深圳大学

操作系统之编程观察

（Linux 平台）

(V 1.00)

罗秋明 著

批注 [l1]:

内容简介

本书弥合了操作系统理论课程和操作系统编码实现之间的断层，为正在学习操作系统的学生或相关技术人员提供实践指引。以 Linux 真实操作系统为观察对象，主要利用 proc 文件系统暴露出来的内核行为数据来讲述操作系统的核心概念。第一章简单介绍了 Linux （CentOS）和 Vituralbox 虚拟机的安装。第二章介绍了进程实体并通过编程实践来观察进程创建及其组织关系；对比了进程和线程的资源开销差异；通过编程观察了孤儿进程、僵尸进程和守护进程。第三章讨论了进程状态转换、CFS 进程调度和实时进程调度；观察了 CFS 和实时进程的负载均衡行为。第四章学习和观察了使用管道、消息队列、共享内存等通信手段；观察了 System V 信号量集和 POSIX 信号量对进程进行同步的活动。第五章通过/proc/PID/masp 观察进程虚存空间，以及程序操作对虚存空间的改变；模拟 CPU 地址部件通过查找多级页表来完成地址转换的过程；观察了进程的物理页帧分配、回收等行为。第六章对前面五章的知识进行一次整合，以 shell 上执行一个可执行文件的全过程而将虚存空间管理、物理内存管理、用户空间与内核空间、中断与系统调用、进程调度等概念有机的联系起来。第七章在 VFS 通过文件模型之下观察了文件的基本操作、目录操作、文件系统挂载/卸载等活动行为；通过编程观察页缓存和交换空间等变化等。第八章针对 EXT2 具体文件系统，通过在一个块设备建立起 EXT2 文件系统，并通过设备的读写操作而完成文件数据和目录内容的读写。通过编程实例，将进程控制、进程线程差异、进程调度、进程通信、内存管理、文件系统和设备管理等关键内容，都以各种编程代码和内核数据的数值形式将操作系统内核行为直观可见地展现给读者。

限于作者的水平和能力，书中必定有不少错漏，欢迎读者指正。联系邮件：lqm@szu.edu.cn。

序

深圳大学计算机与软件学院正在进行教学改革，在明仲教授和王志强教授两位领导的构想下，计算机与软件学院参照美国纽约宾汉姆敦大学教学要求将他们的课程在深圳大学原样重现。计算机系统系列课程和操作系统课程都进行了改革，其中操作系统课程缩减了课堂理论授课增加了实验操作环节。同时我们计算机系统课程组正在承担广东省教育厅应用型人才培养项目（计算机系统系列核心课课程），强调丰富的动手实践经验并提高系统能力、系统思维。在上述环境下，为了充实实验内容保持学生在课程学习中全程紧张，我们完成了《操作系统之编程观察》以配合理论教学。

在过去的操作系统教学过程中，作者深感理论教学与实践的脱节之困。虽然也有老师讲授 Linux 工程实践和系统编程从而拉近了两者的距离，但是都还未能与操作系统的核心概念紧密联系，仍似隔靴搔痒地可望而不可及。另外也考虑过将 Linux 内核源码分析或增强作为实践内容，但是由于学习曲线过于陡峭需要花费太多的时间，并不太适合作为课程内容——难以在一个学期课程中结合进来，甚至还可能让学生产生无功而返的挫败感。

根据我们在个人高性能计算机（PHPC）系统研制过程中的研究生培养经验和积累，将 Linux 系统编程的基础知识结合 Linux 的内核行为观察，利用/proc 文件系统中探测到的内核数据和其他各种工具收集的内核数据，直观生动地将进程与内核的交互、内核的行为展示给学生，获得非常好的学习效果。学生对进程行为、内存分配管理、进程间通信和文件系统等各方面的认知，都远比传统的操作系统课程教学效果好。

读者在学习和体验操作系统各种概念的同时，也获得了初步的系统编程实践锻炼，同时也为读者进一步阅读 Linux 内核源代码做好充足的准备。将 proc 文件系统和相关工具加入到操作系统的学习过程中，相当于有了电路系统课程中的“万用表、示波器和逻辑分析仪”等工具——有了观测工具后，操作系统的教学和实验才算基本成熟了。正因为这些观测工具，使得一些读者在完成全书学习后可能会觉得——哦，原来这才是操作系统。经过这样的实践锻炼后，后续学习中不仅加快了系统编程的学习进度，也加快了学生获得分析和修改 Linux 内核代码能力的培养过程。

《操作系统之编程观察》是我们这些年本科教学和研究生培养过程中积累的经验汇总，相信读者会喜欢。

致谢[1]

感谢深圳大学计算机与软件学院的操作系统课程组的老师，大家一起完成了操作系统实验课程的改革，特别是张滇和周明洋两位老师在相关实验内容的检查和教学工作中起到极大的帮助。

还需要感谢 2014 级的几位同学在相关的材料整理和实验代码的设计中作出的贡献。其中林润胜同学完成了**错误!未找到引用源。**匿名映射、文件映射、meminfo 和 zoneinfo 解读及相关代码，**错误!未找到引用源。**的 EXT2 文件系统中文件内容读取、目录读取及相关代码，协助完成了**错误!未找到引用源。**的勘误；张永昌同学完成了 5.1.1 小节、7.3.3 小节的内容和相应的代码。这两位同学一起提供了 5.3.3 小节的部分内容，完成了**错误!未找到引用源。**内存管理和**错误!未找到引用源。**EXT2 文件系统的勘误工作。罗文杰同学完成了**错误!未找到引用源。**小节的信号量集相关代码和材料。2017 级研究生汤钊扬同学作为第一位读者，协助验证代码和完成勘误工作。

在上述老师和同学的大力支持下，《操作系统之编程观察》终于完稿并与读者见面。再次表示衷心的感谢！

[1] 本书获得深圳市科创委基础研究 JCYJ20150930105133185 项目和 JCYJ20170302153920897 项目的资助

目录

第1章 进程调度

在操作系统理论课程的教学中通常讲述多种调度算法，例如 FIFO、RR 等算法，也会讨论实时调度算法。但是 Linux 作为一个通用操作系统，它在普通进程上使用的是完全公平调度算法 CFS，对于实时进程使用 FIFO 或 RR 调度算法（虽然 Linux 不是硬实时系统）。下面我们来讨论 Linux 的调度行为。

1.1. 调度与均衡

在单处理器/单核系统上可以只讨论调度问题，但是在多处理器/多核系统上还需要考虑各个处理器或核之间的负载均衡，否则无法获得理想的系统利用率。

如果读者使用的是虚拟机环境并且设置为单核，则需要将虚拟机设置为多核才能较好地完成后面的实践操作。以 VirtualBox 虚拟机为例，首先关闭虚拟机，然后选中虚拟机（本例是 Centos7-OSexp-crash）、点击“设置”按钮、选择“系统”、选择“处理器”，再将处理器数目修改为大于 1 即可（不能超过宿主物理机的核数），如图 1-1 所示。

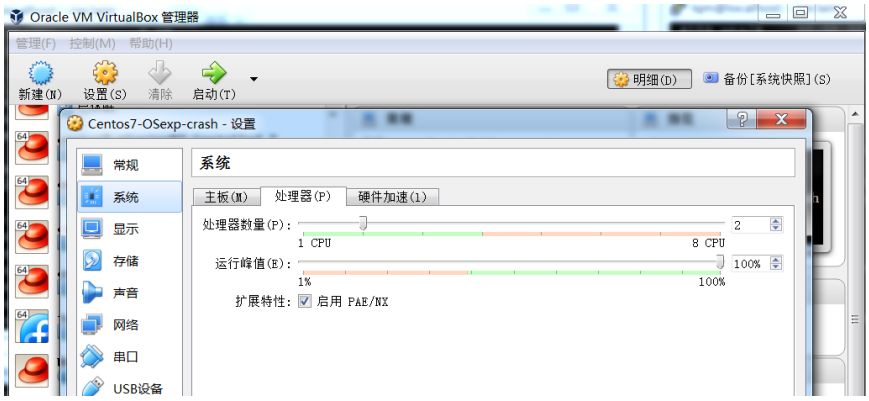

图 1-1 设置 VirtualBox 虚拟机核数

重启系统后，可以从/proc/cpuinfo 中看到多核信息，如屏显 1-1 所示。Linux 操作系统在意的是逻辑 CPU，每个逻辑 CPU 上可以运行一个进程（拥有一个独立的调度队列）并且每个逻辑 CPU 的 processor 编号不同。此处有逻辑 CPU 两个 processor：0 和 processor：1。如果读者不想了解多核中的相关术语可以跳过本段后面的文字。这里有几个处理器相关的概念需要理清，按照层次关系有系统->物理 CPU->物理核->逻辑核的概念。一个系统的主板上可以插有多个物理 CPU（用 physical id 区分），一个物理 CPU 可以包含多个物理核（用 core id 区分），一个物理核可以拥有一个或多个逻辑核。如果没有硬件线程技术则每个物理核就是一个逻辑 CPU，但是如果有硬件线程技术（例如 Intel 的 HT），那么一个物理核就可以有多个逻辑 CPU。

一个物理处理器内的逻辑 CPU 个数用 Siblings 表示。如果有一个以上逻辑 CPU 拥有相同的 core id 和 physical id，则说明系统支持超线程（HT）技术，它们位于同一个物理核上。

屏显 1-1 /proc/cpuinfo 信息

```
[lqm@localhost ~]$ cat /proc/cpuinfo
processor  : 0
vendor_id  : GenuineIntel
cpu family : 6
model      : 60
model name : Intel(R) Core(TM) i7-4790 CPU @ 3.60GHz
stepping   : 3
cpu MHz         : 3591.684
cache size : 8192 KB
physical id     : 0
siblings   : 2
core id         : 0
cpu cores  : 2
apicid          : 0
initial apicid  : 0
fpu        : yes
fpu_exception   : yes
cpuid level     : 13
wp         : yes
flags      : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36 clflush mmx fxsr sse
sse2 ht syscall nx rdtscp lm constant_tsc rep_good nopl xtopology nonstop_tsc pni pclmulqdq ssse3 cx16
sse4_1 sse4_2 x2apic movbe popcnt aes xsave avx rdrand hypervisor lahf_lm abm
bogomips   : 7183.36
clflush size    : 64
cache_alignment : 64
address sizes   : 39 bits physical, 48 bits virtual
power management:

processor  : 1
vendor_id  : GenuineIntel
cpu family : 6
model      : 60
model name : Intel(R) Core(TM) i7-4790 CPU @ 3.60GHz
stepping   : 3
cpu MHz         : 3591.684
cache size : 8192 KB
physical id     : 0
siblings   : 2
core id         : 1
cpu cores  : 2
apicid          : 1
initial apicid  : 1
fpu        : yes
fpu_exception   : yes
cpuid level     : 13
wp         : yes
flags      : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36 clflush mmx fxsr sse
sse2 ht syscall nx rdtscp lm constant_tsc rep_good nopl xtopology nonstop_tsc pni pclmulqdq ssse3 cx16
sse4_1 sse4_2 x2apic movbe popcnt aes xsave avx rdrand hypervisor lahf_lm abm
bogomips   : 7183.36
clflush size    : 64
cache_alignment : 64
address sizes   : 39 bits physical, 48 bits virtual
power management:
```

将虚拟机配置为多核系统并启动后，top 命令也会有些不同——按“1”键可以查看各个核的使用情况信息。例如 top 命令查看系统中按 CPU 使用率排序的进程列表时，通常如屏显 1-2 所示，前 5 行是系统整体的统计信息，后面是各进程的统计信息。从第一行可以看出，该系

统已经启动运行了 12 分钟，有 3 个用户登录（同一个帐号多次登录计为多个），1/5/15 分钟内的平均负载 load average 为 1.0/0.91/0.56。第二行表明任务总数 tasks 为 178，就绪 running 进程 3 个，175 个进程处于阻塞状态，处于暂停/被跟踪 stop 状态的进程数为 0，处于僵尸 zombie 状态的进程数为 0。以“%Cpu(s)”标志开头的第三行是 CPU 使用情况的统计，默认是将所有核的利用率一起统计。其中 53.1 us 表示 53.1%的时间用于运行用户空间的代码，相应地有 sy 对应运行内核代码占用 CPU 时间的百分比、ni 表示低优先级用户态代码占用 CPU 时间的百分比、id 表示空闲（运行 idle 任务进程）CPU 时间的百分比、wa 表示 IO 等待占用 CPU 时间的百分比、hi 表示硬件中断（Hardware IRQ）占用 CPU 时间的百分比、si 表示软中断（Software Interrupts）占用 CPU 时间的百分比以及和虚拟化有关的 st（steal time）占用 CPU 时间的百分比。

第 4 行和第 5 行是内存相关的统计，其内容和 free 命令的输出完全相同，请参见**错误!未找到引用源。**及相关说明。

屏显 1-2 后面是各个进程的统计信息：PR 和 NI 是优先级和 NICE 值，S 是进程的状态，它们与调度有关；%CPU 是该进程占用单个 CPU 时间的百分比；TIME 是进程在 CPU 上运行的时间（不计阻塞时间）。VIRT、RES 和 SHR 与调度没有直接关系，分别是进程虚存空间大小、物理内存占用量和共享物理内存大小（按 KB 统计），%MEM 指出该进程占系统物理内存总量的百分比。

屏显 1-2 top 命令的输出

```
top - 21:48:40 up 12 min,  3 users,  load average: 1.00, 0.91, 0.56
Tasks: 178 total,   3 running, 175 sleeping,   0 stopped,   0 zombie
%Cpu(s): 53.1 us,  0.3 sy,  0.0 ni, 46.6 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
KiB Mem :  1883672 total,   858304 free,   517724 used,   507644 buff/cache
KiB Swap:  1679356 total,  1679356 free,        0 used.  1170184 avail Mem

  PID USER      PR  NI    VIRT    RES    SHR S  %CPU %MEM     TIME+ COMMAND
 3866 lqm       20   0    4160    344    272 R 100.0  0.0  10:31.61 HelloWorld+
 3088 lqm       20   0 1750180 203924  49232 S   3.0 10.8   0:08.87 gnome-shell
 1188 root      20   0  258308  39620  10156 R   2.3  2.1   0:03.42 Xorg
 3720 lqm       20   0  575076  24640  14428 S   0.7  1.3   0:01.13 gnome-term+
  646 root      20   0    4368    592    496 S   0.3  0.0   0:01.21 rngd
 2815 lqm       20   0   35860   2204    948 S   0.3  0.1   0:00.17 dbus-daemon
 3046 lqm       20   0 1094168  24184  16112 S   0.3  1.3   0:00.32 gnome-sett+
 3252 lqm       20   0  880956  27784  17612 S   0.3  1.5   0:00.30 nautilus
 3869 lqm       20   0  157708   2268   1564 R   0.3  0.1   0:00.66 top
    1 root      20   0  193632   6724   3964 S   0.0  0.4   0:01.88 systemd
    2 root      20   0       0      0      0 S   0.0  0.0   0:00.00 kthreadd
    3 root      20   0       0      0      0 S   0.0  0.0   0:00.01 ksoftirqd/0
    5 root       0 -20       0      0      0 S   0.0  0.0   0:00.00 kworker/0:+
    6 root      20   0       0      0      0 S   0.0  0.0   0:00.13 kworker/u4+
    7 root      rt   0       0      0      0 S   0.0  0.0   0:00.00 migration/0
    8 root      20   0       0      0      0 S   0.0  0.0   0:00.00 rcu_bh
```

在屏显 1-2 上可以看到，进程 3866 占有单个处理器核的 100%利用率（占用了双核系统的一半 CPU 资源），且调度/运行状态为 R（就绪/运行）。同为 R 状态的进程还有 1188 和 3869 号进程，其他进程都处于 S 阻塞状态。

如果按数字“1”键将会分别显示各处理器的统计信息，可以不用查看/proc/cpuinfo 也获得处理器个数的信息。例如在作者的系统上执行 top 命令并按“1”键，分别显示 CPU0 和 CPU1 各自的统计信息如屏显 1-3 所示。

屏显 1-3 top 命令的输出（展开 CPU 利用率）

```
top - 22:20:05 up 44 min,  3 users,  load average: 0.92, 0.96, 0.93
Tasks: 179 total,   3 running, 176 sleeping,   0 stopped,   0 zombie
%Cpu0  :100.0 us,  0.0 sy,  0.0 ni,  0.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu1  : 18.5 us,  2.0 sy,  0.0 ni, 79.5 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
KiB Mem :  1883672 total,   846520 free,   529288 used,   507864 buff/cache
KiB Swap:  1679356 total,  1679356 free,        0 used.  1158508 avail Mem

  PID USER      PR  NI    VIRT    RES    SHR S  %CPU %MEM     TIME+ COMMAND
 3866 lqm       20   0    4160    344    272 R  99.7  0.0  41:54.92 HelloWorld+
 3088 lqm       20   0 1755876 210224  49360 S  12.9 11.2   0:18.42 gnome-shell
......
```

读者准备好多核系统后，下面进入调度与均衡的讨论。

1.1.1. 调度与均衡框架

就绪进程在 Linux 内核中的组织分成若干层次，首先是按各处理器（逻辑 CPU）组织成不同的运行队列 rq（run queue），在每个处理器的 rq 队列内部又按紧急程度组织成四种“调度类”，各种调度类管理自己的就绪进程。各种调度类内部的进程间使用优先级进一步区分各自的重要程度。

进程调度解决的就是如何从这些就绪进程队列中各自选择一个到对应 CPU 上去运行，而负载均衡就是在各个处理器上负载严重不均的时候将繁忙处理器 rq 上的进程迁移到空闲 CPU 的 rq 队列之上，因此 rq 是调度和负载均衡的基础数据结构。上述工作可以简单地用图 1-2 表示：

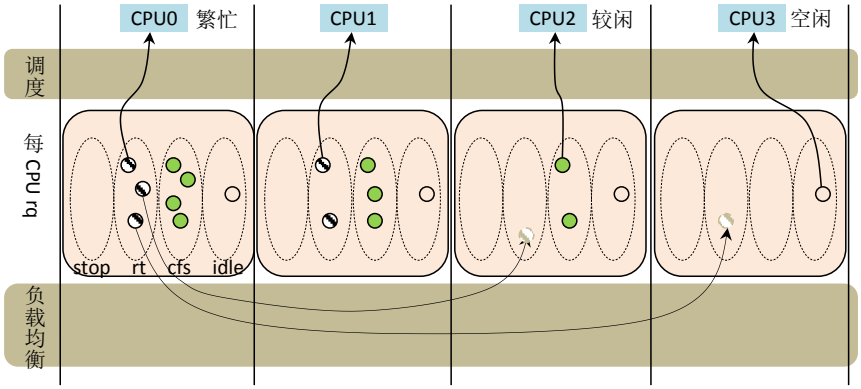

图 1-2 就绪进程组织与调度、均衡示意图

图 1-2 展示了 4 个 CPU 的系统上的例子，每个 CPU 有各自的 rq 就绪队列，每个 CPU 上的 rq 管理着四种类型的任务——分别对应 STOP、RT、CFS 和 IDLE 调度类，优先级依次递减。实时任务属于 RT 调度类，普通进程属于 CFS 完全公平调度类。调度程序从各个 rq 数据结构中挑选一个进程作为对应 CPU 运行的任务，在合适的时机将现有任务撤下 CPU 而换上另一个更紧迫的任务。由于图 1-2 例子中没有 STOP 类进程，因此 CPU0/1 上执行的是实时 RT 类任务，而 CPU2 上执行的是普通 CFS 类任务，CPU3 上没有有效进程因此执行 IDLE 类任务。随着时间

的推进，各个处理器会交替地执行本地的各个就绪进程。对于 CPU3 处于空闲 IDLE 状态，负载均衡器会将其他 CPU 上的就绪进程迁移过来，避免处理器的忙闲不一的状态。此例子中 CPU0 或 CPU1 上的高优先级实时任务可通过负载均衡迁移到到 CPU2/3 上。

我们通常只能创建 RT 类和 CFS 类的进程，前者是实时进程后者是普通进程。

1.1.2. 全系统的调度统计

Linux 在/proc/sched_debug 中给出了整个系统范围的调度相关统计信息，如屏显 1-4 所示。其内容分成多个小节，前面是系统整体信息，后面按逻辑 CPU 分成多个部分。由于我们的系统是双核系统，因此屏显 1-4 中相应地有 cpu#0 和 cpu#1 两部分的统计信息，内部再分成逻辑 CPU 整体信息和 CFS 运行队列、RT 运行队列和本逻辑 CPU 上的就绪进程列表三个小节。

对于 CPU0 有 CFS 类型的 cfs_rq[0]和 RT 类型的 rt_rq[0]两个不同调度类型的进程队列统计信息。本例子中 cfs_rq[0]上有就绪的普通 CFS 进程 nr_running=3 个，而 rt_rq[0]上还没有就绪的实时进程 rt_nr_running=0。curr->pid 指出当前正在 CPU0 上运行的是 4696 号进程。在逻辑 CPU 整体统计中还给出了即时负载信息 load、历史负载信息 cpu_load[0]~[4]和进行下一次负载均衡的时间 next_balance，这些是负载均衡的重要依据。CPU0 上就绪进程则在“runnable_tasks:”标签的后面，各列信息分别是：switches 切换次数、prio 优先级、wait-time 等待 IO 的时间、sum-exec 运行时间以及 sum-sleep 阻塞时间。

对应于 CPU1 则有 cfs_rq[1]和 rt_rq[1]等内容。

屏显 1-4 /proc/sched_debug 信息

```
[lqm@localhost ~]$ cat /proc/sched_debug
Sched Debug Version: v0.11, 3.10.0-514.el7.x86_64 #1
ktime                                   : 3914668.820838
sched_clk                               : 3913632.013169
cpu_clk                                 : 3913632.013216
jiffies                                 : 4298581965
sched_clock_stable()                    : 1

sysctl_sched
  .sysctl_sched_latency                    : 12.000000
  .sysctl_sched_min_granularity            : 10.000000
  .sysctl_sched_wakeup_granularity         : 15.000000
  .sysctl_sched_child_runs_first           : 0
  .sysctl_sched_features                   : 77435
  .sysctl_sched_tunable_scaling            : 1 (logaritmic)

cpu#0, 3591.684 MHz                                  0 号处理器的统计信息
  .nr_running                    : 3                   本处理器上的就绪进程个数（包括正在运行的）
  .load                          : 3072                本处理器上的负载
  .nr_switches                   : 202197              本处理器上累计的进程切换次数
  .nr_load_updates               : 3587360
  .nr_uninterruptible            : 5                   本处理器上进入 uninterruptible 的进程数
  .next_balance                  : 4298.582045         本处理器下次执行负载均衡的时间
  .curr->pid                     : 4696                当前运行的进程 pid
  .clock                         : 3913632.398703
  .cpu_load[0]                   : 3072                本处理器上的历史 load
  .cpu_load[1]                   : 2560                本处理器上的历史 load
  .cpu_load[2]                   : 2302                本处理器上的历史 load
  .cpu_load[3]                   : 2109                本处理器上的历史 load
  .cpu_load[4]                   : 1836                本处理器上的历史 load
```

```
  .avg_idle                      : 1000000
  .max_idle_balance_cost         : 500000

cfs_rq[0]:/                                                    0号处理器上CFS队列的统计信息
  .exec_clock                    : 0.000000
  .MIN_vruntime                  : 3578565.340659
  .min_vruntime                  : 3578571.340659
  .max_vruntime                  : 3578571.341220
  .spread                        : 6.000561
  .spread0                       : 0.000000
  .nr_spread_over                : 0
  .nr_running                    : 3                       本处理器CFS队列上的就绪进程
  .load                          : 3072
  .runnable_load_avg             : 2046
  .blocked_load_avg              : 0
  .tg_load_avg                   : 0
  .tg_load_contrib               : 0
  .tg_runnable_contrib           : 0
  .tg->runnable_avg              : 0
  .tg->cfs_bandwidth.timer_active: 0
  .throttled                     : 0
  .throttle_count                : 0
  .avg->runnable_avg_sum         : 46365
  .avg->runnable_avg_period      : 46365

rt_rq[0]:/
  .rt_nr_running                 : 0
  .rt_throttled                  : 0
  .rt_time                       : 0.000000
  .rt_runtime                    : 950.000000

runnable tasks:
            task   PID         tree-key  switches  prio     wait-time             sum-exec        sum-
sleep
-------------------------------------------------------------------------------------------------------------
     ksoftirqd/0     3   3577023.584799      1291   120         0.000000        18.230017     0.000000 0 /
    kworker/0:0H     5      2007.658749         6   100         0.000000         0.013999     0.000000 0 /
     migration/0     7         0.000000       459     0         0.000000         3.833287     0.000000 0 /

           ......

            bash  3766     34246.354016       103   120         0.000000        69.361223     0.000000 0 /
 HelloWorld-loop  3866   3578571.341220     16776   120         0.000000   3765740.589719     0.000000 0 /
     kworker/0:0  4033   1525959.942626      1028   120         0.000000        85.378501     0.000000 0 /
     kworker/0:2  4165   3578565.340659      4067   120         0.000000       350.751810     0.000000 0 /
R            cat  4696   3578571.576358         0   120         0.000000         5.125432     0.000000 0 /

cpu#1, 3591.684 MHz                                    1号处理器的统计信息
  .nr_running                    : 1
  .load                          : 1024
  .nr_switches                   : 785504

  ......


cfs_rq[1]:/

  ......

rt_rq[1]:/
  .rt_nr_running                 : 0
  .rt_throttled                  : 0
  .rt_time                       : 0.000000
  .rt_runtime                    : 950.000000
```

```
runnable tasks:
            task   PID         tree-key  switches  prio     wait-time             sum-exec        sum-
sleep
------------------------------------------------------------------------------------------------------------
         systemd     1    282499.295959      4443   120         0.000000      2090.978241      0.000000 0 /
        kthreadd     2    282563.909370       144   120         0.000000         3.268203      0.000000 0 /
......
```

另外，在/proc/schedstat 中有其他的调度统计信息，但是由于没有自带标注，因此难以解读。更多信息可以参考 Linux 源代码目录中的文档子目录 linux-3.10.0/Documentation/scheduler/下的 sched-stats.txt。在屏显 1-5 cpu<x> 后面的 9 个数字中，后三个非零的数值分别是该处理器上进程执行时间总和（以 jiffies 计）、进程等待运行的时间总和（以 jiffies 计）以及时间片总数。domain<x> 后面的两个数字是 CPU 掩码，3 表示“11”该调度域包含 cpu0 和 cpu1 两个处理器。

屏显 1-5 /proc/schedstat

```
[lqm@localhost ~]$ cat /proc/schedstat
version 15
timestamp 4317490556
cpu0 0 0 0 0 0 0 17470898269689 145575976930 627107
domain0 3 3 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
cpu1 0 0 0 0 0 0 1059838105220 226443483586 1973678
domain0 3 3 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
[lqm@localhost ~]$
```

1.2. 进程状态及其转变

在观察调度行为前，我们还需要学习一下 Linux 中进程的状态表示。Linux 是一个多用户、多任务的系统，可以同时运行多个用户的多个程序，就必然会产生很多的进程共享地使用 CPU——拥有 CPU 的可以运行而未能获得 CPU 的进程就只能暂停运行。也就是说，每个进程除了正在运行（拥有 CPU）外还有会有其他不同的状态。其中只有就绪的进程才能可能获得调度运行，它们的状态在 Linux 中都是 TASK_RUNNING（并不区分正在 CPU 上运行还是在就绪队列上等待调度）。

1.2.1. 进程状态

在 Linux 内核代码中，这些状态的具体编码如代码 1-1，我们将常见状态做个简单分析。

代码 1-1 进程状态 (linux-3.13/include/linux/sched.h)

```
135 #define TASK_RUNNING            0
136 #define TASK_INTERRUPTIBLE      1
137 #define TASK_UNINTERRUPTIBLE    2
138 #define __TASK_STOPPED          4
139 #define __TASK_TRACED           8
140 /* in tsk->exit_state */            以下两个状态出现在 task_struct->exit_state
141 #define EXIT_ZOMBIE             16
142 #define EXIT_DEAD               32
143 /* in tsk->state again */
144 #define TASK_DEAD               64
```

```
145 #define TASK_WAKEKILL [2]          128
146 #define TASK_WAKING               256
147 #define TASK_PARKED               512   在 get_task_state()返回 TASK_INTERRUPTIBLE
148 #define TASK_STATE_MAX            1024
```

TASK_RUNNING

可执行状态，包括就绪和正在 CPU 上执行。只有在该状态的进程才可能在 CPU 上运行。同一时刻可能有多个进程处于可执行的就绪状态，多核平台上有超过一个的进程正运行在 CPU 上。这些进程的 task_struct 结构（进程控制块）被挂入对应 CPU 的可执行队列（运行队列、就绪队列）中，一个进程最多只能出现在一个 CPU 的运行队列中。很多操作系统教科书将正在 CPU 上执行的进程定义为 RUNNING 状态、而将可执行但是尚未被调度执行的进程定义为 READY 状态，这两种状态在 Linux 下统一为 TASK_RUNNING 状态，对应的状态编码数值为 0。用 ps 命令或/proc/PID/status 查看进程时，可执行状态的进程显示为 R。

TASK_INTERRUPTIBLE

可中断的睡眠状态，也是操作系统课程中所谓的阻塞状态。处于这个状态的进程因为等待某事件的发生（通常是 IO 操作，比如等待 socket 连接、等待信号量）而被阻塞。这些进程的 task_struct 结构从运行队列中取下并放入对应事件的等待队列中。当所等待的事件发生时（由外部中断触发、或由其他进程触发），对应的等待队列中的一个或多个进程将被唤醒，重新挂回到就绪队列中。通过 ps 命令我们会看到，除非机器的负载很高，一般情况下进程列表中的绝大多数进程都处于 TASK_INTERRUPTIBLE 状态（ps 相应的输出显示为 S）。

TASK_UNINTERRUPTIBLE

不可中断的睡眠状态。与 TASK_INTERRUPTIBLE 状态类似，进程处于阻塞状态，但是此刻进程是不会因信号的到来而唤醒。绝大多数情况下，进程处在睡眠状态时，总是应该能够响应异步信号的。由于不响应信号，所以即使用 kill -9 命令也杀不死这样的进程了。这时用 ps 命令或/proc/PID/status 查看进程状态时显示为 D 状态。

而 TASK_UNINTERRUPTIBLE 状态存在的意义就在于，内核的某些处理流程是不能被打断的。如果响应异步信号，程序的执行流程中就会被插入一段用于处理异步信号的流程，于是原有的流程就被中断了。在进程对某些硬件进行操作时（比如进程调用 read 系统调用对某个设备文件进行读操作最终执行到相应设备驱动的代码，并与对应的物理设备进行交互），可能需要使用 TASK_UNINTERRUPTIBLE 状态对进程进行保护，以避免进程与设备交互的过程被打断，造成设备陷入不可控的状态。该状态的进程只能用 wake_up()函数唤醒（例如驱动程序的中断处理代码中发出调用）。这种情况下的 TASK_UNINTERRUPTIBLE 状态总是非常短暂的，通过 ps 命令基本上不可能捕捉到。

Linux 系统中也存在容易捕捉的 TASK_UNINTERRUPTIBLE 状态。执行 vfork 系统调用后，父进程将进入 TASK_UNINTERRUPTIBLE 状态，直到子进程调用 exit()或 execve()。

[2] 这个不是基本状态，表示即使在 TASK_UNINTERRUPTIBLE 状态下也可以响应致命信号（终止进程）

TASK_STOPPED 或 TASK_TRACED

暂停状态或跟踪状态。通过 ctrl-z 向进程发送一个 SIGSTOP 信号，它就会因响应该信号而进入 TASK_STOPPED 状态(除非该进程本身处于 TASK_UNINTERRUPTIBLE 状态而不响应信号)。SIGSTOP 与 SIGKILL 信号一样是强制的，不允许用户进程通过 signal 系列的系统调用重新设置相应的信号处理函数。向进程发送一个 SIGCONT 信号，可以让其从 TASK_STOPPED 状态恢复到 TASK_RUNNING 状态。用 ps 命令或/proc/PID/status 查看这类进程显示的是 T 状态。后台进程执行 getchar()等阻塞操作时会进入 T 状态。

当进程正在被跟踪时，它处于 TASK_TRACED 这个特殊的状态。“正在被跟踪”指的是进程暂停下来，等待跟踪它的进程对它进行操作。比如在 gdb 中对被跟踪的进程设置一个断点，进程在断点处停下来的时候就处于 TASK_TRACED 状态。而在其他时候，被跟踪的进程还是处于前面提到的那些状态。

对于进程本身来说，TASK_STOPPED 和 TASK_TRACED 状态很类似，都是表示进程暂停下来。而 TASK_TRACED 状态相当于在 TASK_STOPPED 之上多了一层保护，处于 TASK_TRACED 状态的进程不能响应 SIGCONT 信号而被唤醒。只能等到调试进程通过 ptrace 系统调用执行 PTRACE_CONT、PTRACE_DETACH 等操作（通过 ptrace 系统调用的参数指定操作），或调试进程退出，被调试的进程才能恢复 TASK_RUNNING 状态。

TASK_DEAD（-EXIT_ZOMBIE）

退出状态，且成为僵尸进程。进程在退出的过程中，处于 TASK_DEAD 状态。在这个退出过程中，进程占有的所有资源将被回收——除了 task_struct 结构（以及少数资源）以外。于是进程就只剩下 task_struct 这么个空壳，故称为僵尸。之所以保留 task_struct，是因为 task_struct 里面保存了进程的退出码、以及一些统计信息。而其父进程很可能会关心这些信息。比如在 shell 中，$?变量就保存了最后一个退出的前台进程的退出码，而这个退出码往往被作为 if 语句的判断条件。保留完整的 task_struct 结构而不是仅仅是退出状态，因为在内核中已经建立了从 pid 到 task_struct 查找关系，还有进程间的父子关系，便于父进程查找。

父进程通过 wait 系列的系统调用（如 wait4、waitid）来等待某个或某些子进程的退出，并获取它的退出信息。然后父进程的 wait 系列系统调用会将子进程的尸体（task_struct）也释放掉。子进程在退出的过程中，内核会给其父进程发送一个信号，通知父进程来收拾残留资源。这个信号默认是 SIGCHLD，但是在通过 clone 系统调用创建子进程时，可以设置这个信号。

只要父进程不退出且没有对已结束的子进程执行 wait 系统调用，这个子进程就处于僵尸状态（参见**错误!未找到引用源。**）并一直持有 task_struct。但是如果父进程结束运行，会将它的所有子进程都托管给别的进程（使之成为别的进程的子进程）——可以是退出进程所在进程组的下一个进程（如果存在的话）或者是 1 号 init 进程，由 init 进程消灭僵尸进程。

用 ps 命令或/proc/PID/status 查看这些进程的时候显示的是 Z 状态。

TASK_DEAD（-EXIT_DEAD）

退出状态，且进程即将被销毁。进程在退出过程中也可能不会保留它的 task_struct，比如这个进程是多线程程序中被 detach 过的线程。或者父进程通过设置 SIGCHLD 信号的 handler 为 SIG_IGN 显式的忽略了 SIGCHLD 信号，子进程结束后将被置于 EXIT_DEAD 退出状态，这意味着

接下来的内核代码立即就会将该进程彻底释放。所以 EXIT_DEAD 状态是非常短暂的，几乎不可能通过 ps 命令捕捉到（显示为 X 状态）。

ps 命令查看进程状态

我们用 ps -aux 查看系统中全部进程的状态获得如屏显 1-6 的输出，其中 STAT 列是 BSD 的格式进程状态。这些状态符号的具体含义可以用 man ps 命令查看到，具体如下：D 表示不可中断睡眠 Uninterruptible sleep（通常由于 IO）、R 表示正在运行或在就绪队列中、S 处于可中断睡眠状态、T 表示由作业控制信号停止、t 表示被跟踪、Z 表示僵尸状态、W 表示换出（从内核 2.6 开始无效）、X 表示死掉的进程（几乎不可能观察到）。另外有一些 BSD 风格的修饰符号：<表示比默认优先级高（NICE 值小于 0）、N 表示比默认优先级低（NICE 值大于 0）、L 表示该进程有些页被锁进内存、s 表示会话组长、+表示位于前台的进程组、l 表示多线程（例如用 NPTL）。屏显 1-6 给出了多种不同的进程状态，僵尸状态的进程可参见**错误!未找到引用源。**所示。

屏显 1-6 ps –aux 查看进程状态

```
[lqm@localhost ~]$ ps -aux
USER       PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root         1  0.0  0.3 193632  6724 ?        Ss   21:35   0:02 /usr/lib/systemd/systemd --switched-root
--system --deserialize 21
root         2  0.0  0.0      0     0 ?        S    21:35   0:00 [kthreadd]
root         3  0.0  0.0      0     0 ?        S    21:35   0:00 [ksoftirqd/0]
root         5  0.0  0.0      0     0 ?        S<   21:35   0:00 [kworker/0:0H]
root         7  0.0  0.0      0     0 ?        S    21:35   0:00 [migration/0]
root         8  0.0  0.0      0     0 ?        S    21:35   0:00 [rcu_bh]
root         9  0.0  0.0      0     0 ?        R    21:35   0:00 [rcu_sched]
…
root        32  0.0  0.0      0     0 ?        SN   21:35   0:00 [ksmd]
…
root       476  0.0  0.1  36936  2956 ?        Ss   21:35   0:00 /usr/lib/systemd/systemd-journald
…
root       625  0.0  0.0  55416  1716 ?        S<sl 21:35   0:00 /sbin/auditd -n
…
root       646  0.0  0.0   4368   592 ?        Rs   21:35   0:01 /sbin/rngd -f
root       648  0.0  0.0  16808  1420 ?        SNs  21:35   0:00 /usr/sbin/alsactl -s -n 19 -c -E
ALSA_CONFIG_PATH=/etc/alsa/alsactl.conf --initfile=/lib/alsa/init/00main rdaemon
rtkit      656  0.0  0.0 164648  1284 ?        SNsl 21:35   0:00 /usr/libexec/rtkit-daemon
…
root       661  0.0  0.2 387448  3812 ?        Ssl  21:35   0:00 /usr/libexec/accounts-daemon
…
root      1188  0.4  3.0 275860 56576 tty1     Ssl+ 21:36   0:27 /usr/bin/Xorg :0 -background none -
noreset -audit 4 -verbose -auth /run/gdm/auth-for-gdm-QbhOEq/database -seat seat0 -nol
…
lqm       3058  0.0  0.4 560212  7584 ?        S<l  21:37   0:00 /usr/bin/pulseaudio --start --log-
target=syslog
…
lqm       3262  0.0  0.4 470016  9216 ?        SNl  21:37   0:00 /usr/libexec/tracker-miner-user-guides
lqm       3316  0.0  0.6 636728 11352 ?        SNl  21:37   0:00 /usr/libexec/tracker-miner-fs
…
lqm       3866 99.9  0.0   4160   344 pts/1    R+   21:38  95:14 ./HelloWorld-loop
…
root      4938  0.0  0.1 116432  3120 pts/2    Ss+  22:51   0:00 -bash
…
lqm       5272  0.0  0.0 153140  1872 pts/0    R+   23:13   0:00 ps -aux
```

1.2.2. 状态转换

进程刚创建的时候处于可执行就绪状态，然后再运行中根据条件的不同在各个状态之间变化，直到退出。

初始状态

进程是通过 fork 系列的系统调用（fork、clone、vfork）来创建的，内核（或内核模块）也可以通过 kernel_thread()函数创建内核线程。由于父进程调用 fork()时处于 TASK_RUNNING 状态，则子进程自然也处于 TASK_RUNNING 状态。另外，在 clone 系统调用和内核函数 kernel_thread()也接受 CLONE_STOPPED 选项，从而将子进程的初始状态置为 TASK_STOPPED。

状态转换

随着进程运行以及被操作系统所调度，进程往往将在几种状态中转换多次。但是进程状态的变迁主要方向却只有两个——从 TASK_RUNNING 状态变为非 TASK_RUNNING 状态、或者从非 TASK_RUNNING 状态变为 TASK_RUNNING 状态。图 1-3 的中间是 TASK_RUNNING 运行状态——具体再细分为是否占有 CPU，如果占有 CPU 又分为用户态和内核态。两边则是非 TASK_RUNNING 状态。

从 RUNNING 状态转入阻塞睡眠有两种情况，一种是不可中断睡眠，另一个是可中断睡眠。从阻塞睡眠状态到 RUNNING 状态则需要调用 wake_up()来实现。

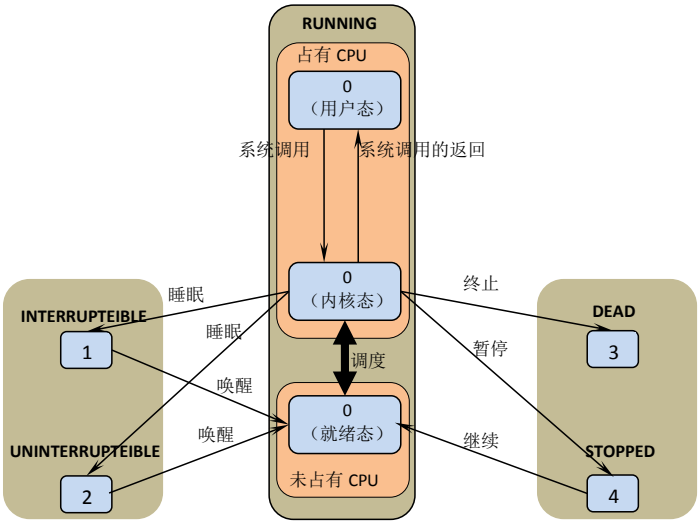

图 1-3 进程状态转换图

操作系统概念上的调度包含了图 1-3 中的所有状态转换，但是后面 1.3 小节讨论调度算法是图 1-3 中关于如何从就绪进程中选择一个进程来使用 CPU 的部分（即图中粗线条的双向箭头）。

可以看出，如果给一个 TASK_INTERRUPTIBLE 状态的进程发送 SIGKILL 信号，这个进程将先被唤醒（进入 TASK_RUNNING 状态），然后再响应 SIGKILL 信号而退出（变为 TASK_DEAD 状态）。并不会从 TASK_INTERRUPTIBLE 状态直接退出。

进程从非 TASK_RUNNING 状态变为 TASK_RUNNING 状态，是由别的进程（也可能是中断处理程序）执行唤醒操作来实现的。执行唤醒的进程设置被唤醒进程的状态为 TASK_RUNNING，然后将其 task_struct 结构加入到某个 CPU 的运行队列中。于是被唤醒的进程将有机会被调度执行。

进程从 TASK_RUNNING 状态变为非 TASK_RUNNING 状态，则有几种途径：1）响应信号而进入 TASK_STOPED 状态或 TASK_DEAD 状态；2）执行系统调用主动进入 TASK_INTERRUPTIBLE 状态（如 nanosleep 系统调用）或 TASK_DEAD 状态（如 exit 系统调用）；3 由于执行系统调用需要的资源得不到满足，而进入 TASK_INTERRUPTIBLE 状态或 TASK_UNINTERRUPTIBLE 状态（如 select 系统调用）。显然，这些情况都只能发生在进程正在 CPU 上执行的情况下。

就绪与阻塞

程序执行中的最基本状态就是运行和阻塞，下面代码 1-2 的 HelloWorld-loop-getchar()将会执行一段循环然后调用 getchar()而阻塞等待按键，不断循环反复执行上述操作。

代码 1-2 HelloWorld-loop-getchar.c

```
#include <stdio.h>

int main()
{
   long i,j,temp;
   printf("HelloWorld!\n");
   while(1)
   {    for (i=0;i<1024*1024*1024;i++)
        {    temp+=1;
        }
        printf("one iteration!\n");
        getchar();
   }
   return 0;
}
```

执行上述程序，并在必要时任意键使得程序的循环得以继续，如屏显 1-7 所示。并在另一个终端上可以用 ps 命令看到 HelloWorld-loop-getchar 进程交替地进入运行和阻塞状态，即进程 8548 的状态交替变为 R+和 S+，如屏显 1-8 所示。

屏显 1-7 HelloWorld-loop-getchar 的运行输出

```
[lqm@localhost ~]$
[lqm@localhost ~]$
[lqm@localhost ~]$ HelloWorld-loop-getchar
HelloWorld!
one iteration!

one iteration!
……
```

屏显 1-8 用 ps 观察 HelloWorld-loop-getchar 进程调度状态变化

```
[lqm@localhost ~]$ ps aux|grep HelloWorld-loop-getchar
lqm       8548 28.6  0.0   4164   344 pts/4    S+   22:15   0:02 HelloWorld-loop-getchar
```

```
lqm       8550  0.0  0.0 112664   976 pts/3    R+   22:15   0:00 grep --color=auto HelloWorld-loop-getchar
[lqm@localhost ~]$ ps aux|grep HelloWorld-loop-getchar
lqm       8548 28.8  0.0   4164   344 pts/4    R+   22:15   0:04 HelloWorld-loop-getchar
lqm       8550  0.0  0.0 112664   976 pts/3    R+   22:15   0:00 grep --color=auto HelloWorld-loop-getchar
[lqm@localhost ~]$
```

在解释了就绪和阻塞状态之后，还需要向读者澄清“运行”这个概念。回顾屏显 1-6，可以看出系统在执行 ps 命令的时已经是 23:13，而 PID=4938 的 bash 从 22:51 就开始“启动运行”，但是它的 TIME 列统计仍为“0:00”——Linux 认为它几乎没有运行。也就是说，虽然 bash 程序一直在终端上存在并看起来是在“运行”着，但实际上没有按键的时候，bash 是在阻塞状态的，仅在按键时短暂处理并在输入完整命令行之后执行相关命令，其他时间都是“睡眠”状态！

1.2.3. 进程的调度统计

在/proc/PID/status 和/proc/PID/stat 中含有进程调度和运行有关的一些信息，下面我们简单解读一下。注意，这里是一个进程范围内的调度信息，要屏显 1-4 中整个系统的信息/proc/sched_debug 进行区分和比较。

/proc/PID/status

屏显 1-9 是对应前面 HelloWorld-loop-getchar 进程的/proc/PID/status 信息。从中可以看到进程名为 HelloWorld-loop（后面还有“-getchar”字符没有显示完整），运行调度状态为睡眠 S，线程组号为 8548，进程号是 8548，父进程号为 3725。voluntary_ctxt_switches 是自愿调度切换的次数，也就是 HelloWorld-loop-getchar 主动放弃 CPU 的次数，该处数字 3 与屏显 1-7 中显示的几次循环执行相匹配（getchar()会阻塞而放弃 CPU）；nonvoluntary_ctxt_switches 记录了非自愿的调度切换次数，本例共有 61 次（在内层循环执行期间发生的）。

靠近结末端的 Cpus_allowed 是一个位图，表示该进程可运行的处理器号，此处 3（二进制为 11）表示可以在 0 号和 1 号 CPU 上运行。紧接着的 Cpus_allowed_list 有相同的信息，但是它不是位图，而直接是逻辑 CPU 号的列表。这些 CPU 亲和性信息与后面讨论的负载均衡有关。

/proc/PID/status 中有很多 Vm 开头的项，与该进程内存管理有关，将在**错误!未找到引用源。**讨论。再后面 Sig 打头的项是和信号处理有关统计量。

屏显 1-9 /proc/PID/status

```
Name:HelloWorld-loop                    程序名
State:     S (sleeping)                 调度状态（当前为阻塞/睡眠）
Tgid:8548                               线程组 ID
Ngid:0                                  NUMA 组 ID（非 NUMA 架构则为 0）
Pid:  8548                              进程 ID（LWP 的 ID）
PPid:3725                               父进程 ID
TracerPid:0                             跟踪该进程的进程 ID，0 表示没有被跟踪
Uid:  1000  1000  1000  1000            这两行各列为：  RUID/RGID 执行者;EUID/EGID 有效用户；
Gid:  1000  1000  1000  1000                            SUID/SGID 保存的；FSUID/FSGID 可执行文件用户
FDSize:    256                          文件描述符数组的大小
Groups:    1000                         执行该进程的用户所属用户组
VmPeak:        4232 kB                  进程运行过程中占用虚存空间的峰值
VmSize:        4164 kB                  现在正在占用的虚存空间大小
VmLck:            0 kB                  已经锁住的虚存空间（有物理映射且不能换出）大小
VmPin:            0 kB                  不可换出且物理地址固定的虚存空间大小
VmHWM:          344 kB                  分配到物理内存的峰值
```

```
VmRSS:           344 kB                 现在使用的物理内存
RssAnon:          72 kB                 匿名页帧总量
RssFile:         272 kB                 文件映射页帧总量
RssShmem:          0 kB                 共享物理内存总量
VmData:           56 kB                 进程数据空间的大小
VmStk:           136 kB                 进程堆栈空间的大小
VmExe:             4 kB                 进程代码空间的大小
VmLib:          1880 kB                 进程所使用 LIB 库而占用的虚存空间的大小
VmPTE:            28 kB                 页表占所用空间的大小
VmSwap:            0 kB                 换出的内存空间总量
Threads:  1                             该进程所在线程组中的线程数量
SigQ:0/72358                            待处理信号数量/已处理信号数量
SigPnd:   0000000000000000              屏蔽位,存储了该线程的待处理信号,等同于线程的 PENDING 信号
ShdPnd:   0000000000000000              屏蔽位,存储了该线程组的待处理信号.等同于进程组的 PENDING 信号
SigBlk:   0000000000000000              存放被阻塞的信号,等同于 BLOCKED 信号
SigIgn:   0000000000000000              存放被忽略的信号,等同于 IGNORED 信号
SigCgt:   0000000000000000              存放捕获的信号,等同于 CAUGHT 信号
CapInh:   0000000000000000              表示能够被当前进程执行的程序继承的能力
CapPrm:   0000000000000000              表示进程能够使用的能力,在 cap_permitted 中可以包含 cap_effective
中没有的能力，这些能力是被进程自己临时放弃的,也可以说 cap_effective 是 cap_permitted 的一个子集.
CapEff:   0000000000000000              当一个进程要进行某个特权操作时,操作系统会检查 cap_effective 的对应位是
否有效,而不再是检查进程的有效 UID 是否为 0.
CapBnd:   0000001fffffffff              是系统的边界能力,我们无法改变它
:    0000000000000000
Seccomp:  0
Cpus_allowed:  3                        该进程可以使用 CPU 的亲和性掩码
Cpus_allowed_list:   0-1                指出该进程可以使用 CPU 的列表
Mems_allowed:
     00000000,00000000,00000000,00000000,00000000,00000000,00000000,00000000,00000000,00000000,00
000000,00000000,00000000,00000000,00000000,00000000,00000000,00000000,00000000,00000000,00000000,
00000000,00000000,00000000,00000000,00000000,00000000,00000000,00000000,00000001
Mems_allowed_list:   0
voluntary_ctxt_switches:  3             进程主动切换的次数
nonvoluntary_ctxt_switches:    61       进程被动切换的次数
```

/proc/PID/stat

在/proc/PID/stat 中记录了另外的一些进程活动的信息，该文件中的所有统计值都是从进程启动开始累计到当前时刻。由于只有数字而没有格式和提示，因此解读比较困难。读者可以用 man 5 proc 命令查看各列的内容解释。

屏显 1-10 /proc/PID/stat 内容

```
[lqm@localhost 4203]# cat /proc/6195/stat
6195 (HelloWorld-loop) R 15221 6195 15221 34818 6195 4202496 177 0 0 0 1891 0 0 0 20 0 1 0 34880339
4259840 86 18446744073709551615 4194304 4196100 140724643835760 140724643835368 4195643 0 0 0 0 0 0 17 3
0 0 0 0 0 6295056 6295604 13881344 140724643841141 140724643841157 140724643841157 140724643844070 0
[lqm@localhost 4203]#
```

我们将相关数据逐项解释，由于有些数据还没有讨论过，因此后面章节会返回来参考这些数据和解释。

pid (1)=6195 进程（包括轻量级进程，即线程）号

comm(2)=HelloWorld-loop 应用程序的名字 HelloWorld-loop（一个循环计算 k+=1 的程序）

task_state(3)=R 程序运行状态 R:runnign，S:sleeping（TASK_INTERRUPTIBLE），D:disk sleep（TASK_UNINTERRUPTIBLE），T: stopped，Z:zombie，X:dead

ppid(4)=15221 父进程 ID

pgid(5)=6195 线程组号

sid(6)=15221 该任务所在的会话组 ID

tty_nr(7)=34818 (pts/1) 该任务的 tty 终端的设备号，INT（34817/256）=主设备号，（34817-主设备号）=次设备号

tty_pgrp(8)=6195 终端的进程组号，当前运行在该任务所在终端的前台任务（包括 shell 应用程序）的 PID。

task->flags(9)=4202496 进程标志位，该任务的特性（参见《Linux 技术内幕》191 页代码 4-37）

min_flt(10)=177 该任务不需要从硬盘拷数据而发生的缺页（次缺页）的次数

cmin_flt(11)=0 累计的该任务的所有的 waited-for 进程曾经发生的次缺页的次数目

maj_flt(12)=0 该任务需要从硬盘拷数据而发生的缺页（主缺页）的次数

cmaj_flt(13)=0 累计的该任务的所有的 waited-for 进程曾经发生的主缺页的次数目

utime(14)=1891 该任务在用户态运行的时间，单位为 jiffies

stime(15)=0 该任务在核心态运行的时间，单位为 jiffies

cutime(16)=0 累计的该任务的所有的 waited-for 进程曾经在用户态运行的时间，单位为 jiffies

cstime(17)=0 累计的该任务的所有的 waited-for 进程曾经在核心态运行的时间，单位为 jiffies

priority(18)=20 任务的动态优先级

nice(19)=0 任务的 NICE 值

num_threads(20)=1 该任务所在的线程组里线程的个数

it_real_value(21)=0 由于计时间隔导致的下一个 SIGALRM 发送进程的时延，单位为 jiffies

start_time(22)=34880339 该任务启动的时间（从系统启动算起），单位为 jiffies

vsize(23)=4259840 该任务的虚拟地址空间大小，按 page 计

rss(24)=86 该任务当前驻留物理地址空间的大小，按 page 计。这些页可能用于代码，数据和栈

rsslim(25)=18446744073709551615（bytes） 该任务能驻留物理地址空间的最大值

start_code(26)=4194304 该任务在虚拟地址空间的代码段的起始地址

end_code(27)=4196100 该任务在虚拟地址空间的代码段的结束地址

start_stack(28)=140724643835760 该任务在虚拟地址空间的栈的结束地址

kstkesp(29)=140724643835368 esp(32 位堆栈指针) 的当前值

Kstkeip(30)=4195643 指向将要执行的指令的指针, EIP(32 位指令指针)的当前值

Pendingsig(31)=0 待处理信号的位图，记录发送给进程的普通信号

block_sig(32)=0 阻塞信号的位图

sigign(33)=0 忽略的信号的位图

sigcatch(34)=0 被俘获的信号的位图

wchan(35)=0 如果该进程是睡眠状态，该值给出调度的调用点

nswap(36)=0 被 swapped 的页数，当前没用

cnswap(37)=0 所有子进程被 swapped 的页数的和，当前没用

exit_signal(38)=17 该进程结束时，向父进程所发送的信号

task_cpu(39)=3 上一次调度时运行在哪个 CPU 上
task_rt_priority(40)=0 实时进程的相对优先级别
task_policy(41)=0 进程的调度策略，0=非实时进程，1=FIFO 实时进程；2=RR 实时进程
delayacct_blkio_ticks(42)=0 因块设备 IO 阻塞的时间
guest_time(43)=0 进程的 geust 时间（用于运行虚拟 CPU 的 guest OS），以时钟 tick 计
cguest_time (44)=0 该进程的子进程的 guest 时间
start_data (45)=6295056 数据（data/bss）起始地址
end_data (46)=6295604 数据（data/bss）结束地址
start_brk (47)=13881344 堆区的当前的高端地址
arg_start (48)=140724643841141 命令行及参数字符串起始地址
arg_end (49)=140724643841157 命令行及参数字符串结束地址
env_start (50)=140724643841157 环境变量字符串的起始地址
env_end (51)=140724643844070 环境字符串的结束地址
exit_code (52)=0 该线程的退出码，提供给 waitpid 的进程

1.3. 进程的调度

Linux 首先调度优先级最高的 STOP 类，没有 STOP 任务则会运行 RT 类任务，然后是 CFS 队列上的进程，最后没有就绪进程则运行 idle 进程，甚至进入睡眠状态。

1.3.1. 普通进程的 CFS 调度

Linux 默认创建的进程都是 CFS 调度类的。当 CPU 的运行队列 rq 中没有 STOP 和 RT 类的任务时，才会调度 CFS 队列中的进程。

CFS 调度算法

本小节分析完全公平调度器 CFS（Complete Fair Scheduler），实时进程的调度在 1.3.2 小节讨论。Linux 的早期的普通进程调度器性能不好，后来修改为 O(1)调度器，在 2.6.23 版本之后开始引入完全公平调度 CFS（Completely Fair Scheduler）。CFS 不需要传统概念上的时间片（与教材中通常提到分时系统使用 RR 调度的说法不同）。这种调度器只考虑进程在就绪队列（run queue）中等待了多长时间，其中对 CPU 要求最迫切（被延迟最久、虚拟时间推进最慢）的进程先被调度执行。完全公平调度是由 Ingo Molnar 实现的一种调度类，它取代了 O（1）调度类，并且在处理交互式任务的系统上获得了很高的评价。几乎所有的 CFS 设计都可总结为一句话：它在真实的硬件上模拟了一个理想的、精确的多任务 CPU。

CFS 为每个进程管理一个虚拟时间，调度时总是选取虚拟时间推进最慢的进程来执行——因此进程在执行过程中不断更新自己的虚拟时间并依此判断是否需要切换到更慢的进程。各个进程的虚拟时间推进速度根据优先级高低而有快有慢，优先级高的任务虚拟时间推进慢因此可以占用更多的真实 CPU 时间。

由于 CFS 总是选择虚拟时间最小的进程来运行，因此我们需要了解虚拟时间的相关概念。表 1-1 是虚拟时间推进的示意图，表中三个任务中 Task-1 权重最低 load=1，它的虚拟时间增长的最快（每个 Ti 增 6），Task-2 权重最高 load=3、虚拟时间增长最慢（每个 Ti 增长 2）。表中 Ti=0 的一列表示起点时各自的虚拟时间。后续阴影方格表示当前进程正被调度执行，经过一个 Ti 间隔后虚拟时间推进程度。例如 T1 之后，task1 的虚拟时间增长了 6，从而在三个任务中跑得最前面。于是 T2 需要调度虚拟时间落后的任务（本例是 Task2），经过一个 Ti 后虚拟时钟增长为 2。在 T3 时 Task-3 的虚拟时间最慢，于是被调度运行，T3 结束时它的虚拟时间为 3。依次规则往下，每次都调度“最慢”的任务即可。在 12 个 Ti 内部，Task-1 运行了两段时间占总时间的 1/（1+2+3）=1/6，Task-2 运行了 3 段时间占总时间的 3/（1+2+3）=3/6，Task-3 运行了 4 段时间占总时间的 2/（1+2+3）=2/6，各自与它们的权重相匹配。也可以说，Task-1 的虚拟时间比真实事件快 6 倍，Task-2 的虚拟时间比真实事件快 6/3=2 倍，Task-3 的虚拟时间比真实事件快 6/2=3 倍。表 1-1 只说明了虚拟时间的推进示意情况，让每个任务获得应有的 CPU 计算能力，但实际上 Linux 不是以定长时间片来调度的。实际上不仅定时中断会定期检查它们的虚拟时间推进情况，中途也会因其他异步事件不定期地检查时间虚拟时间推进情况，任务切换没有这么整齐。

表 1-1 vruntime 推进示意

Tasks \ Ti	0	1	2	3	4	5	6	7	8	9	10	11	12	13	…
Task-1(load=1)	0	6	-	-	-	-	-	12	-	-	-	-	-	18	…
Task-2(load=3)	0	-	2	-	4	-	6	-	8	-	10	-	12	-	…
Task-3(load=2)	0	-	-	3	-	6	-	-	-	9	-	12	-	-	…

CFS 调度示例

根据 CFS 调度的本意，不同由优先级的进程，获得 CPU 的真实时间是不同，优先级每相差 1 则 CPU 时间相差固定值（例如 10%）。在前面屏显 1-2 的 top 命令输出中有进程调度优先级有关的信息，即第三列的 PR 和第四列的 NI。其中后者 NI 是 NICE 值（仅对 CFS 进程有效），表示谦让度——越大越谦让、优先级越低，取值范围是-20~19（默认值是 0）；后者 PR 是优先级 priority，直接用于系统调度参考。对于普通 CFS 进程，两个关系是 PR=20+NI。普通进程在 top 命令输出的 PR 列取值为 0~39，其中 0 为 CFS 进程的最高优先级。屏显 1-2 的 5 号进程就是一个 NICE 为-20（最不谦让）、PR 为 0 的优先级最高的 CFS 进程。

下面用一个 shell 脚本来启动两个 HelloWorld-loop，由于为了比较两个进程在同一个 CPU 上竞争 CPU 时间，因此需要将虚拟机设置为单核或者将进程绑定到同一个核上。编辑 Run-NICE.sh 脚本，其内容如代码 1-3 所示。第一行命令按普通进程优先级（默认 NI=0，PR=20）以后台方式运行一个 HelloWorld-loop 进程，然后用 nice 命令[3]对 NICE 提高 10（NI=0+10、PR=20+10=30）并以后台方式运行第二个 HelloWorld-loop 进程。nice 命令后面的数值是对默认 NICE 的调整量，如果不带数字则将进程 NICE 设为 10，只有 root 用户才能将进程 NICE 设置到小于 0 的值。保存后用 chmod a+x Run-NICE.sh 将该文件修改为可执行。

[3] nice 命令可以用-xx 给出调整的数值（降低优先级 xx），也可以用“-n xx”方式直接设定 NICE 值（-20~19）

代码 1-3 Run-NICE.sh

```
./HelloWorld-loop &
nice  -10   ./HelloWorld-loop &
```

然后执行 Run-NICE.sh 脚本，它将创建两个 HelloWorld-loop 进程，各自输出一行“HelloWorld!”，如屏显 1-11 所示。由于是用后台方式运行，因此很快返回到 shell 提示符下。后面用 ps -a 查看这两个进程，进程号分别是 4168 和 4169，两者在 CPU 上获得的执行时间相差约 10 倍左右——见屏显 1-11 中 ps 输出的 TIME 列。三次查看获得的时间比值是 10:1、21:2 和 28:3。

屏显 1-11 执行 Run-NICE.sh 脚本并用 ps –a 查看

```
[lqm@localhost ~]$ ./Run-NICE.sh
HelloWorld!
HelloWorld!
[lqm@localhost ~]$ ps -a
  PID TTY          TIME CMD
 4086 pts/1    00:00:00 top
 4168 pts/0    00:00:10 HelloWorld-loop
 4169 pts/0    00:00:01 HelloWorld-loop
 4173 pts/0    00:00:00 ps
[lqm@localhost ~]$ ps -a
  PID TTY          TIME CMD
 4086 pts/1    00:00:00 top
 4168 pts/0    00:00:21 HelloWorld-loop
 4169 pts/0    00:00:02 HelloWorld-loop
 4178 pts/0    00:00:00 ps
[lqm@localhost ~]$ ps -a
  PID TTY          TIME CMD
 4086 pts/1    00:00:00 top
 4168 pts/0    00:00:28 HelloWorld-loop
 4169 pts/0    00:00:03 HelloWorld-loop
 4182 pts/0    00:00:00 ps
[lqm@localhost ~]$
```

在另一个终端上启动 top 查看，可以看到 4168 号进程的优先级 PR 是 20，而 4169 号进程的优先级为较低的 30，如屏显 1-12 所示。4168 号进程占用了 CPU 的 83.4%的时间，4169 号进程仅占用了 CPU 的 9%的时间。

屏显 1-12 用 top 观察优先权不同的两个进程

```
top - 04:46:25 up 9 min,  3 users,  load average: 2.10, 1.66, 0.91
Tasks: 170 total,   5 running, 165 sleeping,   0 stopped,   0 zombie
%Cpu(s): 90.7 us,  0.0 sy,  9.3 ni,  0.0 id,  0.0 wa,  0.0 hi,  0.0 si,
KiB Mem :  1883804 total,   887716 free,   499852 used,   496236 buff/cac
KiB Swap:  1679356 total,  1679356 free,        0 used.  1189348 avail Me

  PID USER      PR  NI    VIRT    RES    SHR S %CPU %MEM     TIME+
 4168 lqm       20   0    4160    344    272 R 83.4  0.0   1:26.10
 4169 lqm       30  10    4160    344    272 R  9.0  0.0   0:09.24
 3079 lqm       20   0 1503700 187180  49292 S  5.3  9.9   0:16.10
 1107 root      20   0  242664  39216  10044 R  1.3  2.1   0:06.30
……
```

接着从屏显 1-13 比较一下两者的/proc/PID/sched 差异，两个的启动时间 se.exec_start、虚拟时间推进 se.vruntime 都是接近的，但是各自获得 CPU 上运行的时间 se.sum_exec_runtime 差异较大，分别是 830325.948578 和 89197.462713。由于这两个进程一直在做加法，因此从来没有主动让出 CPU，所以它们的自愿切换次数 nr_voluntary_switches 都为 0，进程切换次数 nr_switches 等于被强制切换的次数 nr_involuntary_switches。切换次数的统计也出现在屏显 1-9

的/proc/PID/status 的最后两行。请读者注意两个进程的负载权重 se.load.weight 与优先级相关，分别为 1024 和 110——是负载均衡的重要依据。

屏显 1-13 /proc/PID/sched

```
[lqm@localhost ~]$ cat /proc/4168/sched; cat /proc/4169/sched
HelloWorld-loop (4168, #threads: 1)
-------------------------------------------------------------------
se.exec_start                                :       1428129.855333
se.vruntime                                  :       4052367.360260
se.sum_exec_runtime                          :        830325.948578
se.nr_migrations                             :                    0
nr_switches                                  :                54741
nr_voluntary_switches                        :                    0
nr_involuntary_switches                      :                54741
se.load.weight                               :                 1024
policy                                       :                    0
prio                                         :                  120
clock-delta                                  :                   30
mm->numa_scan_seq                            :                    0
numa_migrations, 0
numa_faults_memory, 0, 0, 1, 0, -1
numa_faults_memory, 1, 0, 0, 0, -1
HelloWorld-loop (4169, #threads: 1)
-------------------------------------------------------------------
se.exec_start                                :       1428143.869869
se.vruntime                                  :       4052382.141077
se.sum_exec_runtime                          :         89197.462713
se.nr_migrations                             :                    0
nr_switches                                  :                42690
nr_voluntary_switches                        :                    0
nr_involuntary_switches                      :                42690
se.load.weight                               :                  110
policy                                       :                    0
prio                                         :                  130
clock-delta                                  :                   36
mm->numa_scan_seq                            :                    0
numa_migrations, 0
numa_faults_memory, 0, 0, 1, 0, -1
numa_faults_memory, 1, 0, 0, 0, -1
[lqm@localhost ~]$
```

在/proc/PID/schedstat 中统计了进程在 CPU 上运行的时间、在就绪队列上等待的时间和获得的时间片。由于只是三个数字而没有带文字标注，因此需要记住其含义才能解读。可以看到两者在 CPU 上运行时间相差约 10 倍，如屏显 1-14 所示。另外这里的时间是以时钟的计数值表示的，而不是按常见时间单位表示的。

屏显 1-14 /proc/PID/schedstat

```
[lqm@localhost ~]$ cat /proc/4168/schedstat; cat /proc/4169/schedstat
790420244637 99485112991 51903
84911253755 805025223235 40646
```

除了用 nice 命令在启动进程时指定优先级外，还可以在进程启动后用 renice 命令调整优先级。它的用法如下 renice priority [[-p] pid ...] [[-g] pgrp ...] [[-u] user ...]，参数 priority 可是取值是-20~19，后面的参数不仅可以用-p 指定要调整优先级的进程，还也以用-u 指定某个用户的所有进程或-g 指定某个用户组的所有进程。只有 root 用户才能用 renice 提升进程 NICE 到小于 0 的数值。

另外可以在代码中用 nice()函数调整 NICE 值，其函数原型为“int nice(int inc);”头文件是 unistd.h。调整优先级的函数还有 setpriority()和 getpriority()等。

1.3.2. 实时进程调度

Linux 虽然声称支持实时进程，但是其实时进程实际上就是优先级较高的进程。Linux 实时进程在进程实体、进程切换机制等方面和普通进程没有差异，仅仅是在调度时因优先级高于普通进程而使得在竞争 CPU 时更有优势，能及早获得运行。但是由于 Linux 自身的中断响应、进程切换的速度都无法达到工业实施系统的要求，因此至多能称为软实时系统。如需要更快的响应速度，可以考虑使用 RT-Linux，在 Linux 和硬件之间加入一层实时处理层，提高其实时任务的切换速度。

实时调度

Linux 的 RT 实时进程优先级比 CFS 进程高，各处理器运行队列 rq 中的实时进程按照优先级 0~99 以 O(1)调度器方式调度——本处理器上的实时进程按照优先级不同挂入不同的子队列中，高优先级的队列先执行完再执行低优先级的队列上的进程。内核使用的优先级为 0~139，其中 0~99 对应于实时进程，100~139 对应 CFS 进程。结合前面所说的普通 CFS 进程的 nice 和 Prority 优先级，以及后面用到的实时进程相关的优先级，将它们合并画到一起如图 1-4 所示。由于不同场合需要使用不同的数值表示，注意它们之间的差异和对应关系。

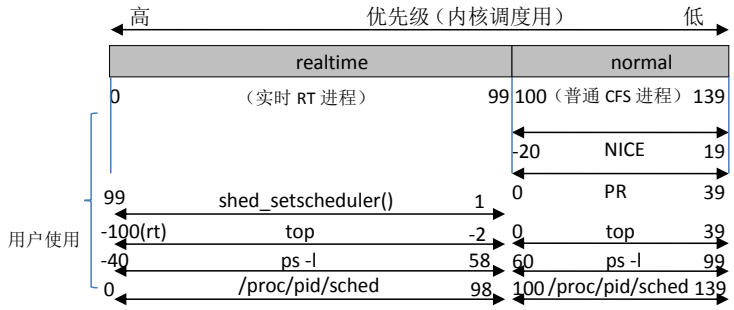

图 1-4 内核中的任务优先级

实时进程可以进一步分为两种调度模式：SCHED_FIFO 和 SCHED_RR。前者的 FIFO 任务执行时不被打断一直到运行结束（除非更高优先级的实时任务来抢占），而后者的 RR 任务在执行一个时间片之后将让出 CPU 给其他相同优先级的 RR 任务实现轮转执行。为了防止实时任务意外地一直占用 CPU，系统对实时任务占用 CPU 的总时间做了一个限制——默认情况下最多只能占 95%。

实时调度仅根据本 CPU 队列的优先级排序，不同队列上的优先级排序问题属于负载均衡的工作。对实时任务而言调度和均衡是紧密联系的，任何一次调度都必须同时满足负载均衡的要求，这个在后面实时负载均衡小节会有体现。

实时任务的均衡原则很简单，对于 N 个处理器的系统，保证优先级最高的 N 个实时任务各自在一个处理器上运行。如果实时任务少于 N 个，则多出来的处理器可以运行普通 CFS 任务。

创建实时进程

创建实时进程比普通进程略有一些区别。可以在普通进程基础之上，通过将调度类从 CFS 转换成 RT 类，并指定优先级即可。相关的一些设置函数有：sched_get_priority_max() 取得静态优先级的上限、sched_get_priority_min()取得静态优先级的下限、sched_getparam() 取得进程的调度参数、sched_getscheduler() 取得指定进程的调度类、sched_rr_get_interval() 取得按 RR 算法调度的实时进程的时间片长度、sched_setparam() 设置进程的调度参数、sched_setscheduler()设置指定进程的调度策略和参数、sched_yield() 进程主动让出处理器并将自己挂入等候调度队列的队尾。使用这些函数时必须在代码中通过“#include <sched.h>”包含 sched.h 头文件。

下面通过 RT-process-demo.c 来展示如何创建 RR 实时进程和 FIFO 实时进程，如代码 1-4 所示。代码核心是通过 sched_setscheduler()来改变进程的调度类型和优先级，第一个参数是进程号（0 表示本进程），第二个参数指出调度类（SCHED_RR/SCHED_FIFO/SCHED_OTHER 三者之一），第三者参数是一个 struct sched_param 类型的调度参数——其 sched_priority 成员可以指定优先级。RT-process-demo.c 首先将自己变成 SCHED_RR 调度类的实时进程，然后变为 SCHED_FIFO，最后又恢复为普通进程。由于代码比较简单，其余内容读者可以自行分析。

代码 1-4 RT-process-demo.c

```
#include <sched.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>

int main()
{
   int rc,current_scheduler_policy;
   struct sched_param my_params;    //将要设置的调度参数

   current_scheduler_policy=sched_getscheduler(0);

   printf("SCHED_OTHER = %d SCHED_FIFO =%d SCHED_RR=%d \n", SCHED_OTHER, SCHED_FIFO,
SCHED_RR);
   printf("the current scheduler = %d \n", current_scheduler_policy);
   printf("press any key to change the current scheduler and priority to SCHED_RR\n");
   getchar();                       //checkpoint 1

   my_params.sched_priority=sched_get_priority_max(SCHED_RR);   // 最高的 RR 实时优先级
   rc=sched_setscheduler(0,SCHED_RR,&my_params);                //设置为 RR 实时进程
   if(rc<0)
   {
       perror("sched_setscheduler to SCHED_RR error");
           exit(0);
   }
   current_scheduler_policy=sched_getscheduler(0);
   printf("the current scheduler = %d \n", current_scheduler_policy);
```

```
    printf("press any key to change the current scheduler and priority to SCHED_FIFO\n");
    getchar();                      //checkpoint 2

    my_params.sched_priority=sched_get_priority_min(SCHED_FIFO); // 最低FIFO实时优先级
    rc=sched_setscheduler(0,SCHED_FIFO,&my_params);              //设置为FIFO实时进程
    if(rc<0)
    {
        perror("sched_setscheduler to SCHED_FIFO error");
        exit(0);
    }
    current_scheduler_policy=sched_getscheduler(0);
    printf("the current scheduler = %d \n", current_scheduler_policy);
    printf("press any key to ange the current scheduler and priority to SCHED_OTHER(CFS)
\n");
    getchar();                      //checkpoint 3

    rc=sched_setscheduler(0,SCHED_OTHER,&my_params);              //设置为普通进程[4]
    if(rc<0)
    {   perror("sched_setscheduler to SCHED_OTHER error");
        exit(0);
    }
    current_scheduler_policy=sched_getscheduler(0);
    printf("the current scheduler = %d \n", current_scheduler_policy);
    printf("press any key to exit\n");
    getchar();                  //checkpoint 4

    return 0;
}
```

以 root 身份运行 RT-process-demo 进程，然后在它的四个 checkpoint 检查点（参见代码 1-4 注释部分）用其他终端执行 ps -al 观察。如屏显 1-15 所示，RT-process-demo 刚开始在 checkpoint 1 处为普通 CFS 进程优先级 PRI=80/NI=0，然后在检查点 2 处变为 RR 实时进程优先级 PRI=-40/NI=-，到检查点 3 处变为 FIFO 实时进程优先级 PRI=58/NI=-，最后到 checkpoint 4 处变回 CFS 普通进程并且优先级恢复到 PRI=80/NI=0。

屏显 1-15 用 ps –al 观察 RT-process-demo 的四个观测点

```
[lqm@localhost ~]$ ps -al
F S   UID   PID  PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
4 S     0  4627  3700  0  80   0 - 53043 wait   pts/0    00:00:00 su
4 S     0  4635  4627  0  80   0 - 29172 wait   pts/0    00:00:00 bash
4 S     0  4868  4635  0  80   0 -  1041 n_tty_ pts/0    00:00:00 RT-process-demo
0 R  1000  4888  3965  0  80   0 - 37233 -      pts/1    00:00:00 ps
[lqm@localhost ~]$ ps -al
F S   UID   PID  PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
4 S     0  4627  3700  0  80   0 - 53043 wait   pts/0    00:00:00 su
4 S     0  4635  4627  0  80   0 - 29172 wait   pts/0    00:00:00 bash
4 S     0  4868  4635  0 -40   - -  1041 n_tty_ pts/0    00:00:00 RT-process-demo
0 R  1000  4897  3965  0  80   0 - 37233 -      pts/1    00:00:00 ps
[lqm@localhost ~]$ ps -al
F S   UID   PID  PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
4 S     0  4627  3700  0  80   0 - 53043 wait   pts/0    00:00:00 su
4 S     0  4635  4627  0  80   0 - 29172 wait   pts/0    00:00:00 bash
4 S     0  4868  4635  0  58   - -  1041 n_tty_ pts/0    00:00:00 RT-process-demo
0 R  1000  4905  3965  0  80   0 - 37233 -      pts/1    00:00:00 ps
[lqm@localhost ~]$ ps -al
F S   UID   PID  PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
4 S     0  4627  3700  0  80   0 - 53043 wait   pts/0    00:00:00 su
```

[4] sched_setscheduler()对 SCHED_OTHER 调度类不能设置优先级。

```
4 S     0  4635  4627  0  80   0 - 29172 wait   pts/0    00:00:00 bash
4 S     0  4868  4635  0  80   0 -  1041 n_tty_ pts/0    00:00:00 RT-process-demo
0 R  1000  4921  3965  0  80   0 - 37233 -      pts/1    00:00:00 ps
```

在屏显 1-15 中，由于 ps -al 没有输出调度类的信息，因此可以通过/proc/PID/sched 来查看调度类的变化情况。如屏显 1-16 所示，调度类的变化是 0-2-1-0 对应于 CFS-RR-FIFO-CFS 的变化顺序，优先级变化顺序为 120-0-98-120 也与代码相一致。

屏显 1-16 用/proc/PID/sched 观察 RT-process-demo 的四个观测点

```
[lqm@localhost ~]$ cat /proc/4868/sched
RT-process-demo (4868, #threads: 1)
-------------------------------------------------------------------
......
policy                                       :                    0
prio                                         :                  120
......
[lqm@localhost ~]$ cat /proc/4868/sched
RT-process-demo (4868, #threads: 1)
-------------------------------------------------------------------
......
policy                                       :                    2
prio                                         :                    0
......
[lqm@localhost ~]$ cat /proc/4868/sched
RT-process-demo (4868, #threads: 1)
-------------------------------------------------------------------
......
policy                                       :                    1
prio                                         :                   98
......
[lqm@localhost ~]$ cat /proc/4868/sched
RT-process-demo (4868, #threads: 1)
-------------------------------------------------------------------
......
policy                                       :                    0
prio                                         :                  120
......
[lqm@localhost ~]$
```

如果以普通用户运行 RT-process-demo 则会报告不允许创建实时进程。

实时进程调度示例

下面用几个实时进程在单核系统上的竞争来展示实时调度的一些行为。我们借用代码 1-4 的方法，在某一时刻创建两个优先级为 90 的 RR 进程，延迟 5 秒后再创建一个优先级为 95 的 FIFO 进程，这三个进程完成相同的计算量。这里由 sched_setscheduler()指定的优先级请参见图 1-4，即 95 的优先级比 90 的高，而在 top 和 ps 命令中将显示为其他数值。

按照 Linux 调度，这三个进程的执行情况如图 1-5 所示。刚开始时两个实时 RR 进程按照轮循方式被调度运行，5 秒之后由于高优先级的 FIFO 实时进程将占有 CPU 的主要时间（两个 RR 进程将得不到调度运行），当该 FIFO 进程结束后又开始轮循地执行 RR 进程。

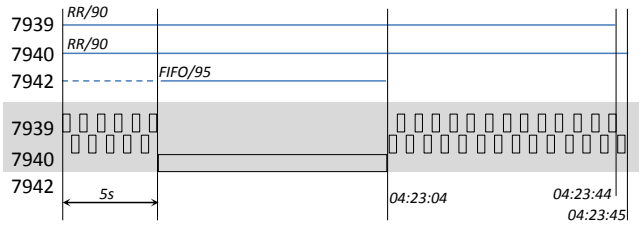

图 1-5 两个 RR/90 进程和一个 FIFO/95 进程在单核系统上调度示意图

用于启动三个进程的脚本如代码 1-5 所示，先用“RR-FIFO-sched 2 90&”创建两个优先级为 90 的 RR 实时进程在后台运行，然后 sleep 5s 等待 5s 后再用“RR-FIFO-sched 1 95&”创建优先级更高（95）的一个 FIFO 进程。RR-FIFO-sched 可执行文件由代码 1-6 产生。这里所示的优先级是用于 sched_setscheduler()的优先级，对应于内核优先级、top 命令、ps 命令、/proc/PID/sched 中的表示都不一样，可按图 1-4 进行转换。

代码 1-5 RR-FIFO.sh 脚本

```
[lqm@localhost lqm]# cat RR-FIFO.sh
RR-FIFO-sched  2 90&
RR-FIFO-sched  2 90&
sleep 5s
RR-FIFO-sched  1 95&
[lqm@localhost lqm]#
```

代码 1-6 根据命令行参数获得所需的 RR 或 FIFO 实时调度类型和优先级，然后由 sched_setscheduler()将自己设置为实时进程。该程序只能由 root 用户执行，其他用户无权设置实时进程。变成实时进程之后，该进程仅作简单的算术运行以持续一段时间便于观察。程序结束前将记录本进程的/proc/PID/sched 调度统计信息和结束时间记录在./sched-*pid* 文件中（文件名中的 *pid* 是指进程号）。

代码 1-6 RR-FIFO-sched.c

```
#include <sched.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>

int main(int argc,char ** argv)
{
   long i,j,temp;
   char cmd_str[100];
   int rc,current_scheduler_policy;
   struct sched_param my_params;    //将要设置的调度参数

   if(argc!=3)
   {    printf("usage:RR-FIFO-sched sched_class priority \nsched_class: 0 for CFS; 1 for
FIFO; 2 for RR\n");
        exit(1);
   }

   my_params.sched_priority=atoi(argv[2]);
   rc=sched_setscheduler(0,atoi(argv[1]),&my_params);
   if(rc<0)
```

```
        {
                perror("sched_setscheduler error\n");
                exit(0);
        }

    current_scheduler_policy=sched_getscheduler(0);
    printf("the PID:%d  current scheduler = %d \n", getpid(),current_scheduler_policy);

    for(i=0;i<1024*1024*1024;i++)
         for(j=0;j<10;j++)
              temp++;

    sprintf(cmd_str,"cat  /proc/%d/sched  > ./sched-%d  ;  date  >> ./sched-%d",  getpid(),
getpid(), getpid());
    system(cmd_str);                    //记录各个进程的/proc/PID/sched 以及时间信息

    return 0;
}
```

运行 RR-FIFO.sh 脚本，然后在其他终端上执行 top 命令，获得如图 1-6 所示的输出（sched_setscheduler()使用的优先级 90 和 95 在 top 命令显示的数值为-91 和-96，具体对应关系参见图 1-4）。可以看到刚开始有两个 RR 进程同时在运行各自拥有一半的 CPU 资源（48% 和 46.7%）。当高优先级的 FIFO 进程创建后，几乎所有 CPU 资源都被该进程所占用（95%，剩余 5%是防止系统无法响应终端命令而保留的），当 FIFO 进程结束后两个 RR 进程又平分 CPU 资源。在实时进程执行时间，普通进程只能分享保留的 5%CPU 资源，因此终端上的命令执行显得比较卡顿。

```
  PID USER      PR  NI    VIRT    RES    SHR S %CPU %MEM     TIME+ COMMAND
 7939 root     -91   0    4160    344    272 R 48.0  0.0   0:02.60 RR-FIF0-sched
 7940 root     -91   0    4160    344    272 R 46.7  0.0   0:01.40 RR-FIF0-sched
 3899 lqm       20   0 1570416 411024  50020 S  2.0 21.8   3:13.07 gnome-shell
 1207 root      20   0  252860  47752  10620 S  0.7  2.5   1:29.25 Xorg
 4562 lqm       20   0  578060  28348  15284 S  0.3  1.5   0:36.74 gnome-terminal

  PID USER      PR  NI    VIRT    RES    SHR S %CPU %MEM     TIME+ COMMAND
 7942 root     -96   0    4160    344    272 R 95.0  0.0   0:02.85 RR-FIF0-sched
 3899 lqm       20   0 1570416 411024  50020 S  1.7 21.8   3:13.18 gnome-shell
 1207 root      20   0  252860  47752  10620 S  0.3  2.5   1:29.28 Xorg
    1 root      20   0  128096   6716   3964 S  0.0  0.4   0:02.55 systemd
    2 root      20   0       0      0      0 S  0.0  0.0   0:00.01 kthreadd

  PID USER      PR  NI    VIRT    RES    SHR S %CPU %MEM     TIME+ COMMAND
 7940 root     -91   0    4160    344    272 R 11.3  0.0   0:03.19 RR-FIF0-sched
 7939 root     -91   0    4160    344    272 R 10.0  0.0   0:04.31 RR-FIF0-sched
 1073 root      20   0  553160  16432   5788 S  0.3  0.9   0:02.00 tuned
 1207 root      20   0  252860  47752  10620 S  0.3  2.5   1:29.37 Xorg
 3899 lqm       20   0 1570416 411032  50020 S  0.3 21.8   3:13.40 gnome-shell
```

图 1-6 top 命令查看高优先级实时进程 7942 抢占 CPU

等三个进程都结束后，我们来分析它们保留下来的 sched-*pid* 文件（即记录下来的/proc/PID/sched）中的信息。屏显 1-17 是运行 RR-FIFO.sh 的输出，以及用 cat 查看 sched-*pid* 文件的输出。我们在一个命令行上执行了三条 shell 语句，一次性将三个文件 sched-7939、sched-7940 和 sched-7942 的内容显示出来。下面请读者对照图 1-5 来解读这里的数据。首先是调度类型，7939 和 7940 的 policy 是 2 对应 RR、优先级 prio 为 9，7942 的 policy 是 1 对应 FIFO、优先级为 4。此处/proc/PID/sched 使用的优先级数值与 sched_setscheduler()、top 命令使用的优先级对应关系请参考图 1-4。由于三个进程都是完成算术运算，因此只有一次自愿调度切换其他都是被动的调度切换。而 RR 是轮循因此有多达 248 和 258 次被动调度切换，而 FIFO 只

被系统打断而发生 31 次被动调度切换。由于 4942 进程比另两个进程优先级高，即使较后开始执行也可以抢到大部分 CPU 资源从而较早结束（04:23:04），而另两个 RR 进程则分别在后面的 04:23:44 和 04:23:45 时间结束。

屏显 1-17 运行脚本并查看三个实时进程的调度信息

```
[root@localhost lqm]# RR-FIFO.sh
the PID:7939  current scheduler = 2
the PID:7940  current scheduler = 2
the PID:7942  current scheduler = 1
[root@localhost lqm]cat ./sched-7939; cat ./sched-7940; cat ./sched-7942
RR-FIFO-sched (7939, #threads: 1)
----------------------------------------------------------------------
se.exec_start                                :      12738161.797655
se.vruntime                                  :             3.069446
se.sum_exec_runtime                          :         23053.874997
se.nr_migrations                             :                    0
nr_switches                                  :                  249
nr_voluntary_switches                        :                    1
nr_involuntary_switches                      :                  248
se.load.weight                               :                 1024
policy                                       :                    2
prio                                         :                    9
clock-delta                                  :                   30
mm->numa_scan_seq                            :                    0
numa_migrations, 0
numa_faults_memory, 0, 0, 1, 0, -1
numa_faults_memory, 1, 0, 0, 0, -1
2017 年 05 月 04 日 星期四 04:23:44 EDT
RR-FIFO-sched (7940, #threads: 1)
----------------------------------------------------------------------
se.exec_start                                :      12739296.986898
se.vruntime                                  :             1.609366
se.sum_exec_runtime                          :         23032.693034
se.nr_migrations                             :                    0
nr_switches                                  :                  259
nr_voluntary_switches                        :                    1
nr_involuntary_switches                      :                  258
se.load.weight                               :                 1024
policy                                       :                    2
prio                                         :                    9
clock-delta                                  :                   33
mm->numa_scan_seq                            :                    0
numa_migrations, 0
numa_faults_memory, 0, 0, 1, 0, -1
numa_faults_memory, 1, 0, 0, 0, -1
2017 年 05 月 04 日 星期四 04:23:45 EDT
RR-FIFO-sched (7942, #threads: 1)
----------------------------------------------------------------------
se.exec_start                                :      12698028.010862
se.vruntime                                  :             0.250183
se.sum_exec_runtime                          :         23110.223056
se.nr_migrations                             :                    0
nr_switches                                  :                   32
nr_voluntary_switches                        :                    1
nr_involuntary_switches                      :                   31
se.load.weight                               :                 1024
policy                                       :                    1
prio                                         :                    4
clock-delta                                  :                   27
mm->numa_scan_seq                            :                    0
numa_migrations, 0
numa_faults_memory, 0, 0, 1, 0, -1
numa_faults_memory, 1, 0, 0, 0, -1
```

```
2017 年 05 月 04 日 星期四 04:23:04 EDT
[root@localhost lqm]#
```

实验中可以看到实时进程消耗了系统绝大部分 CPU 资源，因此只在必要的情况下才谨慎使用。FIFO 由于避免了调度切换的时间开销，因此更有利于本任务的快速完成。但是同时由于 FIFO 也会造成同级 FIFO 任务的明显延迟，而 RR 进程则没有这样的问题。但是无论如何，Linux 实时进程只是优先级上高于普通进程，在调度的切换延迟方面和普通进程是一样，因此 Linux 实时进程并无法满足响应时间非常严格的硬实时环境。另外，通常实时任务都是间断性、周期性的短期任务，而不是代码 1-6 那样的长时间循环。

1.4. 进程迁移与负载均衡

下面来讨论多核相关的一些调度迁移和负载均衡的问题。如果读者前面一直使用的是使单核的虚拟机，那么需要将虚拟机重新配置变为多核的——本节将使用 4 核虚拟机作为例子。具体设置访问参见图 1-1 及相关说明。Linux 主要在以下几个时机进行负载均衡：周期性地触发负载均衡、当处理器进入 IDLE 状态时触发负载均衡、在唤醒/fork/exec 等操作时进行负载均衡。由于负载均衡算法比较复杂，我们仅观察负载均衡的大致行为，而不涉及各种负载、权重以及判决条件等等更细节的内容。

1.4.1. CFS 进程的负载均衡

下面通过给系统施加不同负载权重的进程并观察系统负载情况和进程迁移现象来观察 Linux 多核系统上的进程负载均衡过程。

负载程序

为了方便观察 CPU 之间的负载均衡过程，我们先写了若干个不同负载压力的程序，分别叫做 HelloWorld-loop-nice-15、HelloWorld-loop-nice-10、HelloWorld-loop-nice-5、 HelloWorld-loop-nice-0、HelloWorld-loop-nice+5、HelloWorld-loop-nice+10 和 HelloWorld-loop-nice+15。例如 HelloWorld-loop-nice+5 可执行文件对应的源代码 HelloWorld-loop-nice+5.c 如所示。其余代码的差异在于传入 nice()的参数不同，因此会有不同的优先级（对应不同负载权重）。

代码 1-7 HelloWorld-loop-nice+5.c

```
#include <stdio.h>
int main()
{
     int k;
     printf("HelloWorld!¥n");
     nice(5);
     while(1);
          k=k+1;
     return 0;
}
```

我们尝试在后台运行 HelloWorld-loop-nice+5 和 HelloWorld-loop-nice+15，如屏显 1-18 所示。然后通过/proc/PID/sched 查看它们的负载权重（与优先级直接相关），HelloWorld-loop-nice+5 进程号为 4128，其 NICE=5 权重为 335。而 HelloWorld-loop-nice+15 进程号为 4136，其

NICE=15 优先级较低，因此权重较低为 36。注意-5/-10/-15 的进程都必须用 root 用户执行，否则无法正确设置 NICE 值和优先级。按照优先级-15/-10/-5/0/+5/+10/+15 从高到低排列，对应的负载权重为：29154/9548/3121/1024/335/110/36，相邻两级大致相差 3 倍左右。

屏显 1-18 执行 HelloWorld-loop-nice+5/+15 并查看它们的负载权重.

```
[lqm@localhost ~]$ HelloWorld-loop-nice+5&
[1] 4128
HelloWorld!
[lqm@localhost ~]$
[lqm@localhost ~]$ HelloWorld-loop-nice+15&
[2] 4136
HelloWorld!
[lqm@localhost ~]$
[lqm@localhost ~]$ cat /proc/4128/sched |grep se.load.weight
se.load.weight                               :                335
[lqm@localhost ~]$ cat /proc/4136/sched |grep se.load.weight
se.load.weight                               :                 36
[lqm@localhost ~]$
```

为了查看各个进程在哪个 CPU 上运行，可以用 top 命令。不过默认的 top 命令并不显示进程所在的处理器，因此在启动 top 之后要按 f 进入管理界面（如屏显 1-19 所示），选择显示 P（Last Used Cpu）列（此时前面出现“*”号）。也可以用 ps -eo pid,args,psr 来显示进程号、命令行参数和所在 CPU 号。

屏显 1-19 top 命令中选择所需要显示的列

```
Fields Management for window 1:Def, whose current sort field is %CPU
   Navigate with Up/Dn, Right selects for move then <Enter> or Left commits,
   'd' or <Space> toggles display, 's' sets sort.  Use 'q' or <Esc> to end!

* PID     = Process Id      PGRP    = Process Group   vMj     = Major Faults
* USER    = Effective Use   TTY     = Controlling T   vMn     = Minor Faults
* PR      = Priority        TPGID   = Tty Process G   USED    = Res+Swap Size
* NI      = Nice Value      SID     = Session Id      nsIPC   = IPC namespace
* VIRT    = Virtual Image   nTH     = Number of Thr   nsMNT   = MNT namespace
* RES     = Resident Size * P       = Last Used Cpu   nsNET   = NET namespace
* SHR     = Shared Memory   TIME    = CPU Time        nsPID   = PID namespace
* S       = Process Statu   SWAP    = Swapped Size    nsUSER  = USER namespac
* %CPU    = CPU Usage       CODE    = Code Size (Ki   nsUTS   = UTS namespace
* %MEM    = Memory Usage    DATA    = Data+Stack (K
* TIME+   = CPU Time, hun   nMaj    = Major Page Fa
* COMMAND = Command Name/   nMin    = Minor Page Fa
  PPID    = Parent Proces   nDRT    = Dirty Pages C
  UID     = Effective Use   WCHAN   = Sleeping in F
  RUID    = Real User Id    Flags   = Task Flags <s
  RUSER   = Real User Nam   CGROUPS = Control Group
  SUID    = Saved User Id   SUPGIDS = Supp Groups I
  SUSER   = Saved User Na   SUPGRPS = Supp Groups N
  GID     = Group Id        TGID    = Thread Group
  GROUP   = Group Name      ENVIRON = Environment v
```

按 ESC 键从管理界面退出后，top 的输出就多了一个“P”列，用数字 0~3 表示四个逻辑 CPU，如屏显 1-20 所示。因为我们暂时不关心内存使用，因此 VIRT/RES/SHR/%MEM 四列信息可以去掉。有了这样的 top 输出，就很方便确定进程所在的处理器号，从而能粗略观察负载均衡的行为。

屏显 1-20 显示进程所在处理器编号的 top 输出

```
top - 09:20:56 up  1:01,  4 users,  load average: 1.52, 1.43, 0.89
```

```
Tasks: 196 total,   3 running, 193 sleeping,   0 stopped,   0 zombie
%Cpu(s): 26.4 us,  0.1 sy, 24.6 ni, 48.9 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
KiB Mem :  1883408 total,   869816 free,   513464 used,   500128 buff/cache
KiB Swap:  1679356 total,  1679356 free,        0 used.  1171200 avail Mem

  PID USER      PR  NI    VIRT    RES    SHR S  %CPU %MEM     TIME+ COMMAND    P
 4780 root      10 -10    4160    344    272 R 100.0  0.0   9:38.59 HelloWor+ 2
 4851 root      30  10    4160    344    272 R  98.0  0.0   6:52.76 HelloWor+ 3
 3098 lqm       20   0 1773496 195440  49224 S   4.3 10.4   0:32.11 gnome-sh+ 3
 1410 root      20   0  271436  36464  10416 S   0.7  1.9   0:12.10 Xorg      1
 3800 lqm       20   0  575724  25528  14820 S   0.7  1.4   0:05.69 gnome-te+ 1
 2829 lqm       20   0   35864   2176    944 S   0.3  0.1   0:00.38 dbus-dae+ 3
    1 root      20   0  193632   6728   3964 S   0.0  0.4   0:03.12 systemd   3
    2 root      20   0       0      0      0 S   0.0  0.0   0:00.00 kthreadd  1
    3 root      20   0       0      0      0 S   0.0  0.0   0:00.04 ksoftirq+ 0
    5 root       0 -20       0      0      0 S   0.0  0.0   0:00.00 kworker/+ 0
    6 root      20   0       0      0      0 S   0.0  0.0   0:00.13 kworker/+ 2
    7 root      rt   0       0      0      0 S   0.0  0.0   0:00.00 migratio+ 0
    8 root      20   0       0      0      0 S   0.0  0.0   0:00.00 rcu_bh    0
    9 root      20   0       0      0      0 S   0.0  0.0   0:01.09 rcu_sched 3
   10 root      rt   0       0      0      0 S   0.0  0.0   0:00.04 watchdog+ 0
   11 root      rt   0       0      0      0 S   0.0  0.0   0:00.03 watchdog+ 1
   12 root      rt   0       0      0      0 S   0.0  0.0   0:00.00 migratio+ 1
```

负载均衡观察

为了观察负载均衡过程，很重要的一点是要记录系统中各 CPU 的负载情况，以及新建进程或被撤销进程的负载权重。前面我们在屏显 1-4 中已经看到双核系统上 CPU#0 负载为 3072 而 CPU#1 的负载 load 为 1024。下面的实验，我们将虚拟机设置为 4 核（具体设置方法在本章开头处已经讨论过），然后记录下初始的背景负载情况。由于系统基本上是空闲的，因此大多数处理器的负载为 0，偶尔有一些进程活动才不为 0。屏显 1-21 是在我们 4 核虚拟机上用对 /proc/sched_debug 筛选出带有“.load ”（注意后面有一个空格）行的结果。每个处理器两行，第一行是处理器负载总量，第二行是该处理器上 CFS 进程的负载总量。由于系统不定期有进程活动，因此负载并不全为 0 并且会有一些波动（例如 CPU3 的负载不为 0 而是短暂地为 2048）。

屏显 1-21 空闲系统上的即时负载情况

```
[lqm@localhost ~]$ cat /proc/sched_debug |grep ".load "
  .load                          : 0
  .load                          : 0
  .load                          : 0
  .load                          : 0
  .load                          : 0
  .load                          : 0
  .load                          : 2048
  .load                          : 2048
[lqm@localhost ~]$
```

我们先启动 NICE=-5 四个进程，此时的 top 输出如屏显 1-22 所示。四个进程分别在 0/1/2/3 号 CPU 处理器上运行，使负载分散开来。由于大家优先级相同、权重相同，因此负载是平衡的，也没有必要将进程进行迁移调整。

屏显 1-22 用 top 查看四个 NICE=-5 的进程所在 CPU

```
top - 03:02:55 up 18:43,  4 users,  load average: 3.86, 2.65, 1.25
Tasks: 198 total,   7 running, 191 sleeping,   0 stopped,   0 zombie
%Cpu(s):100.0 us,  0.0 sy,  0.0 ni,  0.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
KiB Mem :  1883408 total,  1001880 free,   374236 used,   507292 buff/cache
KiB Swap:  1679356 total,  1679356 free,        0 used.  1310840 avail Mem
```

```
  PID USER      PR  NI S  %CPU     TIME+ COMMAND                           P
17324 root      15  -5 R 100.0   1:34.83 HelloWorld-loop                   2
17328 root      15  -5 R 100.0   1:29.92 HelloWorld-loop                   0
17332 root      15  -5 R 100.0   1:28.02 HelloWorld-loop                   3
17336 root      15  -5 R 100.0   1:26.24 HelloWorld-loop                   1
 1410 root      20   0 S   0.3   1:11.63 Xorg                             3
    1 root      20   0 S   0.0   0:05.00 systemd                          1
    2 root      20   0 S   0.0   0:00.03 kthreadd                         2
    3 root      20   0 S   0.0   0:00.10 ksoftirqd/0                      0
    5 root       0 -20 S   0.0   0:00.00 kworker/0:0H                     0
    6 root      20   0 S   0.0   0:00.28 kworker/u8:0                     1
```

此时用“cat /proc/sched_debug |grep load”命令检查各处理器负载，如屏显 1-23 所示。这里比屏显 1-22 多显示了一些信息，cpu_load[x]是对即时负载平滑后的数值，其中 cpu_load[4]是最为平滑的数值（能较好地消除即时负载抖动的影响）。

屏显 1-23 察看运行-5/0/+5/+10 四个进程后各处理器负载

```
[root@localhost lqm]# cat /proc/sched_debug |grep load
  .load                          : 3121
  .nr_load_updates               : 49470610
  .cpu_load[0]                   : 3121
  .cpu_load[1]                   : 3629
  .cpu_load[2]                   : 3807
  .cpu_load[3]                   : 3869
  .cpu_load[4]                   : 3958
  .load                          : 3121
  .runnable_load_avg             : 3120
  .blocked_load_avg              : 639
  .tg_load_avg                   : 0
  .tg_load_contrib               : 0
  .load                          : 4145
  .nr_load_updates               : 49603768
  .cpu_load[0]                   : 4145
  .cpu_load[1]                   : 3645
  .cpu_load[2]                   : 3462
  .cpu_load[3]                   : 3466
  .cpu_load[4]                   : 3605
  .load                          : 4145
  .runnable_load_avg             : 3245
  .blocked_load_avg              : 24
  .tg_load_avg                   : 0
  .tg_load_contrib               : 0
  .load                          : 5169
  .nr_load_updates               : 50249068
  .cpu_load[0]                   : 5169
  .cpu_load[1]                   : 5169
  .cpu_load[2]                   : 5166
  .cpu_load[3]                   : 5088
  .cpu_load[4]                   : 4850
  .load                          : 5169
  .runnable_load_avg             : 4901
  .blocked_load_avg              : 0
  .tg_load_avg                   : 0
  .tg_load_contrib               : 0
  .load                          : 4145
  .nr_load_updates               : 49034668
  .cpu_load[0]                   : 4145
  .cpu_load[1]                   : 3665
  .cpu_load[2]                   : 3458
  .cpu_load[3]                   : 3356
  .cpu_load[4]                   : 3462
  .load                          : 4145
  .runnable_load_avg             : 3542
  .blocked_load_avg              : 208
```

```
  .tg_load_avg                    : 0
  .tg_load_contrib                : 0
[root@localhost lqm]#
```

由于此时各处理器的负载几乎相等，因此基本不需要负载均衡——这四个-5 进程不需要在处理器之间迁移，因此从/proc/PID/sched 查看 se.nr_migrations 看迁移次数是接近于 0 的，如屏显 1-24 所示。

屏显 1-24 从/proc/PID/sched 中查看进程的迁移次数

```
[root@localhost lqm]# cat /proc/17324/sched |grep "se.nr_migrations"
se.nr_migrations                            :                    2
[root@localhost lqm]# cat /proc/17328/sched |grep "se.nr_migrations"
se.nr_migrations                            :                    3
[root@localhost lqm]# cat /proc/17332/sched |grep "se.nr_migrations"
se.nr_migrations                            :                    2
[root@localhost lqm]# cat /proc/17336/sched |grep "se.nr_migrations"
se.nr_migrations                            :                    0
[root@localhost lqm]#
```

但是如果此时加入一个新进程，例如加入 NICE=-5 的进程，在四个进程中无法稳定的保持一个均衡状态，必须不断调整才能让各处理器负载相对平衡。先从 top 命令上就看出，进程所在的处理器编号总不断变化，我们截取了三次不同时刻的 top 输出来观察四个进程所在处理器，如屏显 1-25 所示。可以看出进程 17552 在这三个时刻分别在处理 0、2 和 1。

屏显 1-25 用 top 查看五个进程的处理器编号

```
...
  PID USER      PR  NI S  %CPU     TIME+ COMMAND                                P
17336 root      15  -5 R  96.7  16:17.99 HelloWorld-loop                         1
17324 root      15  -5 R  96.3  16:26.48 HelloWorld-loop                         2
17328 root      15  -5 R  96.0  16:21.49 HelloWorld-loop                         0
17332 root      15  -5 R  85.4  16:18.96 HelloWorld-loop                         3
17552 root      20   0 R  24.6   0:04.54 HelloWorld-loop                         0
...
  PID USER      PR  NI S  %CPU     TIME+ COMMAND                                P
17336 root      15  -5 R  96.3  17:03.50 HelloWorld-loop                         1
17332 root      15  -5 R  95.7  17:03.37 HelloWorld-loop                         0
17328 root      15  -5 R  94.4  17:07.85 HelloWorld-loop                         3
17324 root      15  -5 R  89.0  17:10.98 HelloWorld-loop                         2
17552 root      20   0 R  23.9   0:16.23 HelloWorld-loop                         2
...
  PID USER      PR  NI S  %CPU     TIME+ COMMAND                                P
17328 root      15  -5 R  99.0  18:39.96 HelloWorld-loop                         3
17324 root      15  -5 R  95.7  18:44.02 HelloWorld-loop                         2
17332 root      15  -5 R  95.0  18:36.04 HelloWorld-loop                         0
17336 root      15  -5 R  86.7  18:36.60 HelloWorld-loop                         1
17552 root      20   0 R  23.6   0:41.88 HelloWorld-loop                         1
...
```

然后我们进一步通过/proc/PID/sched 查看这些进程的迁移次数，只关注 se.nr_migrations 统计，如屏显 1-26 所示。可以看出负载较轻的进程 17552，被频繁地在各个处理器间迁移。可以想象，如果迁移 NICE=-5 进程会引起更大的负载不均衡，这就是为什么 NICE=0 的 17552 进程被迁移了 1758 次而其他进程才被迁移了 51/63/109 次的原因。通过这些迁移和调度，才能保证各个进程能按照 NICE 或 priority 的比例分配到合适的时间。如果检查/proc/PID/sched 的虚拟时间 se.vruntime，则发现它们是基本同步的（因为都是就绪可运行而不是睡眠状态）；但

是检查它们在 CPU 上的执行时间 se.sum_exec_runtime，就能发现 NICE=-5 的四个进程时间相近，而 NICE=0 的进程所获 CPU 时间要小得多。

屏显 1-26 用/proc/PID/sched 查看进程迁移次数

```
 [root@localhost lqm]# cat /proc/17324/sched /proc/17328/sched /proc/17332/sched /proc/17336/sched
/proc/17552/sched |grep "se.nr_migrations"
se.nr_migrations                            :                 51
se.nr_migrations                            :                 63
se.nr_migrations                            :                109
se.nr_migrations                            :                 46
se.nr_migrations                            :               1758
[root@localhost lqm]#
```

1.4.2. 实时进程的负载均衡

实时任务有自己独立的负载均衡，并且由推（push）和拉（pull）操作实现任务在运行队列之间迁移。实时任务的均衡原则很简单，对于 N 个处理器的系统，保证优先级最高的 N 个实时任务各自在一个处理器上运行。如果实时任务少于 N 个，则多出的处理器可以运行普通 CFS 任务。

实时进程负载均衡

根据前面提到的原则（我们将优先级最高的 N 个进程简称为 N-集合），实时进程的负载均衡发生在以下场合中。

1） 任务唤醒时，如果它的优先级足够高可以进入到 N-集合，那么它将抢夺原来 N-集合中最低优先级任务的处理器（否则就插入到 rt 队列等候调度）。这又分两种情况——在本地优先级最高、在本地优先级次高。

第一种是新唤醒的任务在本处理器运行队列中优先级最高，那么它将抢占本地处理器，具体如图 1-7 所示。被抢占处理器的任务有两种处理方法，如果它的优先级不足以进入 N-集合，那么它就在本地处理器队列中排队（图 1-7-a 所示），否则它仍在 N-集合那么它将通过 post_schedule_rt()的 push 操作推送到“退出 N-集合的任务”所在的处理器上（图 1-7-b 所示）。

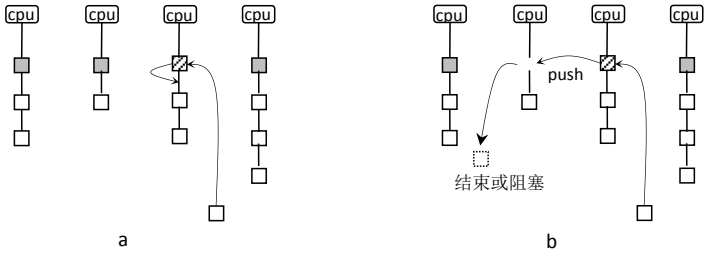

图 1-7 唤醒进程抢占本地处理器

第二种情况是新唤醒的进程虽然在 N-集合中，但是优先级在本地处理器上只能排第二，即便唤醒也不能在本地上运行。那么它能通过 push 操作到优先级最低的处理器上而获得运行，如图 1-8 所示。

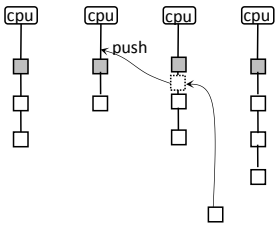

图 1-8 唤醒进程未能抢占本地处理器

2） 当一个正在运行的实时任务结束或阻塞时，调度程序一般将选取本队列中最高优先级（原来优先级第二高）的进程。但是它如果不属于 N-集合，那么将会通过 pull 操作将其他队列中“第二高”优先级的实时任务（属于 N-集合）拉过来，如图 1-9 所示。

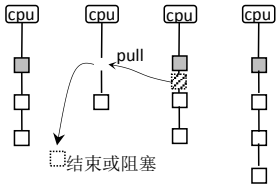

图 1-9 本地任务不是 N-集合时触发 pull 操作

也就是说 Linux 内核对实时任务优先级的排队是全系统范围进行的，也就是说调度与负载均衡同时进行。对于 N 个处理器的系统总是执行优先级排序最高的 N 各实时任务。

实时负载均衡观察

下面先创建 4 个实时进程，我们用前面的 RR-FIFO-sched 程序，用代码 1-8 的脚本来创建 FIFO 实时进程。使用脚本的原因和前面一样，主要考虑到实时进程运行后系统响应较慢，键盘敲命令不方便。

代码 1-8 RT-Balance1.sh

```
root@localhost lqm]# cat RT-Balance1.sh
RR-FIFO-sched 1 10&
RR-FIFO-sched 1 20&
RR-FIFO-sched 1 30&
RR-FIFO-sched 1 40&
```

运行后如预期一样，各个进程被分配到不同的处理器上运行，用 top 命令观察如屏显 1-27 所示。

屏显 1-27 实时进程在四个处理器上运行

```
…
  PID USER      PR  NI S  %CPU     TIME+ COMMAND                                P
18263 root     -11   0 R  95.3   0:18.00 RR-FIFO-sched                          0
18262 root     -21   0 R  95.0   0:17.99 RR-FIFO-sched                          1
18264 root     -31   0 R  95.0   0:17.96 RR-FIFO-sched                          2
18265 root     -41   0 R  95.0   0:17.99 RR-FIFO-sched                          3
…
```

如果在上述四个进程执行期间，用命令“RR-FIFO-sched 2 85”创建一个优先级更高的进程，则它会抢占一个 CPU 并将一个由 “RR-FIFO-sched 1 10”命令创建的进程阻塞。虽然此时有 5 个 FIFO 实时进程，但是只有高优先级的 4 个在运行。当然，这里演示使用的实时进程并不能代表真实应用，仅作负载均衡的演示目的。

屏显 1-28 高优先级实时进程抢占低优先级实时进程的示例

```
...
  PID USER      PR  NI S  %CPU     TIME+ COMMAND                                P
19442 root     -21   0 R  94.7   0:16.99 RR-FIFO-sched                          3
19443 root     -31   0 R  94.7   0:16.98 RR-FIFO-sched                          0
19444 root     -41   0 R  94.7   0:16.98 RR-FIFO-sched                          2
19446 root     -51   0 R  94.7   0:10.45 RR-FIFO-sched                          1
 3800 lqm       20   0 S   0.3   1:14.63 gnome-terminal-                       2
...
```

其他更复杂的负载均衡现象，请读者自行观察。例如有相同优先级的 RR 进程参与竞争等。

1.5. 小结

本章主要讨论进程调度和负载均衡，同时观察了单核系统和多核系统。首先给出了调度框架，指出进程调度类在各个处理器上单独组织成队列。然后讨论了进程状态以及基本的就绪运行和阻塞状态的转换。最后是观察普通进程 CFS 和实时进程 RT 的调度和负载均衡，并用示例加深对调度的认知。由于调度是内核主导的过程，应用程序仅能在调度类型和优先级上进行控制，因此我们主要通过/proc 文件系统给出的调度统计信息来体验和观察调度行为。

1.6. 练习

1. 在一个空闲的单核 Linux 系统上用 nice 命令调整两个进程的优先级，使得它们各自使用 1/5 和 4/5 的 CPU 资源。
2. 在一个空闲的双核 Linux 系统上启动 P1/P2/P3/P4 四个一直就绪不阻塞进程，使得 P1/P3 在第一个处理器上运行各占 50%的 CPU 资源，P2/P4 在另一个处理器上运行，各自 30%和 70%的 CPU 资源。
3. 请预测 Linux 对于相同优先级的 FIFO 实时进程的调度情况，并通过实验给出实测数据以证明。
4. 在一个空载的系统上，如果将代码 1-8 的最后一行替换成两个“RR-FIFO-sched 2 80”，请预测其负载均衡后进程的运行情况，并说明是否只有一种处理器分配方案?