Linux 进程管理体系的研究与分析

赵慧斌　李小群　叶以民
（中国科学院软件所　北京100080）

Research of Process Management of Linux

ZHAO Hui-Bin　LI Xiao-Qun　YE Yi-Min
(Institute of Software, Chinese Academy of Sciences, Beijing 100080, China)

Abstract　Using an opened source system, Linux, as the supported OS is more and more appealing to many developers. So it is attractive to many developers to understand the designing philosophy of Linux and more importantly, improve its performance to satisfy the specific requirements. This paper analyzes process management of Linux ,and the schedule algorithm of Linux is also presented.

Keywords　Linux, Operating system, Process management

1　引言

Linux 作为一个多任务的操作系统，支持分时处理，由于其良好的性能，完善的设计，在服务器、台式机、嵌入实时等领域得到了广泛的应用。在操作系统的几个关键设计中，如内存管理、中断管理、进程、文件系统、进程通信等方面都有独到的特点，而作为操作系统的关键部分之一——进程管理方面，Linux 作了较为周密的考虑，系统的高效、稳定很大一部分来自于进程管理的严整规划。

Linux 是一个多任务操作系统，它要保证 CPU 时刻保持在使用状态，如果某个正在运行的进程等待外部设备完成工作(例如等待打印机完成打印任务)，这时，操作系统就可以选择其他进程运行，从而保持 CPU 的最大利用率，这就是多任务的基本思想。进程之间的切换由调度程序完成，进程调度要考虑 CPU、内存、外设多方面资源的合理利用，因而成为系统实现引起关注的部分，本文主要围绕调度及与调度相关的时钟、进程切换等方面讨论。

2　Linux 的进程模型

2.1　进程

进程是一个动态的概念，进程运行过程实际上是进程的一个生存周期。在这一周期之中，通常可分为典型的3种状态：运行状态，实际占用 CPU；就绪状态，进程可运行，但暂时挂起；阻塞(或中断)状态，资源不可用，等待外部事件的发生。

前两种在系统看来很相似，都是进程处于可运行状态，只是后者暂时未获得 CPU。在 Linux 2.0之后的版本，实际上这两种状态已经合而为一。第3种状态则不同，处于该状态的进程即使在 CPU 空闲时也不能运行，只有外部资源条件满足，才可唤醒执行。

这3种状态的转换关系如图1所示，阻塞状态可能由系统调用引发，也可能在等待外部事件时自动进入(比如读取设备文件时，没有可用的输入)；而运行状态和就绪状态的转换则由调度程序负责，这两种状态一般也只在调度算法和调度程序的观点下是有意义的，系统或用户感觉不到它们的差别。实际的系统中，进程状态可能复杂一些，但都可以归为以上3种状态。

图1　进程状态模型

2.2　线程

与进程概念紧密相关的是线程的概念。线程可以理解为同一进程中相互独立执行的上下文，线程是“多任务”的进程。另一种说法是线程是轻量级的进程——除了同一线程组的线程共享全局变量，且有相同的堆栈这一点外，线程与进程没有什么区别。

线程的概念较为适用于一些紧密耦合的一组处理流程。传统进程的概念中，一个进程有一条执行线索，进程之间空间、时间独立，互不干扰。而线程则逻辑上是一个事务的不同线索，线索之间有着种种联系，可能是共享一段缓存或协调执行一组任务。典型的如一个浏览器，许多 Web 页面有多幅很小的图像，对所有的小图像，浏览器都必须与站点建立连接以获取该图像，如图2(a)所示，线程的优势在于连接建立后可以共享连接，同时请求传输多个图像，而在如图2(b)所示的

图2　(a)进程模型　　(b)线程模型

进程环境下就不合适，连接不是公用的，所以要浪费大量的时间为每个图像建立和释放连接。多数情况下，显然前者会有更高的效率，因为对于小图像，效率主导因素将是建立连接的时间，而不在于传输的速度。

线程的实现有两种方式：核心线程和用户线程。前者是指系统核心为线程定义专门的数据结构，就像进程一样，线程在核心中为所有线程建立了一张线程表，系统在线程表中、在核心态中选择或阻塞某个线程；而用户线程则完全不需核心参与，线程只是用户进程中由一些程序计数器、寄存器值等建立的线程环境，核心仅能了解到进程，而不知道线程的存在。两种方式各有其优缺点：核心线程由于线程切换必须在核心下完成，因而效率较低；而用户线程的情况下，因为内核不知道线程的存在，如果遇到线程由于 I/O 事件的原因阻塞时，所有同一用户空间的线程也被阻塞。

截至到2.2版本的 Linux 内核提供了_clone 函数，这是线程的内核支持函数。Linux 同时也有用户线程的实现库，关于线程的争论和问题很多，但 Linux 操作系统更多地将注意力放在机制的支持上，而不是策略的实现上，这不失为一种良好的态度。

3 Linux 的时钟系统

3.1 时钟中断服务(立即中断服务)

时钟管理是操作系统特别是分时操作系统的脉搏，操作系统环境建立之后，任务的执行和中止很多情况下是时钟直接或间接唤起的。时钟是许多操作系统基本活动的基准，系统用它来维持系统时间，监督系统运作，还是进程调度的重要依据。

对于通用的 PC 机，都包含有8264/8253可编程定时/计数器，Linux 通过对可编程定时/计数器的初始化，定时器将对输入脉冲进行计数分频，以每一秒100次的频率产生时钟脉冲信号。这个信号接到8259中断控制器的0号管脚，触发时钟中断。Linux 的时钟管理基于100Hz 的时钟频率，每次时钟中断将更新 jiffies。时钟系统将维护与时钟有关的核心定时器，而所有的核心定时器依赖于 jiffies 计数，于是 jiffies 就成为整个时钟系统的标度。

Linux 中立即中断服务(Immediate Interrupt Service)，也称为中断的上半部分(top half)，其优先级非常高，执行时需要关中断，但在该部分仅做最简单的处理，而把复杂的耗费时间长的工作放到第二部分处理，即时钟底半(bottom half)处理。从时钟中断的产生到立即中断服务程序执行的时间延迟主要由以下几个部分组成：

·当中断到来时，CPU 首先执行完当前指令，随后从中断控制器取得中断向量，并根据具体的中断向量，从中断向量表 IDT 中找到相应的表项。该表项应该是一个中断门，根据中断门的设置，CPU 就可以找到时钟中断服务程序的入口地址，取出核心堆栈指针，并将堆栈切换到核心堆栈；

·关中断的时间；

·保存中断发生前的系统现场，标识时钟底半，获得中断服务程序的入口地址等的时间；

·执行立即中断服务程序的时间。

以上所有时间之和被称为中断延迟，即中断响应时间。可以看到中断延迟的第一部分主要与硬件有关，所用的时间较少。同样，第二步和第三步的时间也较少。最不可控制和主要时间是在最后一步，可以尽量减少这一步所做的工作，让大部分工作由调度中断服务程序去做以减少延迟时间。

3.2 底半机制

中断服务程序必须足够快，且处理尽量少的任务。而操作系统高层处理相对复杂，所需的处理时间较长，这样就不利于中断的快速响应，乃至丢失重要的中断。为了协调中断处理功能及性能之间的矛盾，Linux 引入了独特底半机制。在这种设计机制下，快速的、简洁的和必要的中断响应放在上半部分即时处理，而将耗时长的、与操作系统高层环境相关的部分放在底半，在合适的时机延后处理。在时钟中断处理方面，底半机制的特点表现得更加突出。

底半机制是 Linux 比较独特的一种内核机制，它是中断底层到操作系统高层转换的通道。尽管这种设计较为新颖，但与此对应的设计可以在一些微内核操作系统中找到。在微内核操作系统中如 Minix，中断信号被理解为一种 IPC 消息，硬件中断可视作是具有特殊 id 的一组线程，它们送出的消息只有发送方 id 即可。消息的生成由内核完成，中断服务完成后返回用户态也由内核负责，但与设备相关的中断处理微内核并不参与。接受消息的线程通过鉴别消息发送者的 id，判断消息是否是硬件中断及中断类型，根据需要作相应的处理。

在这种设计理念下，中断信号的消息自然而然地就会与其它的 IPC 消息具有同样的操作集—send 和 receive。消息机制中，如果 send 阻塞，则将给此进程发送消息的所有进程都链在一个链表上，同样，如果是来自硬件的消息，也就是中断消息，如果不能立刻处理，将会排成一个链表，在恰当的时机，执行相应的期待中断消息的任务并接受这一消息。如图3所示：

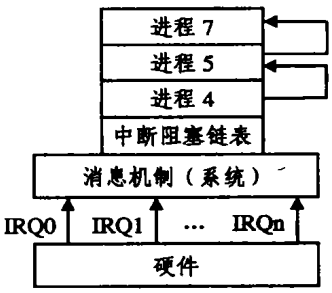

图3 微内核操作系统下的中断处理

底半机制沿用了这种思路，尽管实现方式不尽相同，但原理一样。典型的数据结构实现是一个进程队列 task_queue，如图4所示。

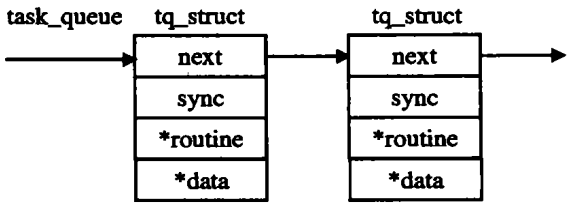

图4 进程队列

Linux 在中断的上半部分可以将处理流程复杂的底半部分放在 task_queue 的队列中，在某些触发条件下，如时钟中断发生，触发底半处理，完成整个处理过程。从以下的 Linux 时钟中断处理的整个过程，我们可以进一步理解底半的概念。

时钟上半部处理较为简单，定时器中断函数timer_interrupt 是注册的 irq0——即时钟中断的响应函数(arch/i386/kernel/time.c)。另外，time_bh 被注册为定时器的底半部分。当触发 IRQ0时，timer_interrupt 从 CPU 时间戳计数器中读取一些属性值。紧接着调用 do_timer_interrupt，它会调用

do_timer，从底半机制的观点来看，这是我们关心的时钟中断的上半部分。do_timer 函数做的工作都是简单、十分必要的：

·更新 jiffies：这是系统启动以来系统时钟的滴答数。

·更新 lost_tick：这是那些自前一次底半处理运行后，未被处理的底半的次数，当底半部分得到运行后，lost_tick 将"补"回来。

·更新 lost_tick_system：除了从上一次时钟的底半处理运行以来时钟的滴答数外，还需要知道多少次滴答发生在系统模式下。这是为统计进程占用的 CPU 资源而设的。

·标记底半处理：如果定时器队列有任务等待，定时器队列的底半部分被标记为准备好运行。

timer_bh 函数是中断的底半，它完成的工作较为复杂，我们只作简要介绍，但有一点应该指出的是，timer_bh 的执行不像 do_timer，执行时间是"不定"的，这是因为 timer_bh 的执行涉及到了定时器队列，而且与信号、调度相关。这正是我们前面提到的底半机制的观点：底半在恰当的时机，处理了一些与底层中断相关的任务。总的来说，时钟底半做了以下的工作：

·更新 xtime，记录当前的 wall_clock；

·更新任务时钟，统计当前进程已运行的时间；

·运行早期的定时器服务，这是最多只有32种的定时器服务；

·运行"新"的定时器服务，这是采用了链表结构的定时器服务，服务数量不再受限制；

·时钟的上半部分 do_timer 是时钟中断发生就会被执行，而 timer_bh 不但执行时间不确定，而且执行时机也不定，在以下3种情况下，且 bh_active 和 bh_mask 都允许的条件下，timer_bh 才会被执行：1）schedule()被调用；2）系统调用返回；3）"慢速型"中断处理返回前。

4 Linux 下的进程实现

4.1 进程调度相关的数据结构

进程的数据结构是系统中最为重要的几个数据结构之一，记录了进程的所有信息，包括进程状态、进程标识、时间、信号、内存资源、上下文环境、文件系统等多方面的内容，我们主要讨论进程状态和与调度相关的内容。

```
struct task_struct {
/* these are hardcoded - don't touch */
 volatile long state; /* -1 unrunnable, 0 runnable, >0 stopped */
 unsigned long flags; /* per process flags, defined below */
 ……
 long need_resched;
/* various fields */
 long counter;
 long priority;
 cycles_t avg_slice;
 ……
 struct task_struct *next_task, *prev_task;
 struct task_struct *next_run, *prev_run;
 ……
 pid_t pid;
 pid_t pgrp;
 pid_t tty_old_pgrp;
 pid_t session;
 ……
 /*
  * pointers to (original) parent process, youngest child, younger
    sibling,
  * older sibling, respectively. (p->father can be replaced with
  * p->p_pptr->pid)
  */
 struct task_struct *p_opptr, *p_pptr, *p_cptr, *p_ysptr,
     *p_osptr;
 /* PID hash table linkage. */
 struct task_struct *pidhash_next;
 struct task_struct **pidhash_pprev;
 ……
 unsigned long policy, rt_priority;
 ……
 /* tss for this task */
 struct thread_struct tss;
 ……
};
```

4.2 进程状态

volatile long state：这是与进程状态相关的变量，表示进程的当前状态。

·运行(TASK_RUNNING)：正在运行和准备运行的任务，系统中有一个可运行队列(run_queue)，用来容纳所有处于可运行状态的进程。调度时，从这个队列中选择一个进程执行。

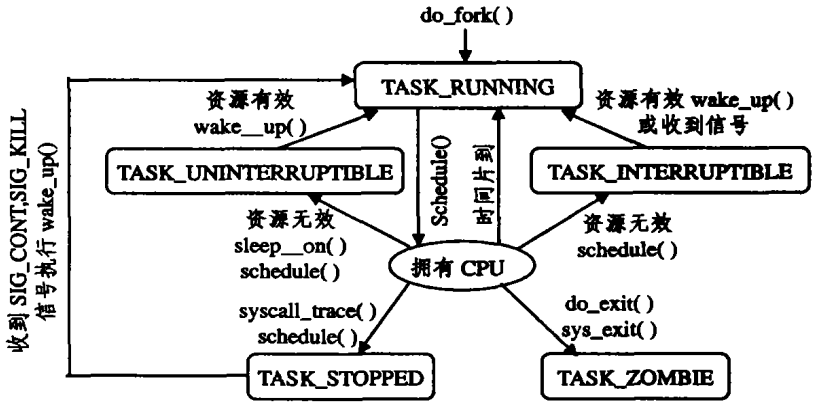

图5 进程状态转换

·可中断睡眠(TASK_INTERRUPTIBLE)：处于等待队列中的进程，待资源有效时唤醒，也可通过信号或定时中断唤醒后进入。

·不可中断睡眠(TASK_UNINTERRUPTIBLE)：处于等待队列中的任务，资源有效时唤醒，但不能由信号或定时中断唤醒。

·暂停(TASK_STOPPED)：进程暂停，通过其他进程的信号才能唤醒。

·僵死(TASK_ZOMBIE)：进程已结束，但尚未消亡的状态，进程已经结束运行且释放大部分资源，但 task_struct 仍未得到释放。

进程的状态转换图如图5所示。

进程在 do_fork 结束之后就进入了 TASK_RUNNING 状态，进入运行任务队列，在特定的时机，由 schedule 函数选

中，拥有 CPU 并执行，在时间片到(如典型的时间片轮转调度策略)，它再次进入运行任务队列，等待调度。在其拥有 CPU 期间，也可能由于等待资源而进入TASK_INTERRUPTIBLE 或 TASK_UNINTERRUPTIBLE 的状态，这两种状态没有本质的区别，只是后者仅能由 wake_up()唤醒，而前者可以在收到信号后唤醒。而 TASK_STOPPED 则是在设置了系统跟踪标志后，进入 syscall_trace 时的状态，一般在调试跟踪系统调用时用到。它在收到 SIG_KILL 或 SIG_CONT 才会被唤醒，重新进入运行任务队列。

4.3 进程调度

进程运行过程中用到多种资源，如 CPU、内存、文件等等，因而调度的实质就是资源的分配。在多任务操作系统中，调度算法的选择取决于多方面的因素。一般来说，一个好的调度算法应当考虑以下几个方面的因素：

·公平：每个进程都应得到合理的 CPU 时间；

·高效：CPU 的利用率更高，即总是有进程在 CPU 上运行；

·响应时间：对用户的响应时间尽可能短；

·周转时间：批处理等待输出的时间尽可能短；

·吞吐量：单位时间内处理的进程数量尽可能多。

这5个目标不可能同时达到，所以不同的操作系统会根据需求取舍，选择不同的调度算法，这就是调度策略的取向了。

4.4 Linux 支持的调度策略

Linux 把3种调度策略合成了一种，task_struct 结构中的 policy 成员，就是与调度策略相关的。policy 的选择不外以下3种：

1) SCHED_OTHER：传统的 Unix 调度，时间片轮转——这不是一个实时进程，在分时系统中，出于人机交互的考虑，系统每个进程依次地按轮转的方式执行，这就是时间片轮转调度。系统将所有的可运行任务按先来先服务的原则，排成一个队列，每次调度时把 CPU 分配给队列的首进程，一个时间片之后，系统会运行调度程序，停止该进程的执行，并将其放入队列末尾，系统再次选择队列首进程，令其执行一个时间片。这样，进程在一个给定的时间内均可以获得一个时间片的执行时间。整个过程如图6所示。

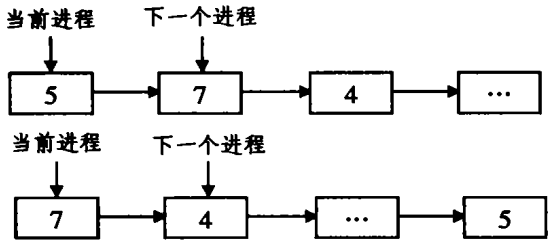

图6 时间片轮转调度

2) SCHED_FIFO：实时进程，遵守 POSIX.1b 标准的 FIFO(先进先出)调度。SCHED_FIFO 进程会始终拥有 CPU，只有更高优先级的进程出现或 I/O 阻塞，进程明确放弃 CPU 后才会换出，这种调度算法可用于某些实时性要求不高的实时系统中。

3) SCHED_RR：实时进程，遵守 POSIX.1b 的 RR (round-robin)调度。除了时间片的限制，它和 SCHED_FIFO 完全相同，SCHED_RR 进程在时间片用完后，进程保持 rt_priority，且被移到进程列表的最后。

4) Policy 还有一个 SCHED_YIELD 的位集，这和调度策略无关，进程在明确交出 CPU 的控制权时，就设定该位。

4.5 Linux 进程调度算法

Linux 遵循快餐式(quick-and dirty)调度设计准则，它只完成了一些尽量简单的合理工作，从而进程本身可以获得尽可能多的 CPU 时间。

进程调度关键的函数是 kernel/sched.c 中的 schedule 和 goodness，在这里不必详细分析函数的执行流程，而只是简单介绍其实现的思想。调度过程是这样的：

1) 遍历任务队列，找到最值得(goodness)运行的任务；

2) 切换到这个进程 mmu 上下文；

3) 切换到这个进程的空间。

schedule 函数中，在 task queue，bottom half 等处理完之后，就要对任务队列进行处理，如 Round Robin 调度策略下的任务移动、任务删除等。这时，中断上锁，保证不会有新任务插入到任务队列时造成混乱。

进程的缺省调度算法是 SCHED_OTHER，即时间片轮转，在这种调度算法下，进程调度的依据就是它所使用的时间片。内核代码中它的实现并不那么明显。进程的时间片是由 task_stuct 成员 counter 决定的，而 counter 的变化是在 kernel/sched.c 中的 update_process_times 函数中，它会将 counter 的值减少从上次时钟底半运行以来经过的 tick(实际是时钟的上半部分中的 lost_tick 数)。当 counter 减少为小于0时，就需要重新调度了，这一点在 update_process_times 中很明确。正如我们在前面提到的，update_process_times 在时钟中断的底半执行，这样，时钟系统就和进程调度相互作用了。

goodness 函数是调度的依据，它的工作比较简单：如果调度策略是实时调度(SCHED_FIFO 和 SCHED_RR)，则简单地给 weight 赋一个与 rt_priority 相关的且很大的值(实际代码是1000+rt_priority)，如图7所示，这样就与非实时进程区分开了；如果是一般进程(SCHED_OTHER)，将进程的 counter 值(一般来说，不会大于20)赋给 weight 即可。结合前面调度策略的说明，我们可以印证：实时进程的执行总优先于非实时进程；在 SCHED_FIFO 策略下，则尽可能地运行最高优先级的进程，且尽可能地占用 CPU 的资源。

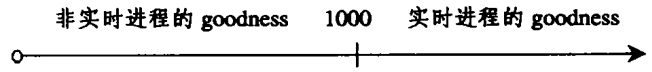

图7 goodness 值

4.6 Linux 进程切换

进程切换涉及当前任务上下文(堆栈，程序指针)的保存，及恢复将要切换的任务的上下文。Linux 对进程切换的管理完全由软件实现，最大限度地保持了内核的可移植性。

与进程切换相关的是 task_struct 中的 tss 字段，关于这一字段的数据结构比较复杂，也不必详细解释，它存储的是进程的上下文环境，这里用到了两个成员：ESP(进程的栈地址)和 EIP(进程的当前运行指针)。

Linux 中任务切换起主要作用的部分是 switch_to(prev，next，last)，这是一个宏定义，其主要工作是：(1)保存 prev 任务的 ESP，EIP；(2)恢复 next 任务的 EIP，ESP，ret 后进入 next 的空间。

EIP 的恢复借助了 ret 语句的功能。实际过程是这样的：先将进程 next 的 EIP 压入堆栈，然后程序跳转到_switch_

to,_switch_to 函数的最后是 ret 语句,而 ret 语句的功能就是弹出堆栈的最上面的值作为当前的 EIP,这样 next 进程的 EIP 就出栈成为当前执行的 EIP,完成了整个任务上下文的切换过程。以下是进程切换的代码。

```
#define switch_to(prev,next,last) do { \
1     asm volatile("pushl %%esi\n\t" \
2                  "pushl %%edi\n\t" \
3                  "pushl %%ebp\n\t" \
4                   "movl %%esp,%0\n\t" / * save ESP,
                    即 %%esp ->prev->tss.esp * /
5                  "movl %3,%%esp\n\t" / * restore ESP,
                   即 next->tss.esp -> %%esp * /\
6                  "movl $1f,%1\n\t" / * save EIP,这里是将第10
                   行的 IP 地址存入 prev->tss.eip * /
7                 pushl %4\n\t" / * restore EIP ,将 next->tss.eip
                  压入栈,这样当执行_switch_to 的 ret 后,程序 EIP
                  就到了 next->tss.eip * / \
8                  "jmp__switch_to\n" \
9                  "1:\t" \
10                 "popl %%ebp\n\t" \/ * 这里就是第6行存储的
                   EIP
11                 "popl %%edi\n\t" \
12                 "popl %%esi\n\t" \
13                 :" = m" (prev->tss.esp)," = m" (prev->
                   tss.eip), \
14                  "=b" (last) \
15                 :"m" (next->tss.esp),"m" (next->tss.eip), \
16                 "a" (prev), "d" (next), \
                   "b" (prev)); \
} while (0)
```

小结 进程实现是分时系统的核心部分之一,且与操作系统多方面的内容联系,到目前,我们已给出了进程调度的完整线索,从时钟系统到调度策略,这是大多数分时操作系统遵循的设计原则。Linux 是十分成功的操作系统,许多方面的设计都十分新颖,而且在实现上也相当富于技巧,其设计思路和方法都是值得借鉴的。

参 考 文 献

1 Tanenbaum A S, Woodhull A S. Operation Systems Design and Implementation, Second Edition, Prentice-Hall, 1997. 47～148

2 The Portable Application Standards Committee of the IEEE Computer Society. P1003. 13 draft standard for information technology--standardized application environment profile--posix realtime application support(aep):[Technical report]. IEEE, 1998

3 Stalling W. 操作系统内核与设计原理,第四版. 电子工业出版社, 2001

4 李善平,郑扣根. Linux 操作系统及实验教程. 机械工业出版社, 2001

5 毛德操,胡希明. Linux 内核源代码情景分析(上册). 浙江大学出版社,2001

(上接第83页)

从图8可以看出,采用快速磁盘队列 FastQueue 后,DATA 和 Deliver 命令的处理时间成倍下降。如上文分析,DATA 和 Deliver 命令主要与磁盘队列交互,DATA 和 Deliver 命令的高效来自快速磁盘队列 FastQueue 优化的存储布局和高效的消息读写策略。因此,快速磁盘队列 FastQueue 从整体上提高了消息传递服务器的性能。

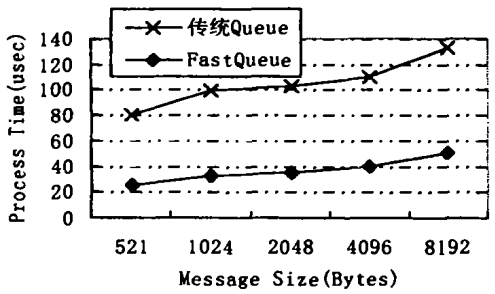

图9 传统磁盘队列和 FastQueue 写性能对比

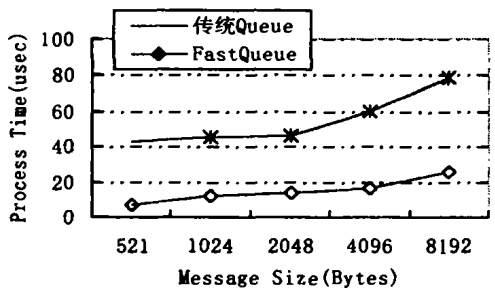

图10 传统磁盘队列和 FastQueue 读性能对比

进一步测试 Data 和 Deliver 命令中读、写性能,图9为写消息性能对比,图10为读消息性能对比。可以看出读、写消息的性能得到成倍的提高。

结束语 本文详尽分析了消息传递服务的存储开销,并提出一种高性能的磁盘队列存储管理机制—FastQueue。FastQueue 采用预先分配的连续磁盘块作为磁盘队列,将多份消息对象保存到一个文件中,消除或减少文件创建、删除等文件管理开销,并减少了磁盘碎片;采用延迟聚集写和预先连续读策略,通过一次磁盘搜索旋转定位,尽量读写更多的有效数据,不仅减少了读写次数,而且读写连续的磁盘块也充分利用了磁盘带宽。实验表明,FastQueue 能获得比传统磁盘队列高出约230%的性能。

参 考 文 献

1 Gregory R, Ganger M, Kaashoek F. Embedded Inodes and Explicit Grouping: Exploiting Disk BandWidth for Small Files. Annual USENIX Technical Conference (Anaheim, CA), Jan. 1997. 1～17

2 Kathy C M, Richardson J. Reducing the Disk I/O of Web Proxy Server Caches, USENIX Annual Technical Conference Monterey, California, USA, June. 1999

3 Evangelos P, Markatos Manolis G, Pnevmatikaos H K D. Secondary Storage Management for Web Proxies, 2nd USENIX Symposium on Internet Technologies & Systems Boulder, Colorado, USA, Oct. 1999

4 John M R, Ousterhout K. The Design and Implementation of a Log-Structured File System. ACM Transaction on Computer Systems, Feb. 1992. 10～22

5 Matthew McCormick Jonathan Ledlie. A Fast File System for Caching Web Objects, First Annual Symposium on Operating Systems Design, Implementation, and Evaluation (OS-DIE 1)