第3章 进程

计算机系统中CPU个数少，多个程序在执行时都想占有它并独自在上面运行，但CPU本身并没有分身术，因此互不相让的程序之间可能会厮打起来。作为管理者的操作系统，决不能袖手旁观。于是，操作系统的设计者发明了进程这一概念。

3.1 进程介绍

现在所有计算机都能同时做几件事情。例如，用户一边运行浏览器程序上网浏览信息，一边运行字处理程序编辑文档。在一个多程序系统中，CPU由这道程序向那道程序切换，使每道程序运行几十毫秒或几百毫秒。然而严格地说，在一个瞬间，CPU只能运行一道程序。但在一段时期内，它却可能轮流运行多个程序，这样就给用户一种并行的错觉。有时人们称其为伪并行——就是指CPU在多道程序之间快速地切换，以此来区分它与多处理机（两个或更多的CPU共享物理存储器）系统是真正的硬件在并行。由于人们很难对多个并行的活动进行跟踪。因此，经过多年的探索，操作系统的设计者抽象出了进程这样一个逻辑概念，使得并行更容易被理解和处理。

3.1.1 程序和进程

程序是一个普通文件，是机器代码指令和数据的集合，这些指令和数据存储在磁盘上的一个可执行映像（Executable Image）中。所谓可执行映像就是一个可执行文件的内容，例如，我们编写了一个C语言源程序，最终这个源程序要经过编译、连接成为一个可执行文件后才能运行。源程序中要定义许多变量，在可执行文件中，这些变量就组成了数据段的一部分；源程序中的许多语句，例如“i++；for（i=0;i<10;i++）”等，在可执行文件中，它们对应着许多不同的机器代码指令，这些机器代码指令经CPU执行后，就完成了我们所期望的工作。可以这么说，程序代表我们期望完成某工作的计划和步骤，它还浮在纸面上，等待具体实现。而具体的实现过程就是由进程来完成的，可以认为进程是运行中的程序，它除了包含程序中的所有内容外，还包含一些额外的数据。

我们知道，程序装入内存后才得以运行。在程序计数器的控制下，指令被不断地从内存取至CPU中运行。实际上，程序的执行过程可以说是一个执行环境的总和，这个执行环境除了包括程序中各种指令和数据外，还有一些额外数据，比如寄存器的值、用来保存临时数据（例如传递给某个函数的参数、函数的返回地址、保存的临时变量等）的堆栈、被打开的文

件及输入/输出设备的状态等。上述执行环境的动态变化表征了程序的运行。为了对这个动态变化的过程进行描述,程序这个概念已经远远不够,于是就引入了“进程”的概念。进程代表程序的执行过程,它是一个动态的实体,随着程序中指令的执行而不断地变化。在某个时刻进程的内容被称为进程映像(Process Image)。

Linux是多任务操作系统,也就是说可以有多个程序同时装入内存并运行,操作系统为每个程序建立一个运行环境即创建进程。从逻辑上说,每个进程都拥有它自己的虚拟CPU。当然,实际上真正的CPU在各进程之间来回切换。但如果为了想研究这种系统而去跟踪CPU如何在程序间来回切换将会是一件相当复杂的事情。于是换个角度,集中考虑在(伪)并行情况下运行的进程集就使问题变得简单、清晰得多。这种快速的切换称作多道程序执行。在一些UNIX书籍中,又把“进程切换”(Process Switching)称为“环境切换”或“上下文切换”(Context Switching)。这里“进程的上下文”就是指进程的执行环境。

进程运行过程中,还需要其他一些系统资源,例如,要用CPU来运行它的指令、要用系统的物理内存来容纳进程本身以及和它有关的数据、要在文件系统中打开和使用文件、并且可以直接或间接的使用系统的物理设备,例如打印机、扫描仪等。由于这些系统资源是由所有进程共享的,所以操作系统必须监视进程和它所拥有的系统资源,使它们可以公平地拥有系统资源以得到运行。

由此,本文给出进程的明确定义:所谓进程是由正文段(Text)、用户数据段(User Segment)以及系统数据段(System Segment)共同组成的一个执行环境,如图3.1所示。

(1) 正文段:存放被执行的机器指令。这个段是只读的,它允许系统中正在运行的两个或多个进程之间能够共享这一代码。例如,有几个用户都在使用文本编辑器,在内存中仅需要该程序指令的一个副本,他们全都共享这一副本。

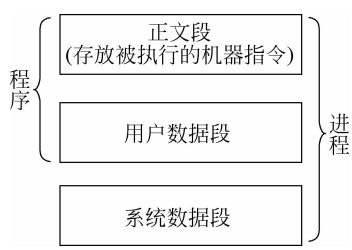

图3.1 进程的组成

(2) 用户数据段:存放进程在执行时直接进行操作的所有数据,包括进程使用的全部变量在内。显然,这里包含的信息可以被改变。虽然进程之间可以共享正文段,但是每个进程需要有它自己的专用用户数据段。例如同时编辑文本的用户,虽然运行着同样的程序——编辑器,但是每个用户都有不同的数据,如正在编辑的文本。

(3) 系统数据段:该段有效地存放程序运行的环境。事实上,这正是程序和进程的区别所在。如前所述,程序是由一组指令和数据组成的静态事物,它们是进程最初使用的正文段和用户数据段。作为动态事物,进程是正文段、用户数据段和系统数据段的信息的交叉综合体,其中系统数据段是进程实体最重要的一部分。之所以说它有效地存放程序运行的环境,是因为这一部分存放有进程的控制信息。系统中有许多进程,操作系统要管理它们、调度它们运行,就是通过这些控制信息。Linux为每个进程建立了task_struct数据结构来容纳这些控制信息。

假设有三道程序A,B,C在系统中运行。程序一旦运行起来,就称它为进程,因此称它们为三个进程Pa,Pb,Pc。假定进程Pa执行到一条输入语句,因为这时要从外设读入数据,于是进程Pa主动放弃CPU。此时操作系统中的调度程序就要选择一个进程投入运行,假设选中Pc,就会发生进程切换,从Pa切换到Pc。同理,在某个时刻可能切换到进程Pb。从

某一时间段看，三个进程在同时执行，从某一时刻看，只有一个进程在运行，我们把这几个进程的伪并行执行叫做进程的并发执行。

在 Linux 系统中还可以使用 ps 命令来查看当前系统中的进程和进程的一些相关信息。使用这个命令可以查看系统中所有进程的状态。该命令可以确定有哪些进程正在运行和运行的状态、进程是否结束、进程有没有僵死、哪些进程占用了过多的资源等。例如：ps -e

```
$ ps -e
  PID   TTY          TIME         CMD
    1   ?            00:00:00     init
    2   ?            00:00:00     kthreadd
 2102   ?            00:04:04     firefox-bin
 2206   pts/0        00:00:00     bash
 2211   pts/0        00:00:05     fcitx
 2809   ?            00:00:01     stardict
 3317   ?            00:00:05     qq
 ⋮
```

这里只是截取了部分进程和部分信息，即进程的进程号、进程相关的终端(? 表示进程不需要终端)、进程已经占用 CPU 的时间和启动进程的程序名。在后面的学习中还要使用这个命令来查看进程的其他信息。

3.1.2 进程的层次结构

进程是一个动态的实体，它具有生命周期，系统中进程的生死随时发生。因此，对操作系统中进程的描述模仿人类的活动。一个进程不会平白无故的诞生，它总会有自己的父母。在 Linux 中，通过调用 fork 系统调用来创建一个新的进程。新创建的子进程同样也能执行 fork，所以，有可能形成一棵完整的进程树。注意，每个进程只有一个父进程，但可以有 0 个、1 个、2 个或多个子进程。

从身边的例子体验进程树的诞生，比如 Linux 的启动。Linux 在启动时就创建一个称为 init 的特殊进程。顾名思义，它是起始进程，是祖先，以后诞生的所有进程都是它的后代——或是它的儿子，或是它的孙子。init 进程为每个终端(TTY)创建一个新的管理进程，这些进程在终端上等待着用户的登录。当用户正确登录后，系统再为每一个用户启动一个 shell 进程，由 shell 进程等待并接收用户输入的命令信息，图 3.2 是一棵进程树。

此外，init 进程还负责管理系统中的“孤儿”进程。如果某个进程创建子进程之后就终止，而子进程还“活着”，则子进程成为孤儿进程。init 进程负责“收养”该进程，即孤儿进程会立即成为 init 进程的儿子，也就说，init 进程承担着养父的角色。这是为了保持进程树的完整性。

在 Linux 系统中可以使用 pstree 命令来查看系统中的树状结构；pstree 将所有进程显示为树状结构，以清楚地表达程序间的相互关系。从该命令的显示结果可以看到，init 进程是系统中唯一一个没有父进程的进程，它是系统中的第一个进程，其他进程都是由它和它的子进程产生的。

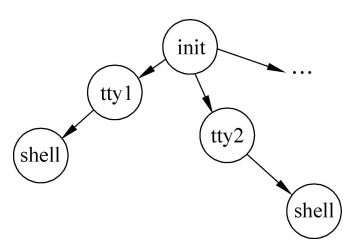

图 3.2 进程树

另外 ps 命令也可以显示进程的树状结构，例如：

```
$ps -elH
```

3.1.3 进程状态

为了对进程从产生到消亡的这个动态变化过程进行捕获和描述,就需要定义进程各种状态并制定相应的状态转换策略,以此来控制进程的运行。

因为不同的操作系统对进程的管理方式和对进程的状态解释可以不同,所以不同的操作系统中描述进程状态的数量和命名也会有所不同,但最基本的进程状态有以下三种:

(1) 运行态:进程占有 CPU,并在 CPU 上运行。

(2) 就绪态:进程已经具备运行条件,但由于 CPU 忙而暂时不能运行。

(3) 阻塞态(或等待态):进程因等待某种事件的发生而暂时不能运行(即使 CPU 空闲,进程也不可运行)。

进程在生命周期内处于且仅处于三种基本状态之一,如图 3.3 所示。

这三种状态之间有以下 4 种可能的转换关系。

(1) 运行态→阻塞态:进程发现它不能运行下去时发生这种转换。这是因为进程发生 I/O 请求或等待某件事情。

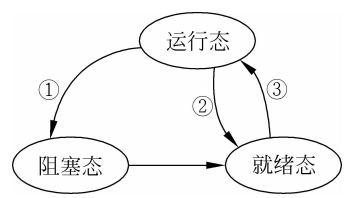

图 3.3 进程的状态及其转换

(2) 运行态→就绪态:在系统认为运行进程占用 CPU 的时间已经过长,决定让其他进程占用 CPU 时发生这种转换。这是由调度程序引起的。调度程序是操作系统的一部分,进程甚至感觉不到它的存在。

(3) 就绪态→运行态:运行进程已经用完分给它的 CPU 时间,调度程序从处于就绪态的进程中选择一个投入运行。

(4) 阻塞态→就绪态:当一个进程等待的一个外部事件发生时(例如输入数据到达),则发生这种转换。如果这时没有其他进程运行,则转换(3)立即被触发,该进程便开始运行。

3.1.4 进程举例

Linux 系统中,用户在程序中可以通过调用 fork 系统调用来创建进程。调用进程叫父进程(Parent),被创建的进程叫子进程(Child)。现在举一个简单的 C 程序 forktest.c,说明进程的创建及进程的并发执行。

```
#include <sys/types.h>          /* 提供类型 pid_t 的定义,在 PC 上与 int 型相同 */
#include <unistd.h>             /* 提供系统调用的定义 */
#include <stdio.h>              /* 提供基本输入输出函数,如 printf */

void do_something(long t)
{
        int i = 0;
        for (i = 0; i < t; i++)
            for (i = 0; i < t; i++)
                for (i = 0; i < t; i++)
                        ;
```

```
}
int main()
{
    pid_t pid;

    /* 此时仅有一个进程 */
    printf("PID before fork(): %d\n", getpid());
    pid = fork();

    /* 此时已经有两个进程在同时运行 */
    pid_t npid = getpid();
    if (pid < 0 )
        perror("fork error\n");
    else if (pid == 0) { /* pid == 0 表示子进程 */
        while (1) {
            printf("I am child process, PID is %d\n", npid);
            do_something(10000000);
        }
    } else if (pid >= 0) { /* pid > 0 表示父进程 */
        while (1) {
            printf("I am father process, PID is %d\n", npid);
            do_something(10000000);
        }
    }
    return 0;
}
```

在 Linux 上运行的每个进程都有一个唯一的进程标识符 PID(Process Identifier)。从进程 ID 的名字就可以看出,它就是进程的身份证号码。每个人的身份证号码都不会相同,每个进程的进程 ID 也不会相同。系统调用 getpid()就是获得进程标识符。pid_t 是用于定义进程 PID 的一个类型,而实际上就是 int 型。

先来编译并运行这个程序:

```
$ ./fork_test
PID before fork():3991
I am child process, PID is 3992
I am child process, PID is 3992
I am father process, PID is 3991
I am child process, PID is 3992
I am father process, PID is 3991
I am child process, PID is 3992
…
```

可以看到这里输出了“child process”和“father process”,它们的 PID 是不一样的,而且是在“不规则”的交替出现。这其实这就是进程的创建和并发执行了。

从概念上讲,fork()就像细胞的裂变,调用 fork()的进程就是父进程,而新裂变出的进程就是子进程。新创建的进程与父进程几乎完全相同,只有少量属性必须不同,例如,每个进程的 PID 必须是唯一的。调用 fork()后,子进程被创建,此时父进程和子进程都从这个系统调用内部继续运行。为了区分父进程和子进程,fork()给两个进程返回不同的值。对

父进程,fork()返回新创建子进程的进程标识符(PID),而对子进程,fork()返回值为0,这一概念表示在图3.4中。

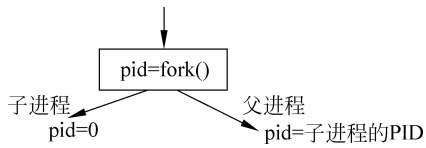

图3.4 fork()产生子进程

当上面那个程序运行时,它会不断地输出信息。第一行将显示fork()被执行前进程的PID,其余的输出行将在fork()执行后由父进程和子进程产生,也就是说,当执行到fork()这个系统调用时,一个进程裂变为两个进程,这两个进程并发执行,到底哪个进程先执行,在这里没有控制,不过,系统一般默认子进程先执行。

再输入ps命令查看一下目前系统中进程的状态和关系:

```
$ ps lf
  UID   PID  PPID PRI  NI   STAT  TTY  TIME  COMMAND
 1000  3836  2204  20   0    Ss   pts/2      0:00 bash
 1000  4008  3836  20   0    R+    pts/2      0:00  \_ ps lf
 1000  2206  2204  20   0    Ss   pts/0      0:00 bash
 1000  3391  2206  20   0    R+    pts/0      0:00  \_ ./fork_test
 1000  3392  3391  20   0    R+    pts/0      0:00      \_ ./fork_test
…
```

为了清晰起见,删除了部分列。这里主要说明其中的PID和PPID列,它们分别表示本进程的PID和父进程的PID。可以看到PID为3391的fork_test和PID为3392的fork_test,尽管名字相同,因PID不同实际上是两个不同的进程。3391的父进程是PID为2206的bash进程,3392的父进程就是3391.

通过这个简单的例子可以使读者对进程有初步的认识,尤其是初步感受一下进程的并发执行。对这个例子的进一步理解请看3.6节。

3.2 Linux系统中的进程控制块

操作系统为了对进程进行管理,就必须对每个进程在其生命周期内涉及的所有事情进行全面的描述。例如,进程当前处于什么状态,它的优先级是什么,它是正在CPU上运行还是因某些事件而被阻塞,给它分配了什么样的地址空间,允许它访问哪个文件等。所有这些信息在内核中可以用一个结构体来描述——Linux中把对进程的描述结构叫做task_struct:

```
struct task_struct {
  …
  …
  };
```

传统上,这样的数据结构被叫做进程控制块PCB(Process Control Block)。Linux中

PCB是一个相当庞大的结构体，将它的所有域按其功能可分为以下几类。

(1) 状态信息——描述进程动态的变化，如就绪态、等待态、僵死态等。

(2) 链接信息——描述进程的亲属关系，如祖父进程、父进程、养父进程、子进程、兄进程、孙进程等。

(3) 各种标识符——用简单数字对进程进行标识，如进程标识符、用户标识符等。

(4) 进程间通信信息——描述多个进程在同一任务上协作工作，如管道、消息队列、共享内存、套接字等。

(5) 时间和定时器信息——描述进程在生存周期内使用CPU时间的统计、计费等信息。

(6) 调度信息——描述进程优先级、调度策略等信息，如静态优先级、动态优先级、时间片轮转、高优先级以及多级反馈队列等的调度策略。

(7) 文件系统信息——对进程使用文件情况进行记录，如文件描述符、系统打开文件表、用户打开文件表等。

(8) 虚拟内存信息——描述每个进程拥有的地址空间，也就是进程编译连接后形成的空间。

(9) 处理器环境信息——描述进程的执行环境(处理器的各种寄存器及堆栈等)，这是体现进程动态变化最主要的场景。

在进程的整个生命周期中，系统(也就是内核)总是通过PCB对进程进行控制的，也就是说，系统是根据进程的PCB感知进程的存在的。例如，当内核要调度某进程执行时，要从该进程的PCB中查出其运行状态和优先级；在某进程被选中投入运行时，要从其PCB中取出其处理机环境信息，恢复其运行现场；进程在执行过程中，当需要和与之合作的进程实现同步、通信或访问文件时，也要访问PCB；当进程因某种原因而暂停执行时，又需将其断点的处理机环境保存在PCB中。所以说，PCB是进程存在和运行的唯一标志。

当系统创建一个新的进程时，就为它建立了一个PCB；进程结束时又收回其PCB，进程随之也消亡。PCB是内核中被频繁读写的数据结构，故应常驻内存。

进程的另外一个名字是任务(Task)。Linux内核通常把进程也叫任务，在本书中交替使用这两个术语。另外，在Linux内核中，对进程和线程也不做明显的区别，也就是说，进程和线程的实现采取了同样的方式。

下面主要讨论PCB中进程的状态、标识符和进程间的父子关系。

3.2.1 进程状态

从操作系统的原理知道，进程一般有三种基本状态——执行态、就绪态和等待态，但是在具体操作系统的实现中，设计者根据具体需要可以设置不同的状态。在Linux的设计中，考虑到任一时刻在CPU上运行的进程最多只有一个，而准备运行的进程可能有若干个，为了管理上的方便，把就绪态和运行态合并为一个状态叫就绪态，这样系统把处于就绪态的进程放在一个队列中，调度程序从这个队列中选中一个进程投入运行。而等待态又被划分为两种，除此之外，还有暂停状态和僵死状态，这几个主要状态描述如下。

(1) 就绪态(TASK_RUNNING)：正在运行或准备运行，处于这个状态的所有进程组成就绪队列。

(2) 睡眠(或等待)态：分为浅度睡眠态和深度睡眠态。

① 浅度睡眠态(TASK_INTERRUPTIBLE)：进程正在睡眠(被阻塞)，等待资源有效时被唤醒，不仅如此，也可以由其他进程通过信号[①]或时钟中断唤醒。

② 深度睡眠态(TASK_UNINTERRUPTIBLE)：与前一个状态类似，但其他进程发来的信号和时钟中断并不能打断它的熟睡。

(3) 暂停状态(TASK_STOPPED)：进程暂停执行，比如，当进程接收到如下信号后，进入暂停状态。

① SIGSTOP——停止进程执行。

② SIGTSTP——从终端发来信号停止进程。

③ SIGTTIN——来自键盘的中断。

④ SIGTTOU——后台进程请求输出。

(4) 僵死状态(TASK_ZOMBIE)：进程执行结束但尚未消亡的一种状态。此时，进程已经结束且释放大部分资源，但尚未释放其 PCB。

图 3.5 给出了 Linux 进程状态的转换及其所调用的内核函数。

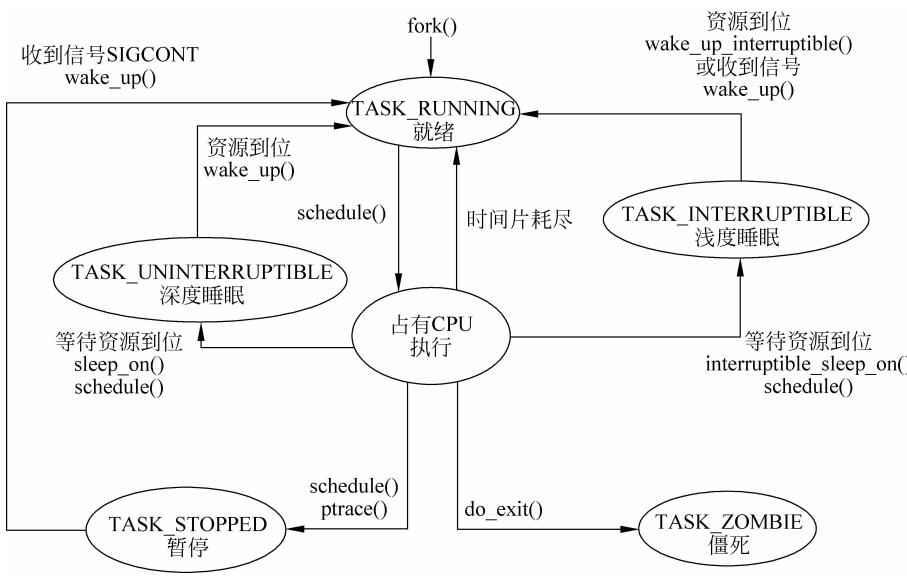

图 3.5 Linux 进程状态的转换及其所调用的内核函数

如图 3.5 所示，通过 fork()创建的进程处于就绪状态，其 PCB 进入就绪队列。如果调度程序 schedule()运行，则从就绪队列中选择一进程投入运行而占有 CPU。在进程执行的过程中，因为输入/输出等原因调用 interruptible_sleep_on()或者 sleep_on()，则进程进入浅度睡眠或者深度睡眠。由于进程进入睡眠状态放弃 CPU，因此调用调度程序 schedule()重新从就绪队列中调用一个进程运行。以此类推，读者可以自己理解图 3.5 中调用其他函数的意义。

在 task_struct 结构中(定义为 shed.h)，状态域定义为：

① 信号是一个很短的信息，通常用来通知进程产生了异步事件。

```
struct tast_struct{
  volatile long state;     /* - 1 unrunnable, 0 runnable, > 0 stopped */
  …
  };
```

其中 volatile 是一种类型修饰符，它告诉编译程序不必优化，从内存读取数据而不是寄存器，以确保状态的变化能及时地反映出来。

对每个具体的状态赋予一个常量，有些状态是在新的内核中增加的：

```
#define TASK_RUNNING            0
#define TASK_INTERRUPTIBLE      1
#define TASK_UNINTERRUPTIBLE    2
#define __TASK_STOPPED          4
#define __TASK_TRACED           8     /* 由调试程序暂停进程的执行 */
/* in tsk->exit_state */
#define EXIT_ZOMBIE             16
#define EXIT_DEAD               32    /* 最终状态，进程将被彻底删除，但需要父进程来回收 */
/* in tsk->state again */
#define TASK_DEAD               64    /* 与 EXIT_DEAD 类似，但不需要父进程回收 */
#define TASK_WAKEKILL           128   /* 接收到致命信号时唤醒进程，即使深度睡眠 */
```

也可以使用 ps 命令查看进程的状态。

3.2.2 进程标识符

每个进程都有进程标识符、用户标识符、组标识符。

不管对内核还是普通用户来说，怎么用一种简单的方式识别不同的进程呢？这就引入了进程标识符(PID)，每个进程都有一个唯一的标识符，内核通过这个标识符来识别不同的进程，同时，进程标识符 PID 也是内核提供给用户程序的接口，用户程序通过 PID 对进程发号施令。PID 是 32 位的无符号整数，它被顺序编号：新创建进程的 PID 通常是前一个进程的 PID 加 1。在 Linux 上允许的最大 PID 号是由变量 pid_max 来指定，可以在内核编译的配置界面里配置 0x1000 和 0x8000 两种值，即在 4096 以内或是 32 768 以内。当内核在系统中创建进程的 PID 大于这个值时，就必须重新开始使用已闲置的 PID 号。

```
#define PID_MAX_DEFAULT (CONFIG_BASE_SMALL ? 0x1000 : 0x8000)
int pid_max = PID_MAX_DEFAULT;
```

这个最大值很重要，因为它实际上就是系统中允许同时存在的进程的最大数目。尽管最大值对于一般的桌面系统足够用了，但是大型服务器可能需要更多进程。这个值越小，转一圈就越快。如果确实需要，可以不考虑与老式系统的兼容，由系统管理员通过修改/proc/sys/kernel/pid_max 来提高上限。可以通过 cat 命令查看系统 pid_max 的值。

```
$ cat /proc/sys/kernel/pid_max
$ 32768
```

另外，每个进程都属于某个用户组。task_struct 结构中定义有用户标识符 UID(User Identifier)和组标识符 GID(Group Identifier)。它们同样是简单的数字，这两种标识符用于系统的安全控制。系统通过这两种标识符控制进程对系统中文件和设备的访问。

3.2.3 进程之间的亲属关系

系统创建的进程具有父子关系。因为一个进程能创建几个子进程,所以子进程之间有兄弟关系。在PCB中引入几个域来表示这些关系。如前所述,进程1(init)是所有进程的祖先,系统中的进程形成一棵进程树。为了描述进程之间的父子及兄弟关系,在进程的PCB中就要引入几个域。假设P表示一个进程,首先要有一个域描述它的父进程;其次,有一个域描述P的子进程,因为子进程不止一个,因此让这个域指向年龄最小的子进程;最后,P可能有兄弟,于是用一个域描述P的长兄进程(Old Sibling),一个域描述P的弟进程(Younger Sibling)。

通过上面对进程状态、标识符及亲属关系的介绍,我们可以把这些域描述如下:

```
struct task_struct{
    volatile long state;                    /* 进程状态 */
    int pid,uid,gid;                        /* 一些标识符 */
    struct task_struct * real_parent;       /* 真正创建当前进程的进程,相当于亲生父亲 */
    struct task_struct * parent;            /* 相当于养父 */
    struct list_head children;              /* 子进程链表 */
    struct list_head sibling;               /* 兄弟进程链表 */
    struct task_struct * group_leader;      /* 线程组的头进程 */
    …
    };
```

这里说明一点,一个进程可能有两个父亲,一个为亲生父亲,一个为养父。因为父进程有可能在子进程之前销毁,就得给子进程重新找个养父,但大多数情况下,生父和养父是相同的,如图3.6所示。

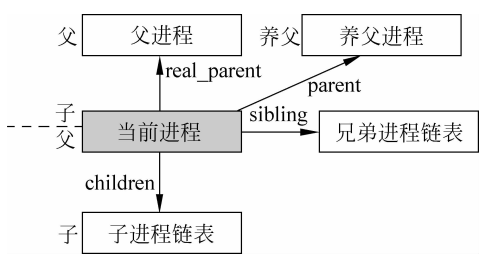

图3.6 Linux父子进程之间关系图

3.2.4 进程控制块的存放

当创建一个新的进程时,内核首先要为其分配一个PCB(task_struct结构)。那么,这个PCB存放在何处?怎样找到这个PCB?

每当进程从用户态进入内核态后都要使用栈,这个栈叫做进程的内核栈。当进程一进入内核态,CPU就自动设置该进程的内核栈,这个栈位于内核的数据段上。为了节省空间,Linux把内核栈和一个紧挨近PCB的小数据结构thread_info放在一起,占用8KB的内存区,如图3.7所示。

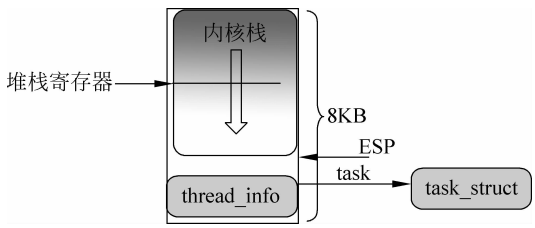

图 3.7 PCB 和内核栈的存放

在 Intel 系统中，栈起始于末端，并朝这个内存区开始的方向增长。从用户态刚切换到内核态以后，进程的内核栈总是空的，因此，堆栈寄存器 ESP 直接指向这个内存区的顶端。在图 3.7 中，从用户态切换到内核态后，只要把数据写进栈中，堆栈寄存器的值就朝箭头方向递减，p 表示 thread_info 的起始地址。而 task 是 thread_info 的第一个数据项，所以只要找到 thread_info 就能很容易找到当前运行的 task_struct 了。

C 语言使用下面的联合结构表示这样一个混合结构：

```
union thread_union {
    struct thread_info thread_info;
    unsigned long stack[THREAD_SIZE/sizeof(long)];     /* 大小一般是 8KB,但也可以配置为
4KB.本书以 8KB 来叙述。 */
};
```

从这个结构可以看出，内核栈占 8KB 的内存区。实际上，进程的 PCB 所占的内存是由内核动态分配的，更确切地说，内核根本不给 PCB 分配内存，而仅仅给内核栈分配 8KB 的内存，并把其中的一部分让给 PCB 使用。

在 x86 上，其中 thread_info 结构在文件 asm/thread_info.h 中定义如下：

```
struct thread_info{
   struct task_sturct   * task;
   struct exec_domain   * exec_domain;
   …
};
```

thread_info 结构并不代表与线程相关信息，而是表示和硬件关系更紧密的一些数据。thread_info 和 task_struct 结构中都有一个域指向对方，因此是一一对应的关系。之所以定义一个 thread_info 结构，原因之一可能是，进程控制块的所有成员中被引用最频繁的是 thread_info。另一个原因可能是，随着 Linux 版本的变化，进程控制块的内容越来越多，所需空间越来越大，这样就使得留给内核栈的空间越来越小，因此把部分进程控制块的内容移出这个空间，只保留访问频繁的 thread_info。

3.2.5 当前进程

从效率的观点来看，刚才所讲的 thread_info 结构与内核栈放在一起的最大好处是，内核很容易从 ESP 寄存器的值获得当前在 CPU 上正在运行的 thread_info 结构的地址。事实上，如果 thread_union 结构长度是 8KB(2^{13} 字节)，则内核屏蔽 ESP 的低 13 位有效位就

可以获得 thread_info 结构的基地址；这可以由 current_thread_info()函数来完成，它产生如下一些汇编指令：

```
movl $0xffffe000, %ecx
andl %esp, %ecx
movl %ecx, p
```

这三条指令执行以后，*p* 就指向进程的 thread_info 结构的指针。

进程最常用的是 task_struct 结构的地址而不是 thread_info 结构的地址，为了获得当前在 CPU 上运行进程的 PCB 指针，内核要调用 current 宏，该宏本质上等价于 current_thread_info()->task。

可以把 current 作为全局变量来使用，例如，current->pid 返回当前正在执行的进程的标识符。对于当前进程，可以通过下面的代码获得其父进程的 PCB。

```
struct  task_struct *my_parent = current->parent;
```

3.3 Linux 系统中进程的组织方式

在一个系统中，通常可以拥有数十个、数百个乃至数千个进程，相应地就有这么多 PCB。为了能有效地对它们加以管理，应该用适当的方式将这些 PCB 组织起来。

3.3.1 进程链表

为了对给定类型的进程(例如在可运行状态的所有进程)进行有效的搜索，内核建立了几个进程链表。每个进程链表由指向进程 PCB 的指针组成。在 task_struct 结构中有如下的定义：

```
struct task_struct {
…
    struct list_head tasks;
    char comm[TASK_COMM_LEN];/*可执行程序的名字(带路径)*/
…
};
```

因此，图 3.8 中通过一个双向循环链表把所有进程联系起来，我们叫它为进程链表。

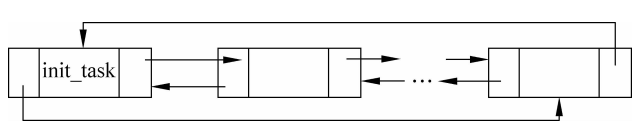

图 3.8 进程链表

链表的头和尾都为 init_task。init_task 是 0 号进程的 PCB，0 号这个进程永远不会被撤销，它的 PCB 被静态地分配到内核数据段中，也就是说 init_task 的 PCB 是预先由编译器分配的，在运行的过程中保持不变，而其他 PCB 是在运行的过程中，由系统根据当前的内存状况随机分配的，撤销时再归还给系统。

自己可编写一个内核模块,打印进程的 PID 和进程名,模块中主要函数的代码如下:

```
static int print_pid( void)
{
        struct task_struct * task, * p;
        struct list_head * pos;
        int count = 0;
        printk("Hello World enter begin:\n");
        task = &init_task;
        list_for_each(pos,&task->tasks)
                {
                p = list_entry(pos, struct task_struct, tasks);
                count++;
                printk("%d--->%s\n",p->pid,p->comm);
                }
        printk(the number of process is: %d\n",count);
        return 0;
}
```

需要注意的是,在一个拥有大量进程的系统中通过重复来遍历所有的进程是非常耗时的。

3.3.2 哈希表

在有些情况下,内核必须能根据进程的 PID 导出对应的 PCB。顺序扫描进程链表并检查 PCB 的 PID 域是可行但相当低效的。为了加速查找,引入了哈希表,于是要有一个哈希函数把 PID 转换成表的索引,Linux 用一个叫做 pid_hashfn 的宏来实现:

```
#define pid_hashfn(x) \
    ((((x) >> 8) ^ (x)) & (PIDHASH_SZ - 1))
```

其中,PIDHASH_SZ 为表中元素的最大个数,通过 pid_hashfn()这个函数,可以把进程的 PID 均匀地散列在哈希表中。

对于一个给定的 PID,可以通过 find_task_by_pid()函数快速地找到对应的进程:

```
static inline struct task_struct * find_task_by_pid(int pid)
{
        struct task_struct *p, **htable = &pidhash[pid_hashfn(pid)];
        for(p = *htable; p && p->pid != pid; p = p->pidhash_next);

        return p;
};
```

其中 pidhash 是哈希表,其定义为:struct task_struct * pidhash[PIDHASH_SZ]。

在数据结构课程中我们已经了解到,哈希函数并不总能确保 PID 与表的索引一一对应,两个不同的 PID 散列到相同的索引上称为冲突。

Linux 利用链地址法来处理冲突的 PID,也就是说,每一个表项是由冲突的 PID 组成的双向链表,task_struct 结构中由两个域 pidhash_next 和 pidhash_prev 来实现这个链表,同一链表中 PID 由小到大排列,如图 3.9 所示。

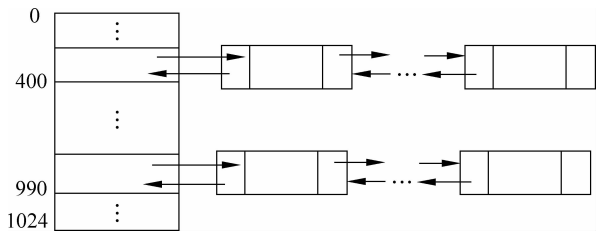

图 3.9 链地址法处理冲突时的哈希表

3.3.3 就绪队列

当内核要寻找一个新的进程在 CPU 上运行时,必须只考虑处于就绪状态的进程,因为扫描整个进程链表是相当低效的,所以把可运行状态的进程组成一个双向循环链表,也叫就绪队列(runqueue)。

就绪队列容纳了系统中所有准备运行的进程,在 task_struct 结构中定义了双向链表。

```
struct task_struct{
 …
struct list_head run_list;
…
};
```

就绪队列的定义以及相关操作在/kernel/sched.c 文件中:

```
static LIST_HEAD(runqueue_head);     /* 定义就绪队列头指针为 runqueue_head */

    add_to_runqueue()函数向就绪队列中插入进程的 PCB.
static inline void add_to_runqueue(struct task_struct * p)
{           list_add_tail(&p->run_list, &runqueue_head);
            nr_running++;            /* 就绪进程数加 1 */
}

    move_last_runqueue()函数从就绪队列中删除进程的 PCB.
static inline void move_last_runqueue(struct task_struct * p)
{
    list_del(&p->run_list);
   list_add_tail(&p->run_list, &runqueue_head);
};
```

以上讲述的进程组织方式,实际上大多数是数据结构中数据的组织方式,因此,读者在阅读本书或者源代码的过程中,首先要抓住事物的本质,找出熟悉的知识,然后,再去体会或应用已有的知识解决问题。

另外,以上是 Linux 2.4 版本中就绪队列简单的组织方式。为了让读者尽可能先掌握原理,因此本章在讨论相关内容的时候将会去繁存简,将其中最核心的调度理论和算法做阐述。

3.3.4 等待队列

如前说述，睡眠有两种相关的进程状态：TASK_INTERRUPTIBLE 和 TASK_UNINTERRUPTIBLE。它们的唯一区别是处于 TASK_UNINTERRUPTIBLE 的进程会忽略信号，而处于 TASK_INTERRUPTIBLE 状态的进程如果接收到一个信号会被提前唤醒并响应该信号。两种状态的进程位于同一个等待队列上，等待某些事件，不能够运行。

等待队列在内核中有很多用途，尤其对中断处理、进程同步及定时用处更大。因为这些内容在以后的章节中会讨论，我们只在这里说明。进程经常等待某些事件的发生，例如，等待一个磁盘操作的终止，等待释放系统资源，或等待时间走过固定的间隔。等待队列实现在事件上的条件等待，也就是说，希望等待特定事件的进程把自己放进合适的等待队列，并放弃控制权。因此，等待队列是一组睡眠的进程，当某一条件变为真时，由内核唤醒它们。

1. 等待队列的数据结构

在 include/linux/wait.h 中，对等待队列的定义如下：

```
struct __wait_queue {
    unsigned int flags;
  #define WQ_FLAG_EXCLUSIVE    0x01
    void * private;
    wait_queue_func_t func;
    struct list_head task_list;
  };
typedef struct _ _wait_queue wait_queue_t;
```

在内核代码中，以两个下划线为开头的标识符一般都是内核内部定义的。typedef 对内部定义重新封装。

在这个结构中，最主要的域是 task_list，它把处于睡眠状态的进程链接成双向链表。睡眠是暂时的，把它唤醒继续运行才是目的。为此，设置了 func 域，该域指向唤醒函数，用于把等待队列中的进程唤醒：

```
typedef int ( * wait_queue_func_t)(wait_queue_t * wait, unsigned mode, int flags, void * key);
```

如何唤醒等待队列中的进程？还需进一步根据等待的原因进行归类。比如，因为争夺某个临界资源，有一组进程由此睡眠，那么在唤醒时，是把这一组全部唤醒还是唤醒其中一个？如果全部唤醒，但实际上只能有一个进程使用临界资源，其他进程还得继续回到睡眠状态，因此仅唤醒等待队列中的一个进程才有意义。结构中的 flags 域就是为了区分睡眠时的互斥进程和非互斥进程。对于互斥进程，flags 的取值为 1(#define WQ_FLAG_EXCLUSIVE0x01)，反之，取值为 0。还有一个 private 域，是传递给 func 函数的参数。

2. 等待队列头

每个等待队列都有一个等待队列头(wait queue head)，定义如下：

```
struct _ _wait_queue_head {
    spinlock_t lock;
```

```
    struct list_head task_list;
};
typedef struct _ _wait_queue_head wait_queue_head_t;
```

因为等待队列是由中断处理程序和主要内核函数修改的,因此必须对其双向链表保护以免对其进行同时访问,因为同时访问会导致不可预测的后果(参见第 7 章)。通过 lock 自旋锁域进行同步,而 task_list 域是等待进程链表的头。图 3.10 是等待队列以及队列头形成的双向链表。

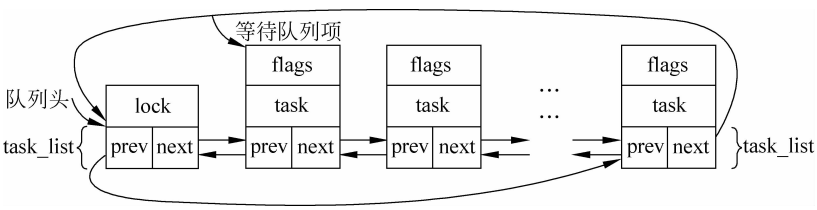

图 3.10 等待队列

3. 等待队列的操作

在使用一个等待队列前,首先需要对等待队列头和等待队列进行初始化,wait.h 中定义了如下宏:

```
#define _WAIT_QUEUE_HEAD_INITIALIZER(name) {                              \
    .lock           = _SPIN_LOCK_UNLOCKED(name.lock),                  \
   .task_list       = { &(name).task_list, &(name).task_list } }

#define DECLARE_WAIT_QUEUE_HEAD(name) \
    wait_queue_head_t name = _WAIT_QUEUE_HEAD_INITIALIZER(name)
```

这两个宏声明并初始化等待队列头 name。

初始化等待队列中的一个元素,则调用如下函数:

```
static inline void init_waitqueue_entry(wait_queue_t *q, struct task_struct *p)
{
      q->flags = 0;
      q->private = p;
      q->func = default_wake_function;
}
```

default_wake_function()唤醒睡眠非互斥进程 p,然后从等待队列链表中将其删除.

定义了一个等待进程后,必须把它插入等待队列。add_wait_queue()把一个非互斥进程插入等待队列链表的第一个位置。add_wait_queue_exclusive()把一个互斥进程插入等待队列链表的最后一个位置。remove_wait_queue()从等待队列链表中删除一个进程。waitqueue_active()检查一个给定的等待队列是否为空。

如何让等待特定条件的进程去睡眠,内核提供了多个函数。下面介绍最基本的睡眠函数

```
sleep_on():
```

```
void sleep_on(wait_queue_head_t *wq)
{
    wait_queue_t wait;
    init_waitqueue_entry(&wait, current);
    current->state = TASK_UNINTERRUPTIBLE;
    add_wait_queue(wq,&wait); /*  wq指向当前队列的头  */
    schedule();
    remove_wait_queue(wq, &wait);
}
```

该函数把当前进程的状态设置为 TASK_UNINTERRUPTIBLE,并把它插入到特定的等待队列。然后,调用调度程序,而调度程序重新调度另一个进程开始执行。当睡眠的进程被唤醒时,调度程序接着执行 sleep_on(),也就是用紧接 schedule()的 remove_wait_queue()函数,把该进程从等待队列中删除。

如果要把等待的进程唤醒,就调用唤醒函数 wake_up(),它让待唤醒的进程进入 TASK_RUNNING 状态。内核代码中,wake_up 定义为一个宏,实际上等价于下列代码片段:

```
void wake_up(wait_queue_head_t *q)
{
    struct list_head *tmp;
    wait_queue_t *curr;
    list_for_each(tmp, &q->task_list) {
        curr = list_entry(tmp, wait_queue_t, task_list);
        if (curr->func(curr,
            TASK_INTERRUPTIBLE|TASK_UNINTERRUPTIBLE,
            0, NULL) && curr->flags)
            break;
    }
}
```

list_for_each 宏扫描双向链表 q->task_list 中的所有项,即等待队列中的所有进程。对每一项,list_entry 宏都计算 wait_queue_t 型变量(curr)对应的地址。这个变量的 func 域存放 wake_up 函数的地址,它试图唤醒由等待队列中的 task_list 域标识的进程。如果一个进程已经被有效地唤醒(函数返回 1)并且进程是互斥的(curr->flags 等于 1),则循环结束。因为所有的非互斥进程总是在双向链表的开始位置,而所有的互斥进程在双向链表的尾部,所以函数总是先唤醒非互斥进程然后再唤醒互斥进程。

3.4 进程调度

在多进程的操作系统中,进程调度是一个全局性、关键性的问题,它对系统的总体设计、系统的实现、功能设置以及各个方面的性能都有着决定性的影响。进程调度算法的设计,还对系统的复杂性有着极大的影响,常常会由于实现的复杂程度而在功能与性能方面作出必要的权衡和让步。在 Linux 2.6 中为了提高性能,对调度算法进行了大幅度改进,其实现复杂度也随之增加。为了简单起见,这里以 Linux 2.4 中的调度算法来说明进程调度原理。后面附加说明 Linux 2.6 中进程调度的改进方法。

3.4.1 基本原理

从前面可以看到,进程运行时需要各种各样的系统资源,如内存、文件、打印机和最宝贵的CPU等,所以说,调度的实质就是资源的分配。系统通过不同的调度算法来实现这种资源的分配。通常来说,选择什么样的调度算法取决于资源的分配策略,一个好的调度算法应当考虑以下几个方面。

(1) 公平:保证每个进程得到合理的CPU时间。

(2) 高效:使CPU保持忙碌状态,即总是有进程在CPU上运行。

(3) 响应时间:使交互用户的响应时间尽可能短。

(4) 周转时间:使批处理用户等待输出的时间尽可能短。

(5) 吞吐量:使单位时间内处理的进程数量尽可能多。

很显然,这5个目标不可能同时达到,所以,不同的操作系统会在这几个方面作出相应的取舍,从而确定自己的调度算法,例如UNIX采用动态优先数调度、BSD采用多级反馈队列调度、Windows采用抢先式多任务调度等。

下面来了解一下主要的调度算法及其基本原理。

1. 时间片轮转调度算法

时间片(Time Slice)就是分配给进程运行的一段时间。

在分时系统中,为了保证人机交互的及时性,系统使每个进程依次地按时间片轮流地执行,此时应采用时间片轮转法进行调度。在通常的轮转法中,系统将所有的可运行(即就绪)进程按先来先服务的原则,排成一个队列,每次调度时把CPU分配给队首进程,并令其执行一个时间片。时间片的大小从几毫秒到几百毫秒不等。当执行的时间片用完时,系统发出信号,通知调度程序,调度程序便根据此信号来停止该进程的执行,并将它送到运行队列的末尾,等待下一次执行;然后,把处理机分配给就绪队列中新的队首进程,同时也让它执行一个时间片。这样就可以保证就绪队列中的所有进程,在一个给定的时间(人所能接受的等待时间)内,均能获得一个时间片的处理机执行时间。

2. 优先权调度算法

为了照顾到紧迫型进程在进入系统后便能获得优先处理,引入了最高优先权调度算法。当将该算法用于进程调度时,系统将把处理机分配给运行队列中优先权最高的进程,这时,又可进一步把该算法分成两种方式。

1) 非抢占式优先权算法(又称不可剥夺调度:Nonpreemptive Scheduling)

在这种方式下,系统一旦将处理机(CPU)分配给运行队列中优先权最高的进程后,该进程便一直执行下去,直至完成;或因发生某事件使该进程放弃处理机时,系统方可将处理机分配给另一个优先权高的进程。这种调度算法主要用于批处理系统中,也可用于某些对实时性要求不严的实时系统中。

2) 抢占式优先权调度算法(又称可剥夺调度:Preemptive Scheduling)

该算法的本质就是系统中当前运行的进程永远是可运行进程中优先权最高的那个。

在这种方式下,系统同样是把处理机分配给优先权最高的进程,使之执行。但是只要一

出现另一个优先权更高的进程，调度程序就暂停原最高优先权进程的执行，而将处理机分配给新出现的优先权最高的进程，即终止当前进程的运行。因此，在采用这种调度算法时，每当出现新的可运行进程时，就将它和当前运行进程进行优先权比较，如果高于当前进程，将触发进程调度。

这种方式的优先权调度算法，能更好地满足紧迫型进程的要求，故而常用于实时性要求比较严格的系统中，以及对性能要求较高的批处理和分时系统中。Linux 目前也采用这种调度算法。

3. 多级反馈队列调度

这是一种折中的调度算法。其本质是综合了时间片轮转调度和抢占式优先权调度的优点，即优先权高的进程先运行给定的时间片，相同优先权的进程轮流运行给定的时间片。

4. 实时调度

最后看一下实时系统中的调度。什么叫实时系统？就是系统对外部事件有求必应、尽快响应。在实时系统中存在有若干个实时进程或任务，它们用来反应或控制某个(些)外部事件，往往带有某种程度的紧迫性，因而一般采用抢占式调度方式。

3.4.2 时间片

时间片表明进程在被抢占前所能持续运行的时间。调度策略必须规定一个默认的时间片，但这并不是件简单的事。时间片过长会导致系统对交互的响应表现欠佳，让人觉得系统无法并发执行应用程序。时间片太短会明显增大进程切换带来的处理器时间，因为肯定会有相当的一部分系统时间用在进程切换上，而用来运行的时间片却很短。从上面的争论中可以看出，长时间片将导致系统交互表现欠佳。很多操作系统中都特别重视这点，所以默认的时间片很短——如 20ms。

Linux 调度程序提高交互式程序的优先级，让它们运行得更频繁。于是，调度程序提供较长的默认时间片给交互式程序。此外，Linux 调度程序还根据进程的优先级动态调整分配给它的时间片。从而保证了优先级高的进程，也应该是重要性高的进程，执行的频率高，执行时间长。通过实现这样一种动态调整优先级和时间片长度的机制，Linux 调度性能不但非常稳定而且也很强健。

3.4.3 Linux 进程调度时机

Linux 的调度程序是一个叫 schedule()的函数，这个函数被调用的频率很高，由它来决定是否要进行进程的切换，如果要切换，切换到哪个进程等。我们先来看在什么情况下要执行调度程序，Linux 调度时机主要有以下几种。

(1) 进程状态转换的时刻：进程终止、进程睡眠；

(2) 当前进程的时间片用完时；

(3) 设备驱动程序运行时；

(4) 从内核态返回到用户态时。

时机(1),进程要调用 sleep_on()或 exit()等函数时,这些函数会主动调用调度程序。

时机(2),由于进程的时间片用完时要放弃 CPU,因此也是主动调用调度程序。

时机(3),当设备驱动程序执行长而重复的任务时,直接调用调度程序。在每次反复循环中,驱动程序都检查调度标志,如果必要,则调用调度程序 schedule()主动放弃 CPU。

时机(4),不管是从中断、异常还是系统调用返回,都要对调度标志进行检测,如果必要,则调用调度程序。那么,为什么从系统调用返回时要调用调度程序呢?这当然是从提高效率考虑的。从系统调用返回意味着要离开内核态而返回到用户态,而状态的转换要花费一定的时间,因此,在返回到用户态前,系统把在内核态该处理的事全部做完。

3.4.4 进程调度的依据

调度程序运行时,要在所有处于可运行状态的进程之中选择最值得运行的进程投入运行。选择进程的依据是什么呢?在进程的 task_struct 结构中有以下几个与调度相关的域。

(1) need_resched:调度标志,决定是否调用 schedule()函数。

(2) counter:进程处于可运行状态时所剩余的时钟节拍数,每次时钟中断到来时,这个值就减 1。这个值将作为进程调度的依据,因此,也把这个域叫做进程的"动态优先级",这就巧妙地把时间片和优先级结合起来了。

(3) nice:进程的基本优先级,或叫做"静态优先级"。它的值决定 counter 的初值。这个域包含的值在-20～19 之间;负值对应"高优先级"进程,正数对应"低优先级"进程。缺省值 0 对应普通进程。这个值也可以由用户通过 nice 系统调用进行改变。

(4) policy:调度的类型,允许的取值是有以下几种。

① SCHED_FIFO:先入先出的实时进程。

② SCHED_RR:时间片轮转的实时进程。当调度程序把 CPU 分配给一个进程时,把这个进程的 PCB 就放在运行队列的末尾。这种策略确保了把 CPU 时间公平地分配给具有相同优先级的所有 SCHED_RR 实时进程。

③ SCHED_OTHER:普通的分时进程。

(5) rt_priority:实时进程的优先级。

这里要说明的是,与其他分时操作系统一样,Linux 的时间单位是"时钟节拍",Linux 设计者将一个时钟节拍定义为 10ms(在内核 2.6 版以后最小可以定义为 1ms)。在这里,把 counter 叫做进程的时间片,系统用时钟节拍数来表示,例如,若 counter 为 2,则分配给该进程的时间片就为 2 个时钟节拍,也就是 2×10ms=20ms。

以下代码片段取自 Linux 2.4。

Linux 中有一个 goodness()函数用来衡量一个处于可运行状态的进程值得运行的程度。该函数综合使用了上面提到的几个域,给每个处于可运行状态的进程赋予一个权值(Weight),调度程序以这个权值作为选择进程的唯一依据。函数主体如下(为了便于理解,笔者对函数做了一些改写和简化,只考虑单处理机的情况):

```
static inline int goodness(struct task_struct * p, struct mm_struct * this_mm)
{   int weight;            /* 权值,作为衡量进程是否运行的唯一依据 *

  weight =- 1;
```

```
  if (p->policy&SCHED_YIELD)
    goto out;              /* 如果该进程愿意"礼让(Yield)",则让其权值为-1 */
switch(p->policy)
    {
      /* 实时进程 */
      case SCHED_FIFO:
      case SCHED_RR:
          weight = 1000 + p->rt_priority;

      /* 普通进程 */
      case SCHED_OTHER:
          {    weight = p->counter;
            if(!weight)
             goto out
          /* 做细微的调整 */
          if (p->mm==this_mm||!p->mm)
                    weight = weight+1;
              weight+=20-p->nice;
          }
     }
out:
    return weight;         /* 返回权值 */
    }
```

其中,在 sched.h 中对调度策略定义如下:

```
#define SCHED_OTHER             0
#define SCHED_FIFO              1
#define SCHED_RR                2
#define SCHED_YIELD             0x10
```

这个函数比较简单。首先,根据 policy 区分实时进程和普通进程。实时进程的权值取决于其实时优先级,其至少是 1000,与 counter 和 nice 无关。普通进程的权值需特别说明以下两点。

(1) 为什么进行细微的调整? 如果 p->mm 为空,则意味着该进程无用户空间(例如内核线程),则无须切换到用户空间。如果 p->mm=this_mm,则说明该进程的用户空间就是当前进程的用户空间,该进程完全有可能再次得到运行。对于以上两种情况,都给其权值加 1,算是对它们小小的奖励。

(2) 进程的优先级 nice 是从早期 UNIX 沿用下来的负向优先级,其数值标志"谦让"的程度,其值越大,就表示其越"谦让",也就是优先级越低,其取值范围为-20~+19,因此,(20-p->nice)的取值范围就是 0~40。可以看出,普通进程的权值不仅考虑了其剩余的时间片,还考虑了其优先级,优先级越高,其权值越大。

有了衡量进程是否应该运行的标准,选择进程就是轻而易举的事情了,弱肉强食,谁的权值大谁就先运行。

根据进程调度的依据,调度程序就可以控制系统中所有处于可运行状态的进程并在它们之间进行选择。

3.4.5 调度函数 schedule()的实现

调度程序在内核中就是一个函数,为了讨论方便,同样对其进行简化,略其对 SMP 的

实现部分。

```
asmlinkage void schedule(void)
{
    struct task_struct * prev, * next, * p;     /* prev 表示调度之前的进程, next 表示调度之
                                                   后的进程 */
    struct list_head * tmp;                     /* 定义一个临时指针,指向双向链表 */
    int this_cpu, c;

  if (!current->active_mm) BUG();               /* 如果当前进程的 active_mm 为空,出错 */
need_resched_back:
      prev = current;                           /* 让 prev 成为当前进程 */
      this_cpu = prev->processor;

  if (in_interrupt()) {                         /* 如果 schedule 是在中断服务程序内部执行,
                                                   就说明发生了错误 */
         printk("Scheduling in interrupt\n");
           BUG();
      }
    release_kernel_lock(prev, this_cpu);        /* 释放全局内核锁,并开 this_cpu 的中断 */
        spin_lock_irq(&runqueue_lock);          /* 锁住运行队列,并且同时关中断 */
        if (prev->policy == SCHED_RR)           /* 将一个时间片用完的 SCHED_RR 实时
              goto move_rr_last;                   进程放到队列的末尾 */
   move_rr_back:
       switch (prev->state) {                   /* 根据 prev 的状态做相应的处理 */
           case TASK_INTERRUPTIBLE:             /* 此状态表明该进程可以被信号中断 */
                    if (signal_pending(prev)) {     /* 如果该进程有未处理的信号,则让其变
                                                       为可运行状态 */
                                prev->state = TASK_RUNNING;
                                  break;
                    }
                  default:                      /* 如果为可中断的等待状态或僵死状态 */
                    del_from_runqueue(prev);    /* 从运行队列中删除 */
                  case TASK_RUNNING:;           /* 如果为可运行状态,继续处理 */
      }
      prev->need_resched = 0;

      /* 下面是调度程序的正文 */
  repeat_schedule:                              /* 真正开始选择值得运行的进程 */
     next = idle_task(this_cpu);                /* 缺省选择空闲进程 */
    c = -1000;
      if (prev->state == TASK_RUNNING)
           goto still_running;
  still_running_back:
      list_for_each(tmp, &runqueue_head) {      /* 遍历运行队列 */
        p = list_entry(tmp, struct task_struct, run_list);
          if (can_schedule(p, this_cpu)) {      /* 单 CPU 中,该函数总返回 1 */
           int weight = goodness(p, this_cpu, prev->active_mm);
              if (weight > c)
                  c = weight, next = p;
           }
      }
```

```
/* 如果 c 为 0,说明运行队列中所有进程的权值都为 0,也就是分配给各个进程的时间片都已用完,
需重新计算各个进程的时间片 */
if (!c) {
            struct task_struct *p;
            spin_unlock_irq(&runqueue_lock);     /* 锁住运行队列 */
             read_lock(&tasklist_lock);          /* 锁住进程的双向链表 */
            for_each_task(p)                     /* 对系统中的每个进程 */
            p->counter = (p->counter >> 1) + NICE_TO_TICKS(p->nice);
            read_unlock(&tasklist_lock);
           spin_lock_irq(&runqueue_lock);
              goto repeat_schedule;
        }

    spin_unlock_irq(&runqueue_lock);             /* 对运行队列解锁,并开中断 */

  if (prev == next) {                            /* 如果选中的进程就是原来的进程 */
           prev->policy &= ~SCHED_YIELD;
              goto same_process;
     }

    /* 下面开始进行进程切换 */
      kstat.context_swtch++;                     /* 统计上下文切换的次数 */

       {
            struct mm_struct *mm = next->mm;
            struct mm_struct *oldmm = prev->active_mm;
           if (!mm) {           /* 如果是内核线程,则借用 prev 的地址空间 */
                   if (next->active_mm) BUG();
                   next->active_mm = oldmm;

            } else {            /* 如果是一般进程,则切换到 next 的用户空间 */
                    if (next->active_mm != mm) BUG();
                    switch_mm(oldmm, mm, next, this_cpu);
            }

          if (!prev->mm) {                  /* 如果切换出去的是内核线程 */
                prev->active_mm = NULL;     /* 归还它所借用的地址空间 */
                  mmdrop(oldmm);            /* mm_struct 中的共享计数减 1 */
             }
  }

       switch_to(prev, next, prev);         /* 进程的真正切换,即堆栈的切换 */
       __schedule_tail(prev);               /* 置 prev->policy 的 SCHED_YIELD 为 0 */

same_process:
      reacquire_kernel_lock(current);       /* 针对 SMP */
       if (current->need_resched)           /* 如果调度标志被置位 */
              goto need_resched_back;       /* 重新开始调度 */
       return;
}
```

以上就是调度程序的主要内容,为了对该程序形成一个清晰的思路,我们对其再给出进一步的解释。

(1) 如果当前进程既没有自己的地址空间,也没有向别的进程借用地址空间,那肯定出错。另外,如果 schedule()在中断服务程序内部执行,那也出错。

(2) 对当前进程做相关处理,为选择下一个进程做好准备。当前进程就是正在运行的进程。可是,当进入 schedule()时,其状态却不一定是 TASK_RUNNIG。例如,在 exit()系统调用中,当前进程的状态可能已被改为 TASK_ZOMBIE; 又例如,在 wait4()系统调用中,当前进程的状态可能被置为 TASK_INTERRUPTIBLE。因此,如果当前进程处于这些状态中的一种,就要把它从运行队列中删除。

(3) 从运行队列中选择最值得运行的进程,也就是权值最大的进程。

(4) 如果已经选择的进程其权值为 0,说明运行队列中所有进程的时间片都用完了(队列中肯定没有实时进程,因为其最小权值为 1000),因此,重新计算所有进程的时间片,其中宏操作 NICE_TO_TICKS 就是把优先级 nice 转换为时钟节拍。

(5) 进程地址空间的切换。如果新进程有自己的用户空间,也就是说,如果 next->mm 与 next->active_mm 相同,那么,switch_mm()函数就把该进程从内核空间切换到用户空间,也就是加载 next 的页目录。如果新进程无用户空间(next->mm 为空),也就是说,如果它是一个内核线程,那它就要在内核空间运行。因此,需要借用前一个进程(prev)的地址空间,因为所有进程的内核空间都是共享的。因此,这种借用是有效的。

(6) 宏 switch_to()进行真正的进程切换。

注意,从 schedule()退出的 return 语句并不是由 next 进程立即执行,而是稍后一点在调度程序又选择 prev 执行时由 prev 进程执行。

switch_to()的实现比较复杂,与具体的硬件体系结构有关,感兴趣的读者可以阅读相关的参考书。

3.4.6 Linux 2.6 调度程序的改进

Linux 2.4 之前的版本,用较为简单的调度算法实现了进程调度。但是,随着 Linux 服务器上多处理器(SMP)的采用以及进程数量的增加,以前的调度算法存在以下问题。

(1) 单就绪队列问题。不管进程的时间片是否耗完,都放在一个就绪队列中,这就使得时间片耗完的进程在不可能被调度的情况下,还参与调度,这是其一。其二,调度算法与系统进程数量密切相关,队列越长,选中一个进程的时间亦愈长,不适合用在硬实时系统中。

(2) 多处理器问题。多个处理器上的进程放在一个就绪队列中,使得这个就绪队列成为临界资源,各个处理器因为等待进入就绪队列而降低了系统效率。

(3) 内核态不可抢占问题。只要一个进程进入了内核态,即使有另一个非常紧迫的任务到来,它也只能等着,只有那个进程从内核态返回到用户态时,紧迫的任务才能占有处理机,这使得紧迫任务无法及时完成。

从以上分析可以看出,单就绪队列是影响调度性能的主要问题之一,因此改进就绪队列就成为改进调度算法的入口点。

1. 就绪队列

针对多处理器问题,每个 CPU 设置一个就绪队列。针对单就绪队列问题,设置两个队

列组：活跃(Active)队列组和时间片到期(Expired)队列组。每个队列组中的元素以优先级再进行分类，相同优先级的进程为一个队列，最多可以有 140 个优先级，也就是对应 140 个队列，如图 3.11 所示。

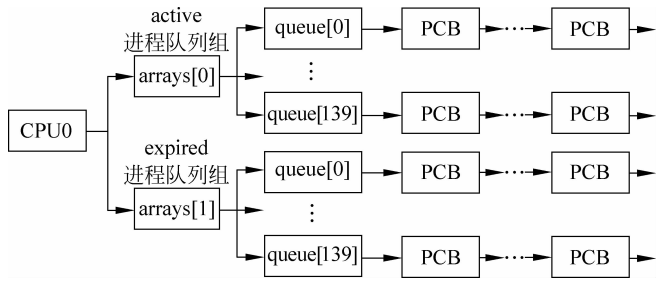

图 3.11 两组就绪进程队列

如图 3.11 所示，没有耗完时间片的进程位于活跃队列组，耗完时间片的进程存放在到期队列组，该组进程不再参与本轮调度，从而节省处理器时间。当一轮调度结束，活跃队列组变为空，所有进程时间片耗完从而进入到期队列组。这时，指向活跃队列组和到期队列组的两个指针互换，从而进入下一轮调度。

为了描述上述队列结构，同时考虑到 SMP，Linux 2.6 中为每个 CPU 定义了一个 struct runqueue 数据结构：

```
struct runqueue {
    …
    prio_array   * active, * expired, array[2];
    …
  }
```

其中，prio_array 定义为：

```
struct prio_array {
          unsigned int nr_active;                    /* 进程总数 */
          struct list_head queue[MAX_PRIO];          /* 进程链表头指针数组 */
          unsigned long bitmap[BITMAP_SIZE];         /* 进程就绪队列位图 */
};
```

runqueue 中的两个指针 active，expired 分别指向 array 数组的 array[0]和 array[1]，而这两个元素又分别指向队列数组 queue[]，进一步，queue[]数组中的每个元素存放的是就绪进程的链表头，其中每个链表中的就绪进程具有相同的优先级。

2. 就绪队列位图

从图 3.11 可以看出，一个 CPU 上就绪队列最多可达 280 个。如何从中快速选中要运行的进程成为关系系统性能的一个关键因素。为此，Linux 2.6 为这两个进程组设置了以优先级为序的就绪队列位图，该位图的每一位对应一个就绪队列，只要队列中有一个就绪进程，则对应的位被置为 1，否则置为 0。这样，调度程序无须遍历所有的就绪队列，而只需遍历位图就可选中要运行的进程。例如，当前所有进程中最高优先级为 50(换句话说，系统中

没有任何进程的优先级大于50)。则调度程序先查找位图,如果找到优先级为38的队列有就绪进程,则直接读取active[37],得到优先级为38的进程队列指针。该队列头上的第一个进程就是被选中的进程。这种算法的时间复杂度为$O(1)$,从而使得调度程序的开销与系统当前的负载(进程数)无关。

3. 优先级的动态调整

为了提高交互式进程的响应时间,$O(1)$调度程序不仅动态地提高了该类进程的优先级,还采用了以下方法。

每次时钟节拍中断时,进程的时间片减1。当时间片为0时,调度程序判断当前进程的类型,如果是交互式进程或者实时进程,则重置其时间片并重新插入active数组。如果不是交互式进程则从active数组中移到expired数组。这样实时进程和交互式进程就总能优先获得CPU。然而这些进程不能始终留在active数组中,否则进入expired数组的进程就会产生饥饿现象。当进程已经占用CPU时间超过一个固定值后,即使它是实时进程或交互式进程也会被移到expired数组中。

当active数组中的所有进程都被移到expired数组中后,调度程序交换active数组和expired数组。当进程被移入expired数组时,调度程序会重置其时间片,因此新的active数组又恢复了初始情况,而expired数组为空,从而开始新的一轮调度。

4. 调度程序的再改进

为了解决优先级动态调整等问题,大量难以维护和阅读的复杂代码被加入到Linux 2.6.0的调度模块中,虽然很多性能问题因此得到了解决,但是另外一个严重问题始终困扰着许多内核开发者,那就是代码的复杂度问题。

在2004年,Con Kolivas提出了一个改进调度程序设计的补丁——楼梯调度程序(Staircase Scheduler,SD)。为调度程序设计提供了一种新的思路。

楼梯算法(SD)在思路上和$O(1)$算法的不同在于,它抛弃了动态优先级的概念,而采用了一种完全公平的思路。$O(1)$算法的主要复杂性来自动态优先级的计算,调度程序根据平均睡眠时间和一些很难理解的经验公式来修正进程的优先级并区分交互式进程。这样的代码很难阅读和维护。

楼梯算法思路简单,但是实验证明它对交互式进程的响应比$O(1)$算法更好,而且极大地简化了代码。

楼梯算法和$O(1)$算法一样,也同样为每一个优先级维护一个进程队列,并将这些队列组织在active数组中。当选取下一个被调度进程时,SD算法也同样从active数组中直接读取进程。

与$O(1)$算法不同在于,当进程用完了自己的时间片后,并不是被移到expired数组中,而是被加入active数组的低一优先级队列中,即将其降低一个级别。不过请注意这里只是将该任务插入低一级优先级任务队列中,任务本身的优先级并没有改变。当时间片再次用完,任务被再次放入更低一级优先级任务队列中。就像一部楼梯,任务每次用完了自己的时间片之后就下一级楼梯。

任务下到最低一级楼梯时,如果时间片再次用完,它会回到初始优先级的下一级任务队

列中。比如某进程的优先级为1,当它到达最后一级台阶140后,再次用完时间片时将回到优先级为2的任务队列中,即第二级台阶。不过此时分配给该任务的时间片将变成原来的2倍。比如原来该任务的时间片为10ms,则现在变成了20ms。基本的原则是,当任务下到楼梯底部时,再次用完时间片就回到上次下楼梯的起点的下一级台阶。并给予该任务相同于其最初分配的时间片。

以上描述的是普通进程的调度算法,实时进程还是采用原来的调度策略,即FIFO或者Round Robin。

楼梯算法能避免进程饥饿现象,高优先级的进程最终会和低优先级的进程竞争,使得低优先级进程最终获得执行机会。

对于交互式应用,当进入睡眠状态时,与它同等优先级的其他进程将一步一步地走下楼梯,进入低优先级进程队列。当该交互式进程再次唤醒后,它还留在高处的楼梯台阶上,从而能更快地被调度程序选中,加速了响应时间。

楼梯算法的优点在于,从实现角度看,SD基本上还是沿用了$O(1)$的整体框架,只是删除了$O(1)$调度程序中动态修改优先级的复杂代码,淘汰了expired数组,从而简化了代码。

3.5 进程的创建

进程创建是UNIX类操作系统中发生最频繁的活动之一。例如,只要用户输入一条命令,shell进程就创建一个新进程,新进程执行shell的另一个拷贝。

很多操作系统都提供了产生进程的机制,其采取的方式是首先在新的地址空间里创建进程,然后读可执行文件,最后开始执行。UNIX采用了与众不同的实现方式,它把上述步骤分为创建和执行两步,也就是fork()和exec()两个函数。首先,fork()通过拷贝当前进程创建一个子进程。然后,exec()函数负责读取可执行文件并将其载入进程的地址空间开始运行。把这两个函数组合起来使用效果跟其他系统使用单一函数的效果类似。

传统的fork()系统调用直接把所有的资源复制给新创建的进程。这种实现过于简单并且效率低。Linux的fork()使用写时复制(Copy-on-write)来实现。也就是在调用fork()时内核并没有把父进程的全部资源给子进程复制一份,而是将这些内容设置为只读状态,当父进程或子进程试图修改某些内容时,内核才在修改之前将被修改的部分进行拷贝。因此,fork()的实际开销就是复制父进程的页表以及给子进程创建唯一的PCB。

3.5.1 创建进程

新进程是通过克隆父进程(当前进程)而建立的。fork()和clone()(用于线程)系统调用可用来建立新的进程。当这两个系统调用结束时,内核在内存中为新的进程分配新的PCB,同时为新进程要使用的堆栈分配物理页。Linux还会为新进程分配新的进程标识符。然后,新的PCB地址保存在链表中,而父进程的PCB内容被复制到新进程的PCB中。该部分也是对Linux 2.4的内核代码的说明。

在克隆进程时,Linux允许父进程和子进程共享相同的资源。可共享的资源包括文件、信号处理程序和进程地址空间等。当某个资源被共享时,该资源的引用计数值会增加1,从

而只有在两个进程均终止时,内核才会释放这些资源。

不管是 fork() 还是 clone()系统调用,最终都调用了内核中的 do_fork()函数,该函数的主要操作如下。

(1) 调用 alloc_task_struct()函数以获得 8KB 的 union task_union 内存区,用来存放进程的 PCB 和新进程的内核栈。

(2) 让当前指针指向父进程的 PCB,并把父进程 PCB 的内容拷贝到刚刚分配的新进程的 PCB 中,此时,子进程和父进程的 PCB 是完全相同的。

(3) 检查新创建这个子进程后,当前用户所拥有的进程数目有没有超出给他分配的资源的限制。

(4) 现在, do_fork()已经获得它从父进程能利用的几乎所有的东西; 剩下的事情就是集中建立子进程的新资源,并让内核知道这个新进程已经诞生。

(5) 接下来,子进程的状态被设置为 TASK_UNINTERRUPTIBLE 以保证它不会马上投入运行。

(6) 调用 get_pid()为新进程获取一个有效的 PID。

(7) 然后,更新不能从父进程继承的 PCB 的其他所有域,例如,进程间亲属关系的域。

(8) 根据传递给 clone()的参数标志,拷贝或共享打开的文件、文件系统信息、信号处理函数、进程的虚拟地址空间(参见第 4 章)等。如果进程包含有线程,则其所有线程共享这些资源,无须拷贝; 否则,这些资源对每个进程是不同的,因此被拷贝。

(9) 把新的 PCB 插入进程链表,以确保进程之间的亲属关系。

(10) 把新的 PCB 插入 pidhash 哈希表。

(11) 把子进程 PCB 的状态域设置成 TASK_RUNNING,并调用 wake_up_process()把子进程插入到运行队列链表。

(12) 让父进程和子进程平分剩余的时间片。

(13) 返回子进程的 PID,这个 PID 最终由用户态下的父进程读取。

现在有了处于可运行状态的完整子进程,但是,它还没有实际运行,由调度程序来决定何时把 CPU 交给这个子进程。在 fork()或 clone()系统调用结束时,新创建的子进程将开始执行。内核有意选择子进程首先执行,这是因为一般子进程都会马上调用 exec()函数,这样可以避免写时复制的额外开销,如果父进程首先执行,有可能会开始向地址空间写入。

子进程创建结束后,就该从内核态返回用户态了。用户态进程根据 fork()的返回值分别安排父进程和子进程执行不同的代码。

3.5.2 线程及其创建

线程是现代编程技术中常用的一种机制。该机制提供了在同一程序内可以运行多个线程,这些线程共享内存地址空间,除此之外还可以共享打开的文件和其他资源。

Linux 实现线程的机制非常独特。从内核的角度来说,它并没有线程这个概念。Linux 把所有的线程都当作进程来实现。内核并没有准备特别的调度算法或是定义特别的数据结构来表征线程。相反,线程仅仅被视为一个使用某些共享资源的进程。每个线程都拥有唯一隶属于自己的 task_struct,所以在内核中,它看起来就像是一个普通的进程,只是该进程和其他一些进程共享某些资源,如地址空间。

Linux的内核线程是由kernel_thread()函数在内核态下创建的，这个函数在内核中的实现是C语言中嵌套着汇编语言，但在某种程度上等价于下面的代码：

```
int kernel_thread(int ( * fn)(void * ), void * arg, unsigned long flags)
{
    pid_t p;
    p = clone( 0, flags | CLONE_VM );
    if ( p )                /* 父 */
        return p;
    else {                  /* 子 */
        fn(arg);
        exit();
    }
}
```

clone()有很多标志，其中CLONE_VM表示父进程和子进程共享的地址空间。在kernel_thread()返回时，父线程退出，并返回一个指向子线程的PID。子线程开始运行fn指向的函数，arg是运行时需要用到的参数。

一般情况下，内核线程会把在创建时得到的函数永远执行下去(除非系统重启)。该函数通常由一个循环构成，在需要的时候，这个内核线程就会被唤醒和执行，完成任务后，它会自动睡眠。

内核线程也可以叫内核任务，它们周期性地执行，例如，磁盘高速缓存的刷新、网络连接的维护、页面的换入换出等。在Linux中，内核线程与普通进程有一些本质的区别，从以下几个方面可以看出二者之间的差异。

(1) 内核线程执行的是内核中的函数，而普通进程只有通过系统调用才能执行内核中的函数。

(2) 内核线程只运行在内核态，而普通进程既可以运行在用户态，也可以运行在内核态。

(3) 因为内核线程只运行在内核态，因此，它只能使用大于PAGE_OFFSET(3GB)的地址空间。另一方面，不管在用户态还是内核态，普通进程可以使用4GB的地址空间(参见第4章)。

下面描述几个特殊的内核线程。

1. 进程0

内核是一个大程序，它可以控制硬件，并创建、运行、终止及控制所有进程。内核被加载到内存后，首先由完成内核初始化工作的start_kernel()函数从无到有地创建一个内核线程swap，并设置其PID为0。因为Linux对进程和线程统一编号，也把它叫进程0，又叫闲逛进程(Idle Process)。进程0执行的是cpu_idle()函数，该函数中只有一条hlt汇编指令，hlt指令在系统闲置时不仅能降低电力的使用还能减少热的产生。如前所述，进程0的PCB叫做init_task，在很多链表中起链表头的作用。当就绪队列没有其他进程时，闲逛进程0就被调度程序选中，以此达到省电的目的。

2. 进程1

如前所述，init进程是1号进程，实际上，Linux 2.6在初始化阶段首先把它建为一个内

核线程 kernel_init：

```
kernel_thread(kernel_init, NULL, CLONE_FS |CLONE_FILE S| CLONE_SIGHAND);
```

参数 CLONE_FS | CLONE_FILES | CLONE_SIGHAND 表示 0 号线程和 1 号线程分别共享文件系统(CLONE_FS)、打开的文件(CLONE_FILES)和信号处理程序(CLONE_SIGHAND)。当调度程序选择到 kernel_init 内核线程时，kernel_init 就开始执行内核的一些初始化函数将系统初始化。

那么，kernel_init()内核线程是怎样变为用户进程的呢？实际上，kernel_init()内核函数中调用了 execve()系统调用，该系统调用装入用户态下的可执行程序 init(/sbin/init)。注意，内核函数 kernel_init()和用户态下的可执行文件 init 是不同的代码，处于不同的位置，也运行在不同的状态，因此，init 是内核线程启动起来的一个普通的进程，这也是用户态下的第一个进程。init 进程从不终止，因为它创建和监控操作系统外层所有进程的活动。

3.6 与进程相关的系统调用及其应用

以上介绍的是操作系统内核对进程所进行的管理。下面从编程者的角度来说明开发人员如何利用内核提供的系统调用进行程序的开发。这一方面有助于读者对操作系统内部有进一步了解，另一方面有助于读者在应用程序的开发中充分利用系统调用来提升程序的质量。

前面我们已经对 getpid，fork，exec 等系统调用有了初步了解，下面在对这些系统调用进一步了解的基础上，另外介绍几个系统调用。

此外，在这里要说明的是每个系统调用在返回时除了返回正常值外，还要返回错误码。Linux 为了防止与正常的返回值混淆，并不直接返回错误码，而是将错误码放入一个名为 errno 的全局变量中。如果一个系统调用失败，就可以读出 errno 的值来确定问题所在。errno 不同数值所代表的错误消息定义在 errno.h 中，可以通过命令“man 3 errno”来察看它们。

3.6.1 fork 系统调用

如前所述，fork 系统调用的作用是复制一个进程。当一个进程调用它时，就出现两个几乎一模一样的进程，我们也由此得到了一个新进程。据说 fork 的名字就是来源于与叉子的形状颇有几分相似的工作流程。

回头看 2.1.4 节的进程举例。再次看到这个程序的时候，必须明确知道，在语句 pid=fork()之前，只有一个进程在执行这段代码。当执行到 fork()时，就陷入内核，具体说就是执行内核中的 do_fork()函数。于是，在这条语句之后，就变成两个进程在执行了。

fork 可能有以下三种不同的返回值。

(1) 父进程中，fork 返回新创建子进程的进程 ID；

(2) 子进程中，fork 返回 0；

(3) 如果出现错误，fork 返回一个负值。

fork 出错可能有两种原因：①当前的进程数已经达到了系统规定的上限，这时 errno

的值被设置为 EAGAIN；②系统内存不足，这时 errno 的值被设置为 ENOMEM。fork 系统调用出错的可能性很小，而且如果出错，一般都为第一种错误。如果出现第二种错误，说明系统已经没有可分配的内存，正处于崩溃的边缘，这种情况对 Linux 来说是很罕见的。

3.6.2 exec 系统调用

如果调用 fork 后，子进程和父进程几乎完全一样，而系统中产生新进程唯一的方法就是 fork，那岂不是系统中所有的进程都要一模一样吗？那要执行新的应用程序时候怎么办？多数情况下，执行完 fork 后，子进程需要执行与父进程不同的代码。例如，对于一个 shell，它首先从终端读取命令，然后创建一个子进程来执行该命令，shell 进程等待子进程执行完毕，然后再读取下一条命令。为了等待子进程结束，父进程执行一条 wait 系统调用。该系统调用使父进程阻塞，直到它的任一个子进程结束。

现在再来看 shell 如何使用 fork。当输入一条命令时，shell 首先创建一个子进程。用户的命令就是由该子进程执行，这是通过调用 exec 系统调用实现的。一个高度简化的 shell 框架如下：

```
while(TURE)                                    /* TURE 为 1,无限循环 */
  read_command(command, parameters);           /* 从终端读取命令 */
  if (fork()! = 0){                            /* 创建子进程 */
    /* Parent code */
    wait(NULL);                                /* 等待子进程结束 */
  } else {
   /* Child code */
      exec(command, parameters,0);             /* 执行命令 */
      }
  }
```

wait 系统调用等待子进程的结束。exec 有三个参数：待执行的文件名、指向参数数组的指针和指向环境变量的指针。系统提供了若干例程来简化这些参数的使用，包括 execl，execv，execle 和 execve。本书采用 exec 来泛指所有这些系统调用。

exec 函数族的作用是根据指定的文件名找到可执行文件，换句话说，就是在调用进程内部执行一个可执行文件。这里的可执行文件既可以是二进制文件，也可以是 Linux 下任何可执行的脚本文件。

与一般情况不同，exec 函数族的函数执行成功后不会返回，因为调用进程的实体都已经被新的内容取代，只留下进程 ID 等一些表面上的信息仍保持原样，颇有些神似“三十六计”中的“金蝉脱壳”。看上去还是旧的躯壳，却已经注入了新的灵魂。只有调用失败了，它们才会返回一个－1，从原程序的调用点接着往下执行。

现在应该明白 Linux 下是如何执行新程序的了，每当有进程认为自己不能为系统和用户做出任何贡献时，它就可以发挥最后一点余热，调用任何一个 exec，让自己以新的面貌重生；或者，更普遍的情况是，如果一个进程想执行另一个程序，它就可以 fork 出一个新进程，然后调用任何一个 exec，这样看起来就好像通过执行应用程序而产生了一个新进程一样。

事实上第二种情况被应用得非常普遍，以至于 Linux 专门为其做了优化，这就是前面所

说的“写时复制”技术,使得 fork 结束后并不立刻复制父进程的内容,而是到了真正实用的时候才复制,这样如果下一条语句是 exec,它就不会白白作无用功了,也就提高了效率。

3.6.3 wait 系统调用

进程一旦调用了 wait,就立即阻塞自己,由 wait 自动分析是否当前进程的某个子进程已经退出,如果它找到了这样一个已经变成僵尸的子进程,wait 就会收集这个子进程的信息,释放其 PCB,并把它彻底销毁后返回;如果没有找到这样一个子进程,wait 就会一直阻塞在这里,直到有一个出现为止。

1. 参数为空

wait 的函数原型为:pid_t wait(int *status)。

其中参数 status 用来保存被收集进程退出时的一些状态,它是一个指向 int 类型的指针。但如果我们对这个子进程是如何死掉的毫不在意,只想把这个僵尸进程消灭(事实上绝大多数情况下,都会这样想),就可以设定这个参数为 NULL,就像下面这样:

```
pid = wait(NULL);
```

如果成功,wait 会返回被收集的子进程的进程 ID,如果调用进程没有子进程,调用就会失败,此时 wait 返回-1,同时 errno 被置为 ECHILD。

下面就用一个例子来实战应用一下 wait 调用:

```
/* wait1.c */
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
#include <stdlib.h>
main()
{
        pid_t pc, pr;
        pc = fork();
        if(pc<0)                          /* 如果出错 */
                printf("error ocurred!\n");
                else if(pc == 0){         /* 如果是子进程 */
                printf("This is child process with pid of %d\n",getpid());
        sleep(10);                        /* 睡眠 10s */
        }
        else{                             /* 如果是父进程 */
                pr = wait(NULL);          /* 在这里等待 */
                printf("I catched a child process with pid of %d\n",pr);
        }
        exit(0);
}
```

编译并运行该程序:

```
$ cc wait1.c -o wait1
$ ./wait1
```

```
This is child process with pid of 1508
I catched a child process with pid of 1508
```

运行时可以明显注意到，在第 2 行结果打印出来前有 10s 的等待时间，这就是我们设定的让子进程睡眠的时间，只有子进程从睡眠中苏醒过来，它才能正常退出，也就才能被父进程捕捉到。其实不管设定子进程睡眠的时间有多长，父进程都会一直等待下去，读者如果有兴趣，可以试着自己修改一下这个数值，看看会出现怎样的结果。

另外，某些时候，父进程要等待子进程算出结果后才进行下一步的运算，或者子进程的功能是为父进程提供了下一步执行的先决条件（例如子进程建立文件，而父进程写入数据），此时父进程就必须在某一个位置停下来，等待子进程运行结束，而如果父进程不等待而直接执行下去，可能会出现极大的混乱。这种情况称为进程之间的同步，更准确地说，这是进程同步的一种特例。进程同步就是要协调好两个以上的进程，使之以安排好的次序依次执行。解决进程同步问题有更通用的方法，将在以后介绍，但对于我们假设的这种情况，则完全可以用 wait 系统调用简单地予以解决。

前面这段程序还说明，当 fork 调用成功后，父进程和子进程各做各的事情，但当父进程的工作告一段落，需要用到子进程的结果时，它就调用 wait 等待，一直到子进程运行结束，然后利用子进程的结果继续执行，这样就圆满地解决了进程同步问题。

2. 参数不为空

如果参数 status 的值不是 NULL，wait 就会把子进程退出时的状态取出并存入其中，这是一个整数值（int），指出了子进程是正常退出还是被非正常结束的（一个进程也可以被其他进程用信号结束），以及正常结束时的返回值，或被哪一个信号结束的等信息。由于这些信息被存放在一个整数的不同二进制位中，所以用常规的方法读取会非常麻烦，人们就设计了一套专门的宏（macro）来完成这项工作，下面说明其中最常用的两个。

（1）WIFEXITED(status)：这个宏用来指出子进程是否为正常退出的，如果是，它会返回一个非零值（注意，这里的 status 为整数，而 wait 的参数为指向整数的指针）。

（2）WEXITSTATUS(status)：当 WIFEXITED 返回非零值时，这个宏用来提取子进程的返回值。

3.6.4 exit 系统调用

从 exit 的名字可以看出，这个系统调用是用来终止一个进程的。无论 exit 在程序中处于什么位置，只要执行到该系统调用就陷入内核，执行该系统调用对应的内核函数 do_exit()。该函数回收与进程相关的各种内核数据结构，把进程的状态置为 TASK_ZOMBIE，并把其所有的子进程都托付给 init 进程，最后调用 schedule()函数，选择一个新的进程运行。

```
exit 的函数原型为：void exit(int status);
```

exit 系统调用带有一个整数类型的参数 status，可以利用这个参数传递进程结束时的状态，比如说，该进程是正常结束的，还是出现某种意外而结束的，一般来说，0 表示没有意外的正常结束；其他的数值表示进程非正常结束，出现了错误。在实际编程时，可以用 wait 系统调用接收子进程的返回值，从而针对不同的情况进行不同的处理。

这里要说明的是,在一个进程调用了 exit 之后,该进程并非马上就消失,而是仅仅变为僵尸状态。僵尸状态的进程(称其为僵死进程)是非常特殊的,虽然它已经放弃了几乎所有内存空间,没有任何可执行代码,也不能被调度,但它的 PCB 还没有被释放。

僵尸进程的 PCB 中保存着对程序员和系统管理员非常重要的很多信息,比如,这个进程是怎么死亡的?是正常退出呢,还是出现了错误,还是被其他进程强迫退出的?其次,这个进程占用的总系统 CPU 时间和总用户 CPU 时间分别是多少?发生缺页中断的次数和收到信号的数目又是多少?这些信息都被存放在其 PCB 中。试想如果没有僵尸状态的进程,进程一退出,所有与之相关的信息都立刻归于无形,而此时程序员或系统管理员想知道这些信息时就束手无策了。

当一个进程调用 exit 已退出,但其父进程还没有调用系统调用 wait 对其进行收集之前的这段时间里,它会一直保持僵尸状态,利用这个特点,下面给出一个简单的小程序:

```
#include <sys/types.h>
#include <unistd.h>
main()
{
        pid_t pid;
        pid = fork();
        if(pid < 0)
                printf("error occurred!\n");
        else if(pid == 0)
                exit(0);
        else
                { sleep(60);        /* 睡眠 60s,这段时间里,父进程什么也干不了 */
                wait(NULL);         /* 收集僵尸进程的信息 */
            }
}
```

sleep 的作用是指定让进程睡眠的秒数,在这 60s 内,子进程已经退出,而父进程正忙着睡觉,不可能对它进行收集,这样,就能保持子进程 60s 的僵尸状态。

那么,如何收集这些信息,并终结这些僵尸的进程呢?这就要靠前面讲到的 wait 系统调用。其作用就是收集僵尸进程留下的信息,同时使这个进程彻底消失。

3.6.5 进程的一生

下面用一些形象的比喻,来对进程短暂的一生做一个小小的总结。

随着一句 fork,一个新进程呱呱落地,但这时它只是老进程的一个克隆。然后,随着 exec,新进程脱胎换骨,离家独立,开始了独立工作的职业生涯。

人有生老病死,进程也一样,它可以是自然死亡,即运行到 main 函数的最后一个"}",从容地离我们而去;也可以是中途退场,退场有两种方式,一种是调用 exit 函数,一种是在 main 函数内使用 return,无论哪一种方式,它都可以留下留言,放在返回值里保留下来;甚至它还可能被谋杀,被其他进程通过另外一些方式结束它的生命。

进程死掉以后,会留下一个空壳,wait 站好最后一班岗,打扫战场,使其最终归于无形。这就是进程完整的一生。

3.7 系统调用及应用

以下是用户态下模拟执行命令的一个示例程序。父进程打印控制菜单，并且接收命令，然后创建子进程，让子进程去处理任务，而父进程继续打印菜单并接收命令。

```
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <string.h>

int main(int argc, char *argv[])
{
        pid_t pid;
        char cmd;
        char *arg_psa[] = {"ps", "-a", NULL};
        char *arg_psx[] = {"ps", "-x", NULL};

        while (1) {
                printf("--------------------------------\n");
                printf("输入 a 执行'ps -a'命令\n");
                printf("输入 x 执行'ps -x'命令\n");
                printf("输入 q 退出\n");
                cmd = getchar();              /*接收输入命令字符*/
                getchar();

                if ((pid = fork()) < 0){      //创建子进程
                        perror("fork error:");
                        return -1;
                }//进程创建成功
                if(pid == 0) {                /*子进程*/
                        switch (cmd) {
                                case 'a':
                                        execve("/bin/ps", arg_psa, NULL);
                                        break;
                                case 'x':
                                        execve("/bin/ps", arg_psx, NULL);
                                        break;
                                case 'q':
                                        break;
                                default:
                                        perror("wrong cmd:\n");
                                        break;
                        }               /*子进程到此结束*/
                        exit(0);  /*此处有意设置子进程提前结束，因为它的任务已
经完成
                } else if (pid > 0) {         /*父进程*/
```

```
                    if ( cmd == 'q' )
                        break;
                }
            }  /* 进程退出循环 */
            while(waitpid( -1,NULL,WNOHANG) > 0);/* 父进程等待回收子进程 */
            return 0;
}
```

3.8 小结

本章从进程的引入开始,阐述了进程的各个方面,包括进程上下文、进程层次结构、进程状态,尤其是对进程控制块进行了比较全面的介绍。task_struct 结构作为描述 Linux 进程的核心数据结构,熟悉和掌握它是深入了解进程的入口点。另外,进程控制块的各种组织方式如链表、散列表、队列等数据结构是管理和调度进程的基础。在这些基础上,对核心内容进程调度进行了代码级的描述,并给出了 Linux 新版本中改进的方法和思路。最后,以进程系统调用的剖析和应用来结束本章。

习题

1. 通过一个程序的执行过程说明程序和进程两个概念的区别。

2. 为什么要引入进程?

3. 什么是进程控制块? 它包含哪些基本信息? 打开源代码,查看 sched.h 文件中对 task_struct 的定义,确认一下你已经认识哪些域。

4. Linux 内核的状态有哪些? 请画出状态转换图,查看最新源代码,以确认有哪些状态。

5. 自己定义一个进程控制块,其中只包含状态信息、标识符及进程的亲属关系信息,写两个函数,一个函数向进程树中插入一个进程,另一个函数从进程树中删除一个进程。

6. Linux 的进程控制块如何存放? 为什么? 假设 ESP 中存放的是栈顶指针,请用三句汇编语句描述如何获得 current 的 PCB 的地址。

7. PCB 的组织方式有哪几种? 为什么要采取这些组织方式?

8. 请编写内核模块,打印系统中各进程的名字以及 PID, 同时统计系统中进程的个数。

9. 一个好的调度算法要考虑哪些方面? 为什么?

10. 查看 2.4 版本内核中 Sched.c 文件中 schedule()的实现代码,画出实现 schedule()的流程图。

11. 什么是写时复制技术,这种技术在什么情况下最能发挥其优势?

12. 查看 fork.c 中 fork 的实现代码,画出实现 fork()的流程图。

13. 0 号进程在什么时候被创建? 在什么情况下才被调度执行?

14. init 内核线程与 init 进程是一回事吗? 它们有什么本质的区别?

15. 用 fork 写一个简单的测试程序,从父进程和子进程中打印信息。信息应该包括父

进程和子进程的 PID。执行程序若干次，看两个信息是否以同样的次序打印。

16. 把 wait()和 exit()系统调用加到前一个练习中，使子进程的退出状态返回给父进程，并将它包含在父进程的打印信息中。执行若干次，观察结果。

17. 根据 3.7 节给出的例子，自己写出一个完整的程序，其中调用了进程相关的系统调用。