPU 使用权，而不管它是否使用完分配给它的时间片。

传统的 UNIX 是严格地非抢占式的。如果一个进程正在执行内核代码(由于系统调用或是中断)，它不能被强迫放弃 CPU 的使用权给一个高优先级的进程。正在运行的进程在阻塞到一个资源的时候会自愿地放弃 CPU 的使用，否则，当它返回到用户态时可能被抢占让内核以非抢占的方式运行解决了许多由于多个进程访问内核数据结构而产生的同步问题（见 2．5 小节)。

以下几小节将描述 4.3BSD 中调度器的设计和实现。SVR3 中的调度器实现和 4．3BSD 中的实现仅在几个小的方面有些差别，如某些函数和变量的名字。

5．4．1　进程优先级

进程的优先级可以是 0～127 之间的任一个整数值，数值上越小对应的优先级越高。0～49 之间的优先级是系统为内核保留的，用户态下的进程的优先级在 50～127 之间。Proc 结构中包含有关优先级信息的字段如下：

p_pri	当前调度优先级。
p_usrpri	用户态优先级，
p_cpu	最近 CPU 使用情况的度量。
p_nice	用户可控制的 nice 因子。

字段 p_pri 和 p_usrpri 以不同的方式进行使用。调度器根据 p_pti 来判断去调度哪一个进程。当进程在用户态下时它的 p_pri 值和 p_usrpri 的值一样。当阻塞在一个系统调用的进程醒来后，为了利于内核态处理而暂时提高它的优先级。因此调度器使用 p_usrpri 来保存当它返回到用户态下时必须赋给它的优先级值，p_pri 保存它的暂时内核优先级。

内核把进程可以阻塞在其上的事件或资源同一个睡眠优先级联系起来。睡眠优先级是一个内核值，因此它处在 0～49 之间。比如，终端输入的睡眠优先级为 29，而磁盘 I／O 操作的睡眠优先级为 20。当一个进程从阻塞中醒来时，内核把它的 p_pri 值设为它所阻塞的事件或资源的睡眠优先级。因为内核优先级要比用户优先级高，所以这些进程将要比那些执行用户级代码的进程提前被调度。这样就允许系统调用很快地完成，这样就可以很好地解决进程因为执行一个系统调用而封锁一些关键的内核资源的问题。

当一个进程完成系统调用后要返回用户态下时，它的调度优先级被设为它的当前用户优先级。这样可能使得这个进程的优先级低于另一个可运行进程的优先级。在这种情况下，内核将进行上下文切换。

用户态优先级依靠两个因素——nice 值和最近 CPU 使用情况。nice 值在 0—39 之间，缺省情况下为 0。增加 nice 值将降低进程的优先级。内核自动地赋予后台进程一个很高的 nice 值。只有超级用户能降低一个进程的 nice 值从而提高它的优先级。之所以这个值被称为 nice 是因为通过给不太重要进程增加 nice 值③，而有利于其他进程的调度，是对其他用户的一件好事(“ nice")。

分时系统要把处理器分配给相互竞争的应用程序，使它们得到差不多相同的 CPU 时间。这就需要监视不同的进程的 CPU 使用情况，并藉此来决定调度。字段 p_cpu 值是测量进程最近 CPU 使用情况的一个尺度。进程创建时，这个字段值初始化为 o。对每一个时标，时钟处理程序增加当前运行进程的 p_cpu 值，最大为 127。进一步地，每两个时标，内核调用一个叫做 schedcpu()的例程(通过一个调出函数调度)，它通过一个衰减因子(decay factor)来减少每个进程的 p_cpu 值。SVR3 中这个衰减因子固定为 1／2。4．3BSD 使用以下的公式：

decay＝(2*load_average)／(2*load_average+1)；

其中 Load_average 是上一秒钟内平均的可运行的进程数目。例 程 schedcpu()也使用下面

的公式来重新计算所有进程的用户优先级。

p_usrpri=PUSER 十(p_cpu／4)十(2*p_nice);

其中 PUSER 是用户优先级的基本线 50。

如果进程最近使用了大量的 CPU 时间，那么它的 p_cpu 字段的值将被增加，从而增大它的 p_usrpri 的值，而降低它的优先级。进程在被调度前等待的时间越长，那么衰减因子就会更多地降低该进程的 p_cpu，这将使它的优先级增加。这种机制防止了低先级进程饿死，这将更有利于 I／O 集中型(I／O bound)的进程而不利于和计算集中(compute bound)型的进程。如果一个进程花了很长时间去等待 I／O(如一个交互的外壳程序进程或一个文本编辑器)，它将一直保持高的优先级。必要的时候，它将很快地得到 CPU 时间，像编译器和链接器等集中在计算的进程拥有高的 p_cpu 值，因此它们在一个很低的优先级上运行。

分时进程的调度中，CPU 使用因子提供了一个公平的机制。基本的想法就是所有这些分时进程的优先级经过一段时间后保持在一个大致相同的范围内，这些进程的优先级根据它们最近的 CPU 使用情况在这个范围内上下波动。如果优先级变化太慢，那么启动时具有低先级的那些进程将会长时间得不到调度从而导致饿死。

衰减因子的作用就是提供一个对进程在整个生命周期内的平均 CPU 使用时间按指数的加权。SVR3 的公式给出了一个简单的幂平均值，当系统的负载增大时[Blac 90]它将对评估优先级产生一个副作用。这是因为在负荷很重的系统中，每个进程只分配给很少的处理器时间，这将让它的 CPU 使用情况一直维持在很低的水平而衰减因子又进一步减少它。这些 CPU 的使用情况对进程的优先级没有太大的影响，启动时有很低优先级的进程将不公平地被饿死。

4．3BSD 解决方法中，它让衰减因子依赖于系统的负载。当负载变大时衰减因子变小。因此，那些分配到 CPU 周期的进程的优先级会很快地变低。

5．4．2　调度器的实现

调度器维持一个叫做 qs 的数组，它由 32 个运行队列(run queues)组成(见图 5—2)。每一个队列对应 4 个相邻的优先级。因此，优先级 0—3 对应于队列 0，优先级 4～7 对应队列 1，依此类推。每个队列包含 proc 结构的双向链表的头部，全局变量 whichqs 包含位掩码，其中每一位代表一个队列。如果这个队列中有一个进程存在，那么位掩码的对应位就被置 1，还有就是只有那些可运行的进程被保存在这些调度器的队列中。

图 5-2　BSD 调度器数据结构

这就简化了选择哪一个进程去运行的任务。执行上下文切换的例程 swtch()通过检查全局变量 whichqs 去找到第一个置位所对应队列的索引。这个索引标识出含有最高优先级的可运行进程的调度队列。例程 swtch<)从这个队列的头部移去一个进程，并切换上下文为这个进程的上下文。当 swtch()返回时，这个新调度的进程开始执行。

上下文切换包括保存当前进程的寄存器内容{通用寄存器，程序计数器，堆栈指针，内存管理寄存器等)到该进程的进程控制块(pcb)中，pcb 是 u 区的一部分，接下来要加载保存的新进程的上下文到这些寄存器中，proc 结构的 p_addr 字段指向它的 u 区的页表项。swtch( )用这个指针来定位新进程的进程控制块(pcb)。

因为 VAX—ll 是 4BSD 和早期的 System 5 版本的参考目标模型。它的体系结构[DEC 86]对调度器的实现有很大的影响。VAX 有两个管理 32 位字段的特殊指令——FFS，或是发现

③　一般 nice(1)命令就用于此目的。它接收-20～19 的任一值(只有超级用户可以使用负值)。这个值累加上当前值作为新的当前值。

第一组(Find First Set)和 FFC(Find First clear)。这样就需要把 128 个优先级分解到 32 个队列。另外，VAX 中还有特殊的指令 INSQHI 和 REMQHI，能自动的从双向链表中插入或移去元素。其他指令如 LDPCTX 和 SVPCTX 能加载和保存一个进程的上下文。所有这些允许 VAX 仅使用少量的机器指令就能执行全部的调度算法。

5．4．3　运行队列管理

除非当前的进程在内核态运行，否则高优先级的进程总是要执行的。分配给进程的固定时间段(在 4．3BSD 中是 100ms)，只影响在同一个运行队列中的多个进程的调度，每个 lOOms，内核调用(通过调出函数)一个叫做 roundrobin()的例程来调度同一个运行队列中的下一个进程去运行。如果一个更高优先级的进程已经就绪可以执行了，那么它就可以优先地被调度，而不用等待例程 roundrobin()。如果所有其他的可执行的进程都在较低优先级的队列中，那么当前进程将一直运行下去，而不管它是否用尽了分配给它的时间段。

每秒钟例程 schedcpu()都重新计算每个进程的优先级。因为进程在运行队列中时它的优先级不能改变。schedcpu()把这个进程从队列中移走，改变它的优先级，然后再重新把它放回队列，可能放到和从前不同的队列中。时钟中断处理程序每四个时标重新计算当前进程的优先级。

上下文切换发生在下列三种情况：

- 当前进程在一个资源上阻塞或是退出时，这是一个自愿的上下文切换。
- 由于优先级的重新计算使得另一个进程的优先级高于当前进程的优先级。
- 当前进程或是一个中断处理程序，唤醒一个高优先级的进程。

自愿的上下文切换是直接的，内核直接从例程 sleep()和 exit()里调用例程 swtch()，非自愿上下文切换的事件的发生是因为系统处在内核态下，不能立即抢占当前进程，内核设置一个叫做 runrun 的标志位，用来标志有一个更高优先级的进程在等待被调度。进程要返回用户态时，内核检查 runrun 标志位，如果该位置“ l”，那么内核将控制权转移给例程 swtch ()，它将进行上下文切换。

5．4．4　分析

传统的调度算法是简单的有效的。对于既有交互式作业又有批出理作业的通用的分时系统已经足够了。动态地计算进程的优先级能够防止进程的饿死，这种方法更适合那些 I／O 集中型(I／O bound)作业，因为它们需要很少的而且不是很频繁的突发的 CPU 周期。

这种调度方法有几个局限性，所以它不能广泛地适用于商用操作系统中：它的外扩展性不好，如果系统中进程的数目太多，调度器不能在每秒钟都充分地计算所有进程的优先级。没有方法能保证部分 CPU 资源能分配给指定的进程或进程组。没有方法来保证有实时特征的应用程序的响应时间。应用程序不能很好的控制自己的优先级，nice 值机制简单，而且不够充分。因为内核是非抢占式的，这样就能发生优先级逆转(priority inversion)现象，即高优先级的可执行的进程可能等待相当长的时间才能被调度运行。

现代的 UNIX 系统用在许多环境中，特别地，对那种支持实时应用程序的调度器有很大的需求，要求更好的行为可预测性及时间响应的有界性，这就需要重新设计调度器。本章的余下部分将要详细介绍在 SVR4，Solaris 2．x 和 OSF／1 以及一些非主流的 UNIX 变体中的新的调度方法。

5.5 SVR4 的调度器

SVR4 的完全重新设计的调度器的特点是，它改进了传统的方法[AT&T 90]。它是为多种环境设计的，并提供了更大的灵活性和更多的控制。以下就是这种新的体系结构的主要目标：

·支持那些需要实时响应的多种应用程序。
·把调度策略同实现它的机制相分离。
·应用程序能够控制它的优先级和调度。
·定义了一个对于内核有良好定义接口的调度框架。
·允许新的调度策略以模块方式加入，包括调度器动态加载的实现。
·限制临界时间性应用程序的调度延迟。

系统要支持实时进程，这种需求驱动人们努力去寻找一个更好的方案，这种体系结构需要足够的通用性和强大的功能来处理许多种不同的调度需要。基本的抽象概念是调度类(scheduling class)，它用来定义这个类中的所有进程的调度策略。系统可以提供几个调度类，缺省的，SVR4 提供两个类——分时(time-sharing)类和实时(real-time)类。

调度器提供了一组和类无关的例程，它们实现一些公共的服务如上下文切换、运行队列的管理和抢占。同时它也定义了类相关函数的过程接口，如优先级重新计算和继承等。类实现这些功能的方法是不同的，比如实时类使用固定优先级，而分时类根据不同的事件来动态地改变优先级。

面向对象的方法和 vnode／vfs 系统(见 5．6 小节)及内存管理子系统(见 14．3 小节)中使用的方法相似。8．6．2 小节中给出一个用于现在 UNIX 系统中面向对象方法的概述。这里调度器代表一个抽象基本类，而每个调度类作为它的一个子类(派生类)。

5．5．1　类无关层

类无关层主要负责上下文切换，运行队列管理以及进程抢占。最高优先级的进程总是要优先运行(除非当前进程处在不可抢占的内核处理中)，优先级的数值被增加到 160，每个优先级有一个单独的调度队列。与传统的实现中不同的是，数值上越大则优先级越高。进程优先级的调度和重新计算是由类相关层来执行的。

图 5－3 描述了运行队列管理的数据结构。位图 dqactmap 指出哪一个调度队列中至少有一个可运行的进程。进程通过调用 setfrontdq()和 setbackdq()加入到队列中，通过使用 dispdeq()从队列中移走这些进程。这些函数可以在内核代码中调用，也可以由类相关例程调用。一般新的可执行的进程被放在运行队列的尾部，而因为时间片到时的被抢占的进程则放在运行队列的头部。

UNIX 系统用来处理实时应用的一个最大的局限性是，它的内核在本质上是不可抢占的。实时进程要求调度延迟很小，也就是尽量要减少进程变得可运行和进程实际开始运行之间的时间。如果一个实时进程变得可执行，而此时当前进程正在执行一个系统调用，那么在上下文切换发生之前将有一个很大的延迟。

为了解决这个问题，SVR4 内核定义了可抢占点(preemption point)。可抢占点是内核代码中这样的一些位置，内核的数据结构处在一个稳定的状态，并且内核马上就要开始长时间的、大量的运算。每次执行到可抢占点时，内核检查标志位 kprunrun。如果这个标志位被置，就表明已经有一个实时进程就绪要运行，内核就会抢占当前进程。这样就限定了一个实时进程在被调度前必须等待的时间界限④。宏 PREEMPT()检查标志位 kqrunrun，调用例程 pre

－

④　除了显式地提到是实时进程外，这个编码是类无关的。内核仅仅检查 kprunrun 这个变量来判断是否可以抢占进程。就当前来看，只有实时类设置这个标志，但是从长远看可能会有一些新的类可能需要内核抢占功能：

empt()来实际抢占进程。抢占点的一些例子如下；

·路径名分析例程 lookuppn()中，开始分析每一个单独地路径名元素之前。
·系统调用 open 中，如果文件不存在，而要创建它之前。
·内存管理子系统中，释放一个进程的页面前。

图 5-3　SVR4 分派队列

传统的系统中使用标志位 runfun，仅仅抢占那些要返回到用户态的进程。函数 preempt () 引起 CL_PREEMPT 操作，并进行类相关的处理，接下来调用 swtch()进行上下文切换。

swtch()调用 pswtch()来执行那些和机器无关的上下文切换部分，接下来调用底层的汇编代码来管理寄存器内容，刷新转换缓冲区等等。系统调用 pswtch()清除 runrun 标志位和 kqrunrun 标志位，选择最高优先级的可运行的进程，并把它从调度队列中移出。它更新位图 dqactmap，并把进程的状态设为 SONPROC(在处理器上运行)。最后，它更新内存管理寄存器，来映射新进程的 u 区和虚拟地址映射图。

5．5．2　调度类的接口

所有的类相关功能都通过通用接口来提供。但是，每一个调度类的通用接口的虚函数的实现方法都互不相同(见 8．6．2 小节)。因此这个接口不但定义了这些函数的语义而且定义了用来调度实现这个类的链接。

图 5-4 说明了类相关接口的实现。结构 classfuncs 是一个指针向量，其中的指针指向为任何类实现的类相关接口函数，全局的类表中每项代表一个类，这个项包括类名字，指向初始化函数的指针以及指向这个类的 class funcs 向量的指针。

进程开始被创建时，它继承它的父进程的优先级类。因此它可以通过系统调用 priocntl 转移到不同的类中，有关系统调用 priocntl 的描述见 5．5．5 小节。结构 proc 中有三个字段用来描述调度类：类表示符 ID　　它仅仅是全局类表的一个索引。指向进程所属类的 classfuncs 向量的指针。这个指针是从类表项中拷贝得到的。
第 5 章　进程调度 p_clproc　　指向和类相关的私有数据结构。一组宏把调用解析为通用接口函数并能调用正确的类相关函数

```
#define Cl_SLEEP(procp，clprocp，...)\
    (*(procp)->p_clfuncs->cl_sleep)(clprocp，...)
```

全局类表

图 5-4　SVR4 类相关接口类相关函数可以通过这种方式从类无关代码和系统调用 priocntl 访问。调度类决定优先级计算和属于该类的进程调度的策略。它决定它的进程的优先级范围，以及在什么条件下进程的优先级可以改变。它决定一个进程每次执行的时间片大小，这个时间片的大小可以是所有进程相同，也可以根据优先级的不同而不同。它可以是 1 个时标也可以是无限个时标。无限长的时间片适用那些必须按时完成的实时任务。

类相关接口的入口点包括：

CL_TICK　　从时钟中断处理程序调用——监视时间片，重新计算优先级以及处理时间片到时等等。

CL_FORK，
CL_FORKRET　　从系统调用 fork 中调用——CL_FORK 初始化子进程的相应类的数据结构。CL_FORKRET 可以设置 runrun 标志位，允许子进程在父进程之前运行。

CL_ENTERCLASS，
CL_EXITCLASS　　当一个进程进入或离开一个调度类时调用——分别负责分配和释放类相关的数据结构。

CL_SLEEP　　从系统调用 sleep()中调用——可能重新计算进程优先级。

CL_WAKEUP　　从系统调用 wakeprocs()中调用——把进程放入合适的运行队列；也可能设置 runrun 标志位或 kprunrun 标志位。

调度类决定每个函数要执行的动作，每个类可能用不同的方法来实现这些函数，这样就提供了一个十分灵活的调度方法。比如，传统的调度器的时钟中断处理程序在每个时标都对当前进程进行处理，并且每 4 个时标重新计算它的优先级。在新的体系结构中，处理程序简单地调用进程所属类的例程 CL_TICK，这个例程决定如何处理时钟时标。举个例子，对于实时类，它使用固定的优先级，而且不进行重新计算优先级。类相关的代码决定什么时候时间片到时，设置 runrun 标志位，进行上下文切换。

缺省地，160 个优先级被分为以下三个范围：

0～59　　分时类

60～99　　系统优先级

100～159　　实时类

接下来的几个小节将要讨论分时类和实时类的实现。

5．5．3　分时类

进程的缺省类为分时类。进程的优先级动态地改变，对于相同优先级的进程使用时间片轮流调度。内核使用一个静态的分配参数表(dispatcher parameter table)来控制进程的优先级和时间片。分配给进程的时间片取决于它的调度优先级，参数表定义了分配给每一个优先级进程的时间片大小。缺省情况下，进程的优先级越低，它所分配到的时间片越大。这好象有点不合常理(counter－intuitive)，但是因为低优先级的进程不能经常被调度执行，因此当它被调度执行时就分给它相对大一点的时间片。

分时类使用事件驱动调度[Stra 86]，而不是每秒都重新计算所有进程的优先级，SVR4 根据和进程有关的特定事件来改变进程的优先级。进程使用完分配给它的时间片后，调度器降低它的优先级。另一方面，如果进程阻塞在事件或一个资源上，或是使用很长时间才用完分配给它的时间片，SVR4 将提高进程的优先级。因为通常事件只影响单个进程，所以优先级重新计算是很快的。分配参数表定义了不同事件是如何改变进程的优先级的。

分时类使用一个 tsproc 结构来保存那些类相关数据。这些字段包括：

ts_timeleft　　在时间片中仍然剩下的时间。

ts_cpupri　　优先级的系统部分。

ts_upri　　优先级的用户部分(nice 值)。

ts_umdpri　　用户态优先级(ts_cpupri+ts_upri，但是不超过 59)。

ts_dispwait　　时间片开始计算后的时钟时间秒数。

进程从内核睡眠恢复执行的时候，它的优先级是内核优先级，这个优先级是由其睡眠条件决定的。它返回到用户态时，它的优先级从 ts_umdpri 中恢复。用户态优先级是 ts_upri 和 ts_cpupri 两者的和，但是它的值被限制在 0～59 之间，ts_upri 的范围是从-20 到+19，缺省值为 0。这个值可以通过使用 priocntl 来改变，但是只有超级用户能增加它。ts_cpupri 是根据下面介绍的分配参数表来调整。

参数表为类中的每个优先级设置一个表项。尽管在 SVR4 中，每一个类有一个分配参数表(对系统优先级也有一个分配参表)，但是每个表都有不同的形式。分配参数表对所有的类来说不是一个必需的表，新的类可以在创建时不使用这个表。对于分时类，表的每一项包含

以下字段：

ts_globpri	这个表项的全局优先级(对于分时类，它对应在表中的索引)
ts_quanturn	这个优先级对应的时间片。
ts_tqexp	当时间片到时后对 ts—cpupri 设的新值。
ts_slpret	睡眠后将返回用户态时 ts—cpupri 设的新值。
ts_maxwait	在使用 ts—lwait 前等待时间片到时的秒数。
ts—lwait	如果进程将使用超过 ts_maxwait 的时间才能用完它的时间片，用该字段值取代 ts—tqexp。

内核用两种方法来使用这个参数表。可以按当前 ts_cpupri 值为索引来访问手段 ts_tq-exp、ts_slpret 和 ts_lwait，因为这些字段提供了新的基于旧值的 ts_cpupri 值。还可以使用 ts_umdpri 为索引来访问字段 ts_globpri，ts_quantum 和 ts_maxwait，这些字段和所有调度优先级都有关。

表 5·1 列出一个典型分时类的参数表。为了看一看它是如何使用的，假设一个进程的 ts_upri 等于 14，ts_cpupri 等于 1，它的全局优先级(ts_globpri)和它的 ts_umdpri 都等于 15。当时间片结束后，它的 ts_cpupri 将被置为 0(因此，它的 ts_umdpri 被设置为 14)。但是，如果进程需要多于 5 秒钟用完它的时间片，那么它的 ts_cpupri 将被设为 11(因此，它的 ts_umd-pri 为 25)。

假设在时间片用完之前，进程进行系统调用从而阻塞在一个资源上。当它最后返回到用户态开始重新执行时，它的 ts_cpupri 被设为 11(从列 ts_slpret 得到)，同时 ts_umdpri 被设为 25，而不管它用多长时间用完时间片。

5．5．4　实时类

实时类使用的优先级范围是在 100～159 之间。这些优先级比任何一个分时进程甚至是内核态的进程的优先级都高。也就是说，实时进程要在任何内核进程之前调度。假设进程已经在内核态运行，这时一个实时进程变的可以执行了，内核不会立即抢占当前进程，因为那样会使系统变的不稳定，实时进程必须等待进程要返回用户态或者是它到达一个内核可抢占点才能抢占它。只有超级用户进程能进入实时类；通过调用 priocntl 来指定优先级和时间片。

实时进程具有固定的优先级和固定的时间片大小。唯一的可改变它们的办法就是通过显式地调用 priocntl 来改变优先级或是时间片大小。实时分配参数表非常简单——它仅保存缺省的对每个优先级的时间片，进程进入实时类时，如果它没有指定时间片，就使用这个缺省的时间片。实时类缺省参数表也给低优先级进程分配大的时间片。实时进程的类相关数据都存储在结构 proc 中，包括当前时间片，时间片中剩下的时间和当前优先级，

实时进程需要有界的分配延迟，同时需要有界的响应时间。这些概念在图 5—5 中解释。分配延迟是进程变得可执行和它开始执行之间的时间，响应时间是需要进程响应处理的事件的发生时间和响应处理之间的时间。所有这些时间都需要有一个在合理范围内的定义好的上界。

响应时间是中断处理程序处理事件所需要的时间、分配延迟时间及实时进程自身来响应事件所花时间三者之和。分配延迟时间和内核关系密切，因为传统的内核本身是非抢占式的，所以它不能提供合理的延迟边界，如果当前进程正进行复杂的内核处理，那么实时进程可能需要等很长的时间才能被调度运行。测试表明内核中的一些代码路径的执行可能需几个毫秒，这对于大部分实时应用程序显然是不能接受的。

图 5—5　响应时间和分派时延

SVR4 使用内核抢占点来把较长的内核算法分成小的、有界的工作单元(unit)。当一个

实时的应用程序可执行时，负责处理类相关唤醒处理的例程 rt_wakeup()就会设置内核标志位 kprunrun，当前进程(假设正在执行内核代码)运行到一个可抢占点时，如果内核检测到标志位 kprunrun 被设置，那么它就会把上下文切换到等待执行的实时进程。因此，实时进
程的最大等待时间被限定在两个可抢占点间的最长代码路径之内，这是一个可以接受的方案。

最后，我们必须指出任何对延迟界限作出的保证，只有当实时进程优先级是系统中所有可运行的进程中最高的时候才适用。如果在它等待的任一时刻，一个更高优先级的进程变的可执行了，它将被优先调度，那么延迟的计算就应该当那个更高优先级的进程放弃 CPU 后，从零重新开始。

5．5．5　系统调用 priocntl

系统调用 priocntl 提供了几种方法来管理进程的优先级和调度特性，它包括一组不同的子命令可用来执行许多操作如：

- 改变进程的优先级类
- 设置分时进程的 ts_upri
- 重新设置实时进程优先级和时间片
- 获得一些调度参数的当前值

这些操作的大部分都限于超级用户使用，因此许多应用程序不能执行它们。SVR4 提供第 5 章　进程调度了系统调用 priocntlset，来对一组相关的进程进行同样地操作。相关的进程可以是：

- 系统中的所有进程
- 在一个进程组或是会话中的所有进程
- 一个调度类中的所有进程
- 某一特定用户的所有进程
- 具有相同父进程的所有进程

5．5．6　分析

SVR4 用一种设计和行为特性完全不同的调度器取代了传统的调度器，它提供了一个非常灵活的方法，允许增加系统调度类，供应商可以根据他的应用程序的需要来剪裁他的调度器。系统管理员可以使用分配表得到更多的控制，他可以改变分配表的设置和重新编译内核，从而改变系统的行为，

传统的 UNIX 系统，每秒需要重新计算每一个进程的优先级。如果系统中的进程数目很多，那么完成重新计算将花费非常多的时间。因此，传统的算法对于具有数以千计进程的系统来说不具有良好可扩展性。SVR4 分时类根据和进程相关的事件来改变进程的优先级，因为事件通常只影响一个进程，新的调度算法将很快而且有高度的可扩展性。

事件驱动的调度明显地有利于那些集中 I／O 操作和交互式作业而不利于 CPU 集中的进程。但这种方法有重大缺陷。如果交互式用户的作业需要大量地计算，他会发现系统响应不是很良好，因为这种进程不能产生足够的升高优先级的事件来减弱使用 CPU 对它优先级的影响。同时，与事件有关的最佳的提高或减小优先级取决于整个系统的负荷以及在给定时间内运行的作业的特性。因此为了保持系统的高效性和响应能力，需要经常重新调整这些值。

增加一个调度类不需要访问内核源代码，开发者必须按下列步骤来进行：

1. 提供对每个类相关调度函数的一个实现。

2. 初始化一个 classfuncs 向量指向这些函数。
3. 提供一个初始化函数来执行开始任务如分配内部数据结构。
4. 在主要配置文件的类表中为这个调度类增加一个表项。主配置文件缺省地在内核编译目录的 master.d 于目录下，这个项包含指向初始化函数和 classfuncs 向量的指针。
5. 重新编译内核。

SVR4 中存在一个很严重的局限，就是它没有提供一个很好切换分时类进程到其他类进程的方法。调用 priocntl 局限于只能由超级用户来执行，而能把特定用户的 ID 或是指定程序映射到非缺省类的机制是十分有意义的。

尽管 SVR4 对实时任务的处理方法已经向前迈了重要的一步，但是仍然达不到理想的要求。它没有提供最终期限驱动的调度方法(5．9．2 小节)，对许多实时应用程序来说，两个可抢占点之间的代码路径太长。此外，真正的实时系统需要一个完全可抢占的内核。这些问题中的一些后来在 Solaris 2．x 中得到解决，Solaris 2．x 对 SVR4 调度器做了一些改善和增强，我们将在下节介绍 Solaris 中的调度机制。

SVR4 调度器存在的一个重大问题就是如何正确地协调一组混合应用程序，这是一项 UNIX 高级教程困难的工作。[Nict 93]描述了一个实验，其中并发地运行三个不同程序——一个交互式键入会话，一个批处理作业，一个电影图像演示程序。因为键入和电影图像演示都需要 X 服务器交互操作，这就使得实验变的很困难。

实验人员试了几种优先级和调度器的组合，分配给 4 个进程(三个应用程序加上 X 服务器)，但是找到一种能让所有的应用程序充分地向前运行的组合是非常困难的。比如，凭直觉应该以实时类来播放图像，但是这将是灾难性的，甚至连图像应用程序本身也不能很好地进展，这是因为播放图像所依靠的 X 服务器得不到足够的 CPU 时间。如果把播放图像应用程序和 X 服务器都放入实时类中，则能保证播放图像有较好的性能(在正确地改变 (tweaking) 它们的相对优先级之后)，但是交互式作业和批处理作业就几乎停止了，系统就不能响应鼠标事件和键盘事件。

通过仔细地实验，可能找到一个正确的优先级组合，分配给集合中的每个应用程序，但是这样设置只能针对特定的应用程序集合。系统的负荷是不停变化的，这样，随着时间或是负载的改变，人工地仔细协调各个应用程序的优先级成为不可能。

5．6 Solaris 2．x 调度的改善

Solaris 2．x 在几个方面改进增强了 SVR4 调度的基本的体系结构[Khan 92]。Solaris 是一个多线程的，对称的多处理系统。因此，它的调度器必须支持这些特性。此外，Solaris 对高优先级，有临界时间的进程的分配延迟进行了一些优化，这样 Solaris 的调度器就非常适用于实时处理。

5．6．1 抢占式内核

对于实时进程的有界延迟需求来说，SVR4 中的内核抢占点最多只能是一个折衷的解决方法。Solaris 2．x 的内核是完全抢占式的，它能够保证良好的响应时间，这是对 UNIX 内核的实际改变，并具有深远的影响。大部分的内核全局数据结构必须用正确的同步对象如互斥锁(互斥变量)或信号量来保护，尽管这是一个十分麻烦的任务，但这是多处理操作系统的一个最本质的需要。

另一个改变，就是通过使用特殊的内核线程来实现中断，这些内核线程使用标准的内核同步原语，并能在必要的时候，阻塞在某个资源上(见 3．6．5 小节)。这样，Solaris 可以不用提高优先级的方法来保护临界区(CR)，内核中只有很少的不可抢占的代码段，因此，高优

先级的进程在它可运行的时候就能很快地被调度运行。

中断线程总是以系统中的最高优先级运行，Solaris 允许动态加载调度类。如果加载了调度类，那么内核重新计算中断线程的优先级，保证它们仍然具有最高优先级，如果中断线程需要在一个资源上阻塞，那么它能在同一个处理器上重新开始。

5．6．2 多处理器的支持

Solaris 为所有的处理器维持一个单独的分配队列，但是一些线程(如中断线程)可能要限制在指定的处理器上运行。处理器之间可以通过发送跨处理器中断来进行通信。每个处理器有如下的调度变量，保存在每处理器的数据结构中：

cpu_thread	当前在处理器上运行的线程
cpu_dispthread	上次选择在这个处理器上运行的线程
cpu_idle	这个处理器上的空闲线程
cpu_runrun	为分时线程使用的抢占标志
cpu_kprunrun	为实时线程使用的内核抢占标志
cpu_chosen_level	将要抢占当前处理器上运行线程的线程优先级

图 5-6 描述了在各处理环境中的调度情况，在处理器 P1 上发生一个事件，使线程 T6 (优先级为 130)可以执行了。内核把线程 T6 放在分配队列中，并调用 cpu_choose()，找到有最低优先级线程运行的处理器(本例中为 P3 处理器)，因为它的优先级(100)比线程 T6 的优先级低，cpu_choose()标志这个处理器为可以抢占，并设置它的 cpu_chosen_level 为线程 T6 的优先级(130)，同时向它发送一个跨处理器中断。假设此时，在处理器 P3 处理中断和抢占线程 T3 之前，另一个处理器，假设为 P2，处理一个事件使线程 T7(优先级为 115)变得可运行了，现在 cpu_choose()将要检查处理器 P3 的 cpu_chosen_level，发现它是 130，也就是有一个更高优先级的线程将在这个处理器上运行。因此，在这种情况下，cpu_choose()将把线程 T7 放在分配队列上，而避免了冲突。

几种情况下，低优先级的线程可能很长时间阻塞高优先级线程的执行。这些情况的引起或由隐式调度(hidden scheduling)，或是由优先级反转(priority inversion)。下面我们将看到，Solaris 中解决了这些问题。

5．6．3 隐式调度

内核经常代表线程执行一些异步的任务，内核调度这些任务，而不考虑那些线程的优先级，这种调度方式叫做隐式调度(hidden scheduling)。它的两个例子是流(STREAM)服务例程(见 17．4 小节)和调出队列(callout)。

SVR4 中，无论何时，进程将返回到用户态时，内核将要调用一个例程叫做 runqueues()来检查是否有流(STREAM)服务在等待处理。如果有的话，内核通过调用正确的流模块服务例程来处理这个请求。因此当前进程(将要返回用户态的那个进程)就代表其他进程来处理这个请求。如果其他进程的优先级比当前进程的优先级低，那么这个请求就以一个错误的优先级处理，也就是当前进程被一低优先级的任务所阻塞而发生延迟

Solaris 是这样解决这个问题的，它把流(STREAMS)处理移到内核线程，内核线程运行的优先级比任何实时线程都要低。这样就产生了新的问题，一些流(STREAMS)请求是由实时线程发出的，因为这些请求是由内核线程处理的，它们运行的优先级比期望的更低。如果不改变流处理的语义的话，这个问题是不能得到满意的解决，而且它将是满足实时要求的一个障碍。

对于调出队列处理(见 5．2．1 小节)也存在一个问题。UNIX 在最低的中断优先级处理所有的调出函数。但是这个优先级仍然比任何实时的优先级要高，如果这个调出函数是由低优先级线程发出的，那么对它的处理可能阻止调度高优先级线程。早期 Sun OS 版本的测量中显示处理调出队列要花费高达 5ms 的时间。

为了解决这个问题，Solaris 通过调出线程(callout thread)来处理调出函数。它运行在系

图 5－6 Solaris 2.x 的多处理机调度

统的最高优先级，但是它比任何的实时优先级都要低。实时进程请求的调出函数被分别维护，并在最低优先级上调用，因此保证立即调度临界时间的调出函数。

5．6．4　优先级逆转

优先级逆转问题，首先是在[lamp 80]中提出。它指的是这样一种情形，低优先级的进程拥有高优先级所需要的资源，从而阻塞了高优先级进程的执行。这个问题有几种情况，先让我们看一些例子(图 5-7)。

最简单的情形就是低优先级线程 T1 只有一个资源 R，高优先级的线程 T2 需要它，那么 T2 就要阻塞等待 T1 释放资源 R。现在考虑相同的情形并加上线程 T3，线程 T3 的优先级在 TI 和 T2 之间(图 5-7(a))，假设线程 T2 和 T3 是实时线程，因为线程 T2 被阻塞，那么 T3 变为可运行的最高优先级线程，并且它要抢占 T1(图 5-7(b))。结果 T2 将一直阻塞直到

图 5—7　简单的优先级逆转

线程 T3 完成或是阻塞，接下来线程 T1 运行，并释放资源。

解决这种问题的技术叫做优先级继承或优先级借用(priority lending)。高优先级线程阻塞在一个资源时，它暂时把它的优先级转移给拥有资源的低优先级的线程。因此，在上面的例子中，线程 T1 继承线程 T2 的优先级，这样，它不会被线程 T3 抢占(图 5-7(c))。线程 T1 释放资源时，它的优先级返回为它的原来的值，线程 T2 就可以抢占它。线程 T3 只能在线程 T1 释放资源和线程 T2 运行完并释放了 CPU 后才能被调度。

优先级继承必须是可传递的，在图 5-8 中线程 T4 阻塞在线程 T5 占有的资源上，线程 T5 阻塞在线程 T6 占有的资源上，如果线程 T4 的优先级比线程 T5 和线程 T6 的优先级都要高的话，那么线程 T6 一定通过线程 T5 继承线程 T4 的优先级。否则，假设线程 T7，它的优先级大于线程 T5 和 T6 的优先级，但是小于线程 T4 的优先级，它将抢占线程 T6，并引起相对于 T4 的逆转。因此，线程的继承的优先级必须是直接或间接阻塞线程中最高的。

Solaris 内核必须维持有关被锁对象的状态来实现优先级继承。Solaris 的内核需要识别出哪一个线程是被锁对象的当前所有者，同时也要知道被阻塞线程等待的是哪个对象。因为继承是可传递的，内核必须能够从任何给定对象开始遍历所有的对象和在同步链(Synchronization chain)中阻塞的线程。下一小节将介绍 Solaris 是如何实现优先级继承的。

图 5—8　传递优先级继承

5．6．5 优先级继承的实现

线程有两个优先级——一个是全局优先级，由线程调度类决定的；还有一个就是继承优先级，由线程和同步对象间的交互决定的。除非是线程受益于继承的优先级，否则线程的继承优先级为 0。线程的调度优先高于继承优先级和其全局优先级。

线程阻塞在资源上时，它就调用 pi_willto()函数来传递它的优先级给直接或间接阻塞的

线程。因为继承是可传递的，pi_willto()传递调用线程的继承优先级，函数 pi_willto()从直接阻塞这个线程的那个对象开始，遍历该线程的同步链，这个对象有一个指针指向属主线程(当前拥有锁的线程)。如果属主线程的调度优先级低于调用 pi_willto()线程的继承优先级，那么属主线程就继承高的优先级，如果属主线程阻塞在另一资源上，那么它的线程结构含有指针指向相应的同步对象。pi_willto()沿着这个指针传递优先级到那个对象的属主线程。依此类推，当我们到达一个没有阻塞的线程或者一个没优先级逆转⑤的对象时，结束同步链遍历。图 5-9 遍历同步链。

考虑图 5—9，当前运行的线程是 T6，它的优先级是 110，它需资源 R4，线程 T5 占有资源 R4。内核调用 pi_willto()从 R4 开始遍历同步链，采取下列动作：

1. 线程 T5 占有资源 R4，它的全局优先级是 80，没有继承优先级。因为它小于 110，因此，设置线程 T5 的继承优先级为 110。
2. 线程 T5 阻塞在资源 R3 上，线程 T1 占有 R3，线程 T1 的全局优先级是 60，继承优先级是 100(通过 R2)，它也小于 110，因此提高线程 T1 的继承优先级为 110。
3. 因为线程 T1 没有被任何资源阻塞，中止遍历同步链并返回。

⑤ 这个算法称为传递闭包计算，并且遍历的链就形成了有向无环图。

图 5-9 遍历同步链

pi_willto()返回后，内核阻塞线程 T6，选择另一个线程来运行。因为线程 T1 的优先级刚刚被升到 110，那么它会立即被调度运行。图 5-10 示出上下文切换的情景。

图 5-10 优先圾继承

线程释放对象后，它通过调用 pi_waive()来递交(surrender)它的继承优先级。有时线程可能阻塞许多对象，那么它的继承优先级是它所继承的所有对象中的最高值。这个线程释放一个对象后，它要在其余的占有对象的基础上重新计算继承优先级。这种优先级的丢失可能造成线程优先级下降到另一个可运行的线程的优先级之下，这种情况下，前一个线程将被抢占。

5．6．6 优先继承的局限性

只有当我们知道哪个线程要释放资源的情况下，才能实现优先级继承，资源被单个已知线程占有时，也可能实现优先级继承。Solaris 2．x 提供了 4 种类型的同步对象——互斥、信号量、条件变量和读写锁(第 7 章详细解释这些原语)。使用互斥量时，属主是已知的。然而，对于信号量和条件变量，属主是不可决定的，不能使用优先级继承。很遗憾会这样，因为条件变量经常和互斥量一起使用来实现高层的同步抽象，某些条件变量有确定的属主。

当一个读写锁正在因写操作而锁上时，它只有单个已知属主。但是，读写锁可以同时被多个读者占有。写进程必须阻塞等待所有的读进程释放这个对象，这种情况下，这个对象没有单个的属主。因此它不可能拥有指向这个对象的所有属主的指针，Solaris 通过定义一个 owner-of-record 解决这个问题，owner-of-record 是获得读锁的第一个线程，如果一个高优先级的写线程阻塞在这个对象上时，线程 owner-of—record 将继承它的优先级。线程 owner-of-record 释放读锁时，可能有其他不可识别的线程仍拥有这个对象的读锁，而这些线程无法继承写线程的优先级。尽管这种解决方案有很大的局限性，但是仍然十分有利。因为，许多情况下，只有一个单独的读线程拥有这个对象的锁。

优先级继承减少了高优先级进程阻塞在低优先级进程占有的资源的时间，但是最坏情况下，延迟仍超出大多数实时应用的可接受的延迟。一个重要的原因就是阻塞链(blocking chain)可能无限制地增长。另一个原因是，高优先级进程在它的执行路径上有几个临界区，

它可能阻塞在每一个临界区上，这样总的延迟将会很大。这个问题在研究团体中得到极大的关注，已经提出一些可替换的解决方案，如 ceiling 协议[Sha 90]，它控制进程对资源的锁，以保证高优先级的进程对低优先级进程占有的资源的每次活动操作最多只能阻塞一次。尽管这限制了高优先级的进程的阻塞延迟，但是使用这种方法，使的低优先级的进程经常阻塞。这种方法需要对系统中所有进程的优先级情况和它们资源的请求情况有个了解。这些不足之处限制了这个协议只适用于小范围的应用。

5．6．7 Turnstitle

内核包括数百个同步对象，每个都对应一个必须保护的数据结构。这些对象必须维护大量的信息，例如阻塞在这个对象上的线程队列。为每一个对象都设一个大的数据结构是浪费的。虽然内核中有数以百计的同步对象，但是在任何一给定时刻仅有几个在使用。Solaris 使用一个叫做 turnstile 的抽象概念，它提供一个空间效率很高的解决方案。同步对象包含一个指向 turnstile 指针，turnstile 包括所有管理这个对象所需的数据，如阻塞线程队列，指向当前占有资源的线程的指针(见图 5－11)。turnstile 从一个池中动态分配，这个池是随着系统中分配的线程数目而动态变化。turnstile 由阻塞在对象上的第一个线程分配，如果没有线程阻塞在那个对象上，turnstile 释放回池中。

传统的 UNIX 系统中，内核把进程可能阻塞的资源或事件同一个特定的睡眠通道(sleepchannel)联系起来。典型地，这个通道和资源或事件地址相关。内核在等待通道的基础上散列这些进程到一个睡眠队列，因为不同的等待通道可能映射到同一个睡眠队列上，遍历

图 5－11 旋转门队列

所花时间只能由系统中所有线程的数目决定的。Solaris 2．x 用 turnstile 代替这种传统的机制，turnstile 限制只有阻塞在资源上的线程才能排到资源的睡眠队列上，因此提供了更合理的处理队列的延迟时间。

线程在 turnstile 中按优先级排队，同步对象支持两种解锁行为———信号⑥，它唤醒单独的阻塞线程；另一个是广播，它唤醒阻塞在资源上的所有线程。Solaris 中，信号唤醒队列中最高优先级的那个线程。

5．6．8 分析

Solaris 2．x 提供了一个复杂的环境来进行单处理器或多处理器的多线程实时处理。它解决了 SVR4 调度实现中存在的一些不足。对 Sparcstationl[Khan 92]的测量表明在多数情况下调度延迟低于 2ms，这主要归功于完全可抢占的内核和优先级继承。

尽管 solaris 提供了一个适用于实时应用的环境。但是，它主要还是一个通用的操作系统。完全为实时设计的系统将提供许多其他特点，如处理器的群调度(gang-scheduling)和 deadline 驱动，或对 I／O 设备的基于优先级的调度。这些调度方法将在 5．9 节中进一步介绍。现在让我们回顾其他用在商用和实验中的 UNIX 变体的调度算法。

5．7 mach 中的调度

Mach 是支持多线程多处理器的操作系统，它设计的目的是能够在各种类型的机器上运行的，从单处理器到大型的由几百个共享地址空间的处理器组成的并行系统。因此，它需一个能对各种目标机扩展良好的调度器[Blac 90]。

Mach 的最基本的编程概念——任务和线程在 3．7 节中已经介绍过了。线程是最基本的调度实体，Mach 调度线程而不必考虑它所从属的任务。这种调度方法牺牲了一些性能，如相同任务之间两个线程的上下文切换速度要比不同任务中的两个线程之间的上下文切换

快的多(因为不同任务情况下，内存管理映射需要改变)，这种策略有利于进程间的切换，但是

⑥ 这种信号行为完全与传统 UNIX 信号无关。随着 UNIX 从多个来源继承了各种技术，某些术语也会有类似的使用

可能引起使用目标和负载平衡之间的冲突。进一步地讲，两种类型的上下文切换在性能上表现的差距可能微不足道，这取决于使用的硬件和具体的应用。

线程从它所属的任务继承一个基本调度优先级，这个优先级同 CPU 使用因子结合起来。内核为每一个线程单独维护 CPU 使用因子。Mach 对不活动的线程每秒按因子 5/o 来衰减它的 CPU 使用情况。衰减算法是分布式的，每一个线程监视自己的 CPU 使用情况，当它从阻塞中醒来的时候，它要重新计算 CPU 使用情况。时钟中断处理程序重新计算当前线程的 CPU 使用因子，为了防止运行队列中低优先级的线程因为得不到机会重新计算优先级而饿死，内部的内核线程每两秒重新计算所有线程的优先级。

被调度的线程运行固定的时间片，时间片结束后这个线程可以被相同或高优先级的线程抢占。当前线程的优先级可能在初始时间片用完前降到其他可运行线程的优先级以下。Mach 中，这种优先级的降低并不会引起上下文的切换，从而减少了只相关于使用情况的上下文切换的数目。如果一个高优先级的线程变的可运行了，那么不管当前线程的时间片是否用完，它都可能被抢占。

Mach 提供一个特殊的调度方法叫做 handoff，线程可以直接把处理器给其他线程而不用搜索运行队列。进程间通信(IPC)子系统使用这种技术来传递消息——如果线程已经等待接收消息，那么发送线程直接把处理器给接收者。这就提高了 IPC 调用的性能。

5．7．1 多处理器的支持

Mach 支持大范围的硬件体系结构，从单处理器到由数百个处理器组成的大型并行系统。它的调度器提供了几种有效管理处理器的方法。

Mach 并不使用跨处理器的中断来抢占线程。假设处理器上发生的事件使得线程可执行了，而且这个线程的优先级比在不同处理器上执行的另一个线程的优先级高，但是在处理器处理一个时钟中断或是另一个调度有关的事件发生之前，后一个线程不会被抢占。缺少跨处理器的抢占不会降低分时行为的性能，但是它可能延长实时应用有效响应时间。

Mach 通过创建处理器集合让用户控制处理器分配，每个集合可能含有零个或多个处理器，处理器属于一个单独的处理器集合，但是它可以从一个集合转移到另一个集合。任务或线程被分配到一个处理器集合，同样地这种分配也可以随时间改变而改变，仅仅授权任务才能分配处理器、任务和线程到处理器集合。

线程可能只在分配集合中的一个处理器上运行。任务到处理器的分配建立了这个任务中新的线程缺省的分配集合。任务从它们的父任务继承这种分配，初始任务(initial task)分配到缺省处理器集合，缺省的处理器集合包括系统中所有的处理器，它必须至少包含有一个处理器，因为内部内核线程和 daemons 进程被分配到这个集合。

决定分配策略的用户级服务器程序(以一个特权身份运行的任务)分配处理器。图 5-12 描述了典型的应用程序，服务器和内核三者之间的交互。应用程序分配处理器集合并分配线程到这个集合，服务器分配处理器到集合。这些事件的顺序如下：

1. 应用程序请求内核分配一个处理器集合。
2. 应用程序为这个集合向服务器请求处理器。
3. 服务器请求内核为这个集合分配处理器。

4. 服务器应答应用程序，指出所请求的处理器已经被分配。
5. 应用程序请求内核为这个集合分配线程。
6. 应用程序使用处理器，并且当它完成时通知服务器。
7. 服务器重新分配处理器。

图 5—12 Mach 中的处理器分配

这样系统中 CPU 使用变得十分灵活。特别是由大量处理器组成的大型并行系统。比如，可以为一个任务或一组任务专门分配几个处理器，来保证这些任务所需要的资源，而不用考虑系统的整个负载。极端情况下，应用程序可能为它的每一个线程分配一个专门的处理器，这就是群调度(gang scheduling)。

群调度(gang scheduling)对于需要 barrier 同步的应用程序十分有用。这类应用程序创建几个线程，这些线程独立运行一段时间，接着到达一个叫做 barrier 的同步点，线程必须在 barrier 处等待其他线程的到达，所有的线程在 barrier 处同步。应用程序可能运行一些单独的代码，接着创建另一批线程，并重复这种模式。

保证这类应用程序能够优化执行，在 barrier 处的延迟必须最小。这就要求所有的线程要差不多同时到达 barrier 点。群调度允许应用程序同时运行线程，给每一个线程分配一个处理器，这就能使 barrier 的同步延迟最小。

群调度对于线程间交互频繁的细粒度应用程序也十分有用。这类应用程序中，如果一个线程被抢占，那么它可能阻塞所有其他和它交互操作的线程。把处理器专门让一个线程使用的缺点就是如果这个线程阻塞的话，分配给它的处理器就不能被其他线程使用了。

系统中的所有处理器并不是完全相同的——一些可能很快，一些可能带有浮点运算单元等等。处理器集合能让不同的处理器执行不同的作业。比如，有浮点运算单元的处理器可能只分配给那些需要执行浮点算术的进程。

此外，可以暂时分配指定的处理器给线程。这个特性主要用来支持 Mach 的 UNIX 兼容代码的非并行处理器安全(not multiprocessor safe)的那一部分。这部分代码运行在指定的单个主处理器上(master processor)，每个处理器都有一个局部的运行队列。每个处理器集合都有一个全局的运行队列，集合中的每一个处理器共享这个全局队列。处理器首先检查它们的局部运行队列，因此对于绑定(bound)线程绝对优先(而超过较高优先级的非绑定的线程)。这种策略对非并行的 UNIX 代码提供最大的吞吐率，这样可避免瓶颈的发生。

5.8 Digital UNIX 的实时调度器

Digital UNIX 调度器支持分时和实时两种应用[DEC94]。它和 POSIX 1003.1b[IEEE 93]一起编译，POSIX 1003.1b 定义了实时编程扩展接口，尽管 Digital UNIX 派生于 Mach，但是它的调度器完全是重新设计的，它支持以下三种调度类：

- SCHED_OTHER 或是分时
- SCHED_FIFO 或先入先出
- SCHED_RR 或时间片轮询

用户可以调用 sched_setscheduler 来对进程设置调度类和优先级。缺省的类是分时类，在 nice 值和 CPU 使用情况的基础上，内核动态地改变进程的优先级，FIFO 类和时间片轮询类使用固定的优先级。使用 SCHED_FIFO 策略的进程没有时间片，它们将一直运行到它们自愿交出处理器或者是被高优先级的进程抢占。而分时类和时间片轮询类增加了一个时间片，它影响相同优先级的进程的调度。分时类或时间片轮询类进程用完它的时间片后，内核把它放到优先级队列的尾部，当然如果没有更高的或相同优先级的进程可以运行，它将继续运行。

调度器总是选择最高优先级的可运行的进程来运行，每一个进程有一个优先级范围是从

0～63，小的数表示低的优先级，调度器为每一个优先级维护一个有序队列，并选择非空的最高优先级队列的前面的那个进程。当阻塞的进程变得可执行了，或是当前运行进程交出它的处理器，内核通常把它放到对应优先级队列的尾部。一个例外情况，就是进程在用完它的时间片前被抢占，此时它将放在队列的首部。这样这个进程就可以在同样优先级的其他进程运行前用完它的时间片。

三个类的优先级有重叠部分这就提供了更大的灵活性。以下规则用来控制和管理进程优先级的分配：

- 分时类进程具有 0～29 的优先级。超级用户的特权要求优先级大于 19。
- 用户通过系统调用 nice 来改变进程的 nice 值，控制分时进程的优先级。nice 值的范围是从－20～＋20，其值越小表示优先级越高(向后兼容性)，只有超级用户能够设置负的 nice 值，对应的进程优先级在 20～29 之间。
- CPU 使用情况因子根据它所得到的 CPU 时间降低分时进程的优先级。
- 系统进程具有固定的 20～3l 范围内的优先级。
- 固定优先级的进程的优先级可以设定为 0～63 之间的任意值，但是超级用户特权要求优先级至少在 19 以上，优先级在 32～63 之间是实时优先级，这些实时进程不能被系统进程所抢占。

系统调用 sched_setparam 改变 FIFO 类和时间片轮询类中的进程。调用 sched_yield 把调用进程放在对应优先级队列的尾部，同时把 CPU 给了相同优先级的另一个可以运行的进程，如果系统中没有这样的可执行的进程，调用者继续运行。

5．8．1　多处理器支持

Digital UNIX 通过协调它的调度器去优化上下文转换和 cache 的使用[Denh 94]，从而高效使用多处理器。理想情况下，调度器总是能够让最高优先级的可运行的线程在所有可得到的处理器上面运行。这种策略需要调度器维持所有处理器共享的全局的运行队列，这可能产生一个瓶颈，因为所有的处理器都竞争去锁这些队列。进一步的讲，线程运行时，它占据处理器上的数据 cache 和指令 cache。如果这个线程被抢占，而且在很短的时间后又被重新调度，那么能让它在同一个处理器上运行是十分有意义的，因为它可以从以前的缓冲(caching)中受益，

为了重新协调这些冲突的目标，Digital UNIX 对分时类线程使用一个叫做软相近策略，这类线程保持在每个处理器的局部运行队列中，因此通常是在同一个处理器上重新调度。这也减少了运行队列的竞争，调度器监视每个处理器上的运行队列，通过把负载过重的处理器的运行队列中的线程转移到负载很轻的处理器的运行队列中，防止出现负载不平衡。

固定优先级线程从一个全局的运行队列中调度，因为它们应该尽快地运行。任何可能的时候，内核在上次运行的处理器上调度它们。最后，Digital UNIX 提供了系统调用 bind_to_cpu，要求线程必须在特定的处理器上运行。对于那些不是多处理器安全的代码，这个调用是十分有用的。

Digital UNIX 提供了一个 POSIX 兼容的实时调度接口。但是，它缺少 Mach 和 SVR4 的许多特性。它不能提供一个为处理器集合分配或不干涉调度的接口。它的内核是非抢占式的，而且不能控制优先级反转。

5.9　其他的一些调度实现

系统所要达到的调度目标取决于上面运行的应用程序的需要。一些系统执行实时的临界时间应用程序，一些系统主要执行分时应用程序，另一些系统是两者都有。一些系统中有

非常多的运行进程，这种情况下，现在的调度算法都不能很好地扩展。这就促进了多种不同的调度器的设计，其中的一些已经在不同的 UNIX 中实现。本节看一看一些有趣的方法。

5.9.1 fair-share 调度

fair-share 调度器给每个 share 进程组分配固定的 CPU 资源。每个 share 组可能只由单个进程组成，也可能是单个用户的所有进程，或是一个登录会话中的所有进程等等。超级用户能够选择如何在 share 组之间分配 CPU 时间。内核监测 CPU 使用情况来强化所选择的分配公式。如果任何一组没有用完分配给它的那份(share)CPU 时间，那么剩余时间就会按照其他组的“原始份”(original share)在其他组之间分配。

这种方法保证每份(share)进程组的可预测的处理时间。这和整个系统的负荷是无关的。对于计算时间是计账资源(billable resource)的环境，这种方法十分有意义，因为资源可以以固定的花费分配给用户。同时在分时系统中，它也能够保证分配给关键任务的资源。[Henr 84]中描述了 fair-share 调度器的实现。

5.9.2 最终期限驱动调度

许多实时应用程序必须在某些最终期限(deadline)内响应事件。比如，一个多媒体服务器，必须每 23 毫秒向客户传送一个图像帧，如果它从一个磁盘上读取数据，内核可以设置最终期限(deadline)，到时磁盘读操作必须完成。如果错过了最终期限，一个帧将被延迟。最终期限可以应用到 I／O 请求或是计算中，后一种情况，线程可以在最终期限之前请求一段已知的 CPU 时间。

最终期限驱动调度算法对于这类应用很有意义。这个算法的基本原则就是动态地改变优先级。最终期限接近时，增加应用的优先级，这种调度器的一个例子是在[Bond 88]中介绍的 FORTUNIX(Ferranti-Originated Real-time Extension to Unix)的调度器。它的算法定义四个不同的优先级层次：

- 硬(Hard)实时必须满足 Deadline 要求。
- 软(Soft)实时最终期限应该能以一个可计算的概率得到满足，偶然的不能满足也是可以忍受的。
- 分时，没有特定的 Deadline，但是希望有一个合理的响应时间。
- 批处理作业，它们的 Deadline 是以小时计的而不是毫秒。

系统是按进程的优先级来调度进程的，因此，比如一个软实时进程只能在系统中没有硬实时进程可运行时才能被调度执行。在 1，2，4 类中，进程是按照它们的最终期限调度的，最终期限最早的进程最先被调度。除非是高优先级类中的进程或同一个类中最终期限更早的进程变的可运行，否则进程将一直运行到完成或阻塞。分时类进程以传统的 UNIX 中的方式调度，它的优先级取决于它的 nice 值和最近 CPU 使用情况。

最终期限驱动调度适合于运行那些有已知响应时间需求的进程的系统，相同优先级策略应用于调度磁盘 I／O 请求等等。

5．9．3 三级(Three-Level)调度器

UNIX 调度器不能在允许任意负载混合的同时，保证实时应用的要求。主要的原因就是 UNIX 调度器缺少允许控制(admission Control)机制。对于系统中开始运行的实时的或通用的任务的数目没有限制，系统允许所有的进程以一个无控制的方式来竞争资源。依靠用户在得到足够的信息后和系统协作来保证系统不过载。

[Rama 95]描述了一个三级的调度器模型，这个调度器用在多协议文件和媒体服务器的实时操作系统中。调度器提供了 3 类服务——同步服务，实时服务和通用服务。同步类支持

固定时间间隔的活动，如图像帧传送，它必须在固定的时间间隔处理，只能允许最小的巅簸(jitter)和抖动(variation)。实时类支持不定期的，要求分配延迟有界的任务。最后，通用类支持优先级低的后台活动，调度器保证这种低优先级的进程向前执行的同时，不会影响提供给同步任务流。

这个调度器中增加了允许控制策略。接受一个新的图像流之前，它要拥有所有的流要求的资源，包括一部分 CPU 时间，磁盘带宽，网络控制器带宽等。如果服务不能具备这些资源，它就不能接受请求。每个实时服务每次活动中只能处理有界数目的工作单元(workunit)，比如，网络驱程序在它让出 CPU 之前只能处理进入消息的一部分。调度器也保留一部分固定资源为通用目的(general-purpose)活动，这些活动不受允许控制的影响。这样就避免了在负荷很重的系统中，通用目的请求的饿死。

为了让这种策略可行，系统不但调度 CPU 时间，而且调度磁盘和网络活动。根据初始化 I／O 请求的任务，系统给每个 I／O 请求分配一个优先级，处理低优先级 I／O 请求前必须处理高优先级的 I／O 请求。端到端的所有资源带宽的预留保证了系统能达到许可视频流的要求，也保证最大的系统负荷下，低优先级的任务也能向前运行。

三级调度器中，通用目的任务是完全可抢占的。实时任务可以在定义好的抢占点被同步任务抢占，这些抢占点出在每一个工作单元(work unit)完成之时。同步任务使用一个单调频率(rate-monotonic)调度算法[Liu 73]。这类任务有一个固定的调度优先级，它取决于任务发生的间隔。时间间隔越小，任务的优先级越高。高优先级的同步任务可在抢占点抢占低优先级的同步任务。[Sha 86]表明单调频率算法非常适合调度那些固定优先级并且间歇性发生的任务。

传统 UNIX 服务器存在的另一个问题就是无法处理系统大量负载造成的饱和。UNIX 系统在中断级别上处理大量的网络请求，如果进入的消息太多，系统就会花大量的时间处理中断，而留下非常少的 CPU 时间来处理请求。一旦进来的负载超过某个临界线，服务器的吞吐率会很快掉下来，这就是所谓的 receive livelock。三级调度器把所有的网络处理化为实时任务来解决这个问题。它限定在一次调用中处理的流量，如果进来的流量超过了这个临界值，服务器就会抛弃超出的请求。此时，系统仍然能对接收的请求进行正常的处理。这个高峰过后，系统的吞吐率差不多保持在固定的水平，而不是下降。

5．10 小　　结

我们已经仔细研究了几个不同的调度体系结构，也指出了系统是如何对不同的应用程序进行响应的。因为计算机使用在不同的环境中，每个都有它自己的需求集合，因此没有一个调度器能适合所有的系统。Solaris 2.x 的调度器对许多应用是足够的，它提供了动态增加调度类的框架，来满足特定环境的需求。尽管 Solaris 2．x 缺少一些特性，如实时 I／O 流和用户控制的磁盘调度，但它仍然是对传统的 UNIX 调度器的一个改进。我们已看到的一些其他的调度器解决方案，目标主要在于一些特定的应用领域如并行处理和多媒体。

5．11 练　　习

1．为什么调出函数不由时钟中断处理函数处理?

2．在什么情况下时间轮比 4．3BSD 管理调出函数的算法更有效?

3．使用 delta 间隔时间与绝对时间处理调出函数各有哪些优缺点?

4．为什么 UNIX 系统更适合于 I／O 密集型进程，而不是 CPU 密集型进程?

5．SVR4 调度器面向对象接口的优点有哪些?其缺点有哪些?

6．为什么在表 5—1 中每一行的 slpret 和 lwait 的值比同行的 tqexp 值高?

7．为什么要将实时进程的优先级设置得比内核进程还要高?这样有什么缺点?

8．事件驱动调度方法对 I／O 密集和交互式应用的影响怎样?

9．考虑第 5．5．6 小节介绍的[Nieh 93]试验，如果赋予 X 服务器，视频应用和交互式任务实时优先级，同时设置批处理作业分时优先级，这样一来的结果怎样?

10．假设一个进程释放了若干个进程正在等待的资源。什么时候唤醒所有进程好，什么时候只唤醒一个进程好?如果只唤醒一个进程，该如何选择这个进程?

11．调度技术假设每个线程都运行在一个独立的处理器上。如果处理器比可运行的线程少，而应用需要完成一次 barrier 同步操作时将会怎样?在这种情况下，线程会在 barrier 上忙等吗?

12．持实时应用，Solaris 2．x 中采用哪些不同的方法?从哪些方面看这是不合适的?

13．为什么 deadline 驱动调度方法不适合于传统的操作系统?

14．实时进程的特性有哪些?试列举周期型和非周期型实时应用的例子。

15．可以通过简单地通过使用更快的处理机来降低相应时间和分派延时。实时系统与快速高性能系统间的区别是什么?是否一个很慢的系统却更适合于实时应用?

16．软实时和硬实时的需求有哪些不同?

17．什么实时系统中的允许进入控制非常重要?

5．12 参考文献

[AT&T 90] Americal Telephone and Telegraph，UNIX System V Release 4 Inter—nals Students Guide，1990．

[Blac 90] Black，D．L，“SCheduling Support for Concurrency and Parallelism in the Mach Operating System，”IEEE Computer，May 1990，pp，35—43．

[Bond 88] Bond，P．G，，“Priority and Deadline Scheduling On Real—Time U-NIX，”Proceedings of the Autumn 1988 European UNIX Users，Group conference，()ct．1988，pp．201—207．

[DEC 86] Digital Equipment corporation，VAX Architecture Handbook，Digital Press，1986．

[DEC 94] Digital Equipment Corporation，DEC OSF／1 Guide to Realtime Pro—gramming，Part No．AA—PS33C—TE，Aug，1994．

[Denh 94] Denham，J．M.，Long，P.，and Woodward，J．A.，“DEC OSF／l Ver—sion 3．0 Symmetric Multiprocessing ImplementatiOn，”Digital Techni-cal Journal，Vol.6，NO.3，summer 1994，pp．29—54．

[Henr 84] Henry，G．J.，“The Fair Share Scheduler，”AT&T Bell Laboratories Technical Journal，V01．63，No.8，Oct 1984，pp.1845—1857．

[IEEE 93] Institute for Electrical and Electronic Engineers，POSIX P1003．4b，Real-Time Extensions for Portable Operating Systems，1993.

[Khan 92] Khanna，S.，Sebree，M.，and Zolnowsky，J.，“Realtime Scheduling in SunOS 5．0，”Proceedings Of the Winter 1992 USENIX Technical Conference，Jan．1992．

[Lamp 80] Lampson，B．W．and Redell，D．D.，“Experiences with Processes and Monitors in Mesa，”Communications of the ACM，vol．23，no．2，Feb 1980，pp．105—117．

[Lin 73] Lin，C．L．and Layland，J．W.，“Scheduling Algorithms for Multipro-gramming in a Hard real-Time Environment，”journal of the ACM，V01．20，no．1，Jan．1973，PP．46—6l．

[Leff 89] Leffler, S. J., McKusick, M. K., Karels, M. J., and quarterman, J. S., The Design and Implementation of the 4. 3 BSD UNIX Operating Systerm, Addison-Wesley, Reading, MA, 1989.

[Nieh 93] Nieh, J., "SVR4 UNlX Scheduler Unacceptable for Multimedia applications," Proceedings of the Fourth International Workshop on Network and Operating Support for Digial Audio and Video, 1993.

[Rama 95] Ramakrishnan, K. K., Vaitzblit, L, Gray, C, G,, Vahalia, U., Ting, D., Tzelnic, P., Glaser, S., and Duso, W. W., "Operating System Support for a Video-on-Demand File Server," Multimedia Systems, Vol. 3, NO. 2, May 1995, PP. 53—65.

[Sha 86] Sha, L., and Lehoczky, J. P., "Performance of Real-Time bus Scheduling Algorithms," ACM Performance Evaluation Review, Special Issue, Vol. 14, No. 1, May 1986.

[SHA 90] Sha, L., Rajkumar, R., and Lehoczky, J. P., "Priority Inheritance Protocols : An Approach to ReaL-Time Synchronization," IEEE Tansactions on computers, Vol, 39, No. 9, Sep. 1990, pp. 1175—1185.

[Stra 86] Straathof, J. H,, Thareja, A, K. , and Agrawala, A. K., "UNIX Scheduling for Large Systems," Proceedings of the Winter 1986 USENIX Technical Conference, Jan, 1986.

[Varg 87] varghese, G., and Lauch, T., "Hashed and Hierarchical Timing Wheels : Data Structures for the Efficient Implementation of a Timer Facility" Eleventh ACM Symposium On Operating system Principles, Nov. 1987, pp. 25—38.