文章编号：1006-2475(2007)04-0121-03

信号量机制在 Linux中的实现

李雪斌，王昌晶

(江西师范大学计算机信息工程学院，江西 南昌 330022)

摘要：以源代码分析为基础，全面剖析信号量在 Linux中的实现。分析时，与基本原理相比较，重点阐述信号量在 Linux中的实现特色，从而将 Linux中的信号量实现机制较完整地呈现出来。

关键词：信号量；任务结构；等待队列；进程调度

中图分类号：TP316 文献标识码：A

Implementation of Semaphore Mechanism in Linux

LI Xue-bin, WANG Chang-jing

(Computer Information Engineering College of Jiangxi Normal University, Nanchang 330022, China)

Abstract: Based on the source code analysis in Linux 1.0, this text analyzes how the semaphore mechanism is implemented in Linux. The text also describes the feature of semaphore in Linux just compared with the related principles, so the implementation of semaphore in Linux is presented roundly.

Key words: semaphore; task structure; waiting queue; process scheduling

0 引 言

中断嵌套可能会破坏数据访问的完整性，这可由中断屏蔽来避免。但多进程/线程对共享数据的访问却是导致数据不一致的主要原因。为适应计算机系统的发展和满足用户不断提出的要求，Linux内核面临的同步问题越来越多，由此产生了多种多样的解决方案，如原子操作、自旋锁、信号量、完成变量、屏障等。信号量是在 1968年由荷兰计算机科学家 Dijkstra提出来的。目前，操作系统教科书一般都将其作为经典的系统同步机制进行介绍，但一般都仅限于原理的阐述。由于一种机制的实现往往涉及到系统的方方面面，因此如果不看其在一个系统中的具体实现，则很难真正地理解这种机制。因此，本文以 Linux内核为基础，分析信号量在其中的具体实现。

1 相关概念

1. 信号量。

信号量（semaphore）在 Linux中是一种睡眠锁。如果有一个进程试图获得一个已被占用的信号量，则信号量会将其推入一个等待队列。当持有信号量的进程将信号量释放时，处于等待队列中的进程将被唤醒，并获得该信号量。通常情况下，信号量的值为 1，即在一个时刻，最多允许一个进程占有信号量，该种信号量也称为互斥信号量。当然，也可将信号量的值设为大于 1，则该种信号量称为计数信号量。内核中基本使用互斥信号量，计数信号量只在特殊情况下才会用到，本文主要分析的是互斥信号量。

2. P/V操作。

P/V操作是两个对信号量进行操作的过程，一般定义如下：

```
procedure P(var s: semaphore)        procedure V(var s: semaphore)
begin s:=s-1;                        begin s:=s+1;
if s<0 then W(s);                    if s<=0 then R(s);
end                                  end
```

其中 W(s)表示将调用 P(s)的进程置为等待态，并排入等待队列；R(s)表示释放一个等待信号量 s

收稿日期：2006-09-19

基金项目：江西省教育厅教研课题资助项目（教务字 2004第 67号）

作者简介：（1974-），男，江西南昌人，江西师范大学计算机信息工程学院讲师，硕士，研究方向：操作系统，计算机网络；王昌晶（1978-），男，江西南昌人，讲师，硕士，研究方向：信息系统，形式化方法。

的进程,使该进程退出等待队列并设为就绪态。通常为了保证正确, P(s)和 V(s)被称为“原语”,即在关中断条件下执行,但在 Linux中有所不同。

3.任务结构。

任务结构(task structure)即一般所说的进程控制块(PCB), Linux中将“进程”称为“任务”。 task structure就是 Linux内核标识进程的数据结构, Linux中的每个进程都有一个 task structure。信号量对进程的管理就是通过对 task structure的管理而实现的。此外, Linux中的每个进程都有一个核心栈(kernel stack),用于动态分配进程进入内核执行时所需的数据空间。所有进程共享一个内核虚空间,因此分配在内核中的各种资源都有各自不同的虚地址, task structure和 kernel stack也不例外。

2 信号量机制的实现

本部分将结合源代码分析信号量在 Linux中的实现。由于目前 Linux的版本是 2.6.*,体系庞大,结构复杂,并且还考虑多 CPU结构,不适于教学需要。这里选取 Linux 1.0作为分析基础,这是 Linux的较早版本,仅支持单 CPU结构,结构精炼,适宜理解,所以文中所引用的源码均来自 Linux 1.0。

1.信号量的定义。

```
struct semaphore{ /*定义于 wait.h*/
int count;
struct wait_queue * wait; };
```

结构体 semaphore定义信号量,包含两个成员: count和指针 wait;前者代表信号量的值,若是互斥信号量,则 count初始化为 1; wait是指向 wait_queue结构的指针,该结构定义如下:

```
struct wait_queue{ /*定义于 wait.h*/
struct task_struct * task;
struct wait_queue * next; };
```

wait_queue用于管理被暂时排斥在信号量锁之外的进程,成员 next将这些进程链成一等待队列,成员 task指向对应进程的 task structure。教科书上大多是说将等待信号量的进程的 PCB直接链入等待队列,而在 linux中实际上是形成以 wait_queue为元素的等待队列。

2. P/V操作的实现。

在 Linux中, P()和 V()操作的实现分别由 down()和 up()完成。先看 up()的定义:

```
void wake_up( struct wait_queue ** q) /*睡眠的进程被一起唤醒*/
{ struct wait_queue * tmp;
struct task_struct * p;
if(!q||!(tmp=*q)) return;
do{ if((p=tmp->task)!=NULL)
{ if((p->state==TASK_UNINTERRUPTIBLE)||
(p->state==TASK_INTERRUPTIBLE))
{ p->state=TASK_RUNNING;
if(p->counter>current->counter)
need_resched=1; } }
tmp=tmp->next;
}while(tmp!=*q);
}
void up(struct semaphore * sem)
{ sem->count++;
wake_up(&sem->wait); }
```

up()操作与 V()操作基本一致,但不同的是, V()每次只唤醒一个进程,然后再由该进程去唤醒下一个进程,这么依次唤醒下去。但从 wake_up()中的 do.. while循环可看到,调用 up()操作的进程会将等待队列中的所有进程全部唤醒。但对于互斥信号量而言,唤醒的诸进程中只能有一个进入临界区,面对突然出现的诸多竞争者,系统该如何选择呢?

查看 down()的源代码就能明白 Linux如何解决这一问题。 down()的实现如下:

```
void down(struct semaphore * sem)
{ /* current宏用于获取当前进程的 task structure指针,此处用于初始化 wait */
struct wait_queue wait={ current, NULL};
add_wait_queue(&sem->wait, &wait); /*在关中断下执行*/
current->state=TASK_UNINTERRUPTIBLE;
while(sem->count<=0){
schedule(); /*由于 count<=0,当前进程被挂起,去调度其它进程执行*/
current->state=TASK_UNINTERRUPTIBLE;
}
current->state=TASK_RUNNING;
remove_wait_queue(&sem->wait, &wait); /*在关中断下执行*/
}
void down(struct semaphore * sem)
{ if(sem->count<=0) down(sem);
sem->count--;
}
```

这里 down()与 P()的不同主要有两点:

一是, 传统的 P()操作称为"原语", 要在关中断下执行, 但分析源代码可看出, down()没有全部在关中断下执行, 只是在 __down()调用下的 add_wait_queue()和 remove_wait_queue()须在关中断下执行。 Linux2.6之前的各版本规定, 当内核位于中断中时不允许进程调度, 但可置调度标志, 当进程要返回到用户态时才考虑重新调度。因此这可避免对 down()的并发执行, 也就是说不会存在两个以上的进程同时执行 down()函数, 这从另一个角度保证了"原子性"。但这不能避免由于中断嵌套而导致的错误, 因此对关键数据的访问还需关中断。所以为了安全, add_wait_queue()和 remove_wait_queue()是在关中断下执行的(二者源代码略), 因为它们要对等待队列进行入队和出队操作。所以, 无需将 down()操作全部置于关中断下执行, 否则可能会因中断关闭时间过长而丢失中断。上面的 up()操作没有关中断也是这个原因。

二是, 传统的 P()操作中, 是先将信号量的值减 1, 这样通过查看信号量的绝对值就能知道当前有多少进程在等待进入临界区。而 down()操作是先判断信号量的值(若 count<=0则当前进程挂起), 之后再减 1。因此在 Linux中无法知道当前有多少进程在等待队列中, 但这无关紧要。率先进入临界区的进程, 由于 count为 1, 不会受到 __down(sem)的阻挠, 继续执行 sem->count--, 将 count值置 0。在它未执行 up()操作之前, 临界区被锁住了。而后面想进入临界区的进程, 在调用 down()操作时, 由于 sem->count<=0成立而被 __down(sem)置为等待态(TASK_UNINTERRUPTIBLE), 并被加入到等待队列中。之后, 先前进入临界区的进程调用 up()操作, 执行 sem->count++, 将 count值又置为 1, 并唤醒等待队列中的所有进程, 将它们全部置为就绪态(TASK_RUNNING)。注意, 此时这些被唤醒的进程还没有脱离等待队列, 只有执行 remove_wait_queue()才宣告真正脱离了等待队列。这些被唤醒的进程中, 有一个被选中而获得 CPU执行, 则其必然要从 __down()中的"schedule();"处恢复, 因为当初它就是在这里被挂起的。可以看到, 从 schedule()返回后, 进程又被无条件地置为等待态, 这里不禁会产生疑问: 刚刚恢复执行怎么又要等待? 因为唤醒的进程可能不止一个, 但允许进入临界区的却只能有一个, 因此被唤醒并恢复执行的进程还要再次看看自身是否有资格进入临界区, 如果被另外的一个抢先了, 则自己只有再次等待。抢先获得 CPU执行的进程, 由于 count为 1, 循环条件 while(sem->count<=0)为假, 因此它顺利地离开循环, 执行完 __down(), 并返回到 down(), 然后继续执行 sem->count--, 又将 count置为 0, 宣告再次锁住临界区。其它被唤醒的进程当执行到 while(sem->count<=0)时, 条件为真, 发现临界区又被上锁, 只能再次回到等待队列, 等候再次被唤醒。

由上分析可看到, 仅仅知道原理还是很不够的, 没有哪个系统是完全按照原理来实现的, 因为要考虑到诸方面的因素, 如系统的条件、性能等。

3　结束语

本文以信号量原理为基础, 从一个具体的实例分析出发, 不但能加深对原理的理解, 还能加深对系统的理解。本文的源码分析基于 Linux1.0, 现在的版本是 2.6.*, 在信号量机制的实现方面有所不同, 但主要的区别在于 2.6.*中增加了对多 CPU的考虑, 其它大致是一样的。

参考文献:

[1] [美] Robert Love. Linux内核设计与实现[M]. 陈莉君, 等译. 北京: 机械工业出版社, 2004.

[2] Robert Love. Linux Kernel Development(2nd Edition)[M]. Sams Publishing, 2005.

[3] [荷] Andrew S Tanenbaum. 现代操作系统[M]. 北京: 机械工业出版社, 2002.

[4] 谭耀铭. 操作系统[M]. 北京: 中国人民大学出版社, 2000.