1813950

分类号 _____ 密级 _____

UDC[注1] _____

学 位 论 文

数据结构在操作系统进程调度中的应用研究

（题名和副题名）

张颖慈

（作者姓名）

指导教师姓名 吴 跃 教 授

电子科技大学 成 都

（职务、职称、学位、单位名称及地址）

申请专业学位级别 硕士 专业名称 软 件 工 程

论文提交日期 _____ 论文答辩日期 _____

学位授予单位和日期 电子科技大学

答辩委员会主席 _____

评阅人 _____

2009 年 月 日

注 1：注明《国际十进分类法 UDC》的类号。

独创性声明

本人声明所呈交的学位论文是本人在导师指导下进行的研究工作及取得的研究成果。据我所知，除了文中特别加以标注和致谢的地方外，论文中不包含其他人已经发表或撰写过的研究成果，也不包含为获得电子科技大学或其它教育机构的学位或证书而使用过的材料。与我一同工作的同志对本研究所做的任何贡献均已在论文中作了明确的说明并表示谢意。

签名：张韩江 日期：2009年12月1日

论文使用授权

本学位论文作者完全了解电子科技大学有关保留、使用学位论文的规定，有权保留并向国家有关部门或机构送交论文的复印件和磁盘，允许论文被查阅和借阅。本人授权电子科技大学可以将学位论文的全部或部分内容编入有关数据库进行检索，可以采用影印、缩印或扫描等复制手段保存、汇编学位论文。

（保密的学位论文在解密后应遵守此规定）

签名：张韩江 导师签名：

日期：2009年12月1日

摘 要

本文基于 linux 操作系统，分析其内核中数据结构的应用，重点对进程调度和进程控制部分的源代码进行了详细的分析研究，通过分析这两个模块代码的实现，把握数据结构在软件开发中的应用。论文对 Linux2.4 和 2.6 的内核进行了对比研究，总结了 Linux2.6 基于 Linux2.4 在实时性方面的改进。在进程调度部分，主要涉及了 Linux 的时钟中断、定时器、Linux 内核机制以及系统调用 nanosleep、pause。同时深入研究各种典型的进程调度算法，阐述其优缺点并积极寻找继续优化改进的途径。本论文针对目前标准 Linux2.6 在实时调度方面仍然存在的不足之处提出了解决策略，并对工作方案和其中重要数据结构的修改作出了详细介绍。通过对比实验测试数据，该解决方案有效提升了实时调度能力。

关键词：操作系统，数据结构，linux ，RM，EDF

ABSTRACT

Based on the linux operating system, we analyzed the data structure application of its kernel and focused on analyzing the source code of the process schedule and the process control so as to apply the data structure in the software development. In the article, we revealed that Linux2.6 was more practical than Linux2.4 based on analyzing the two system kernels. The part of process schedule mainly included the Linux clock interrupt, the timer, the kernel mechanism of Linux and stystem invoking *nanosleep* and *pause*. We further studied the various typical algorithms of process schedule, revealed the advantages and indicated to continuously search more optimal methods. The article pointed out the solution strategy according to the defect of real time scheduling of Linux2.6 and detailedly described the task laying and the modify of important data structrue. The solution strategy can effectively enhance the real time scheduling based on analysis of the trial test data.

Key Words: Oprerating System , Data Structrue, Linux, Process , RM, EDF

目 录

第一章 引言

“数据结构”作为一门独立的课程在国外是从1968年才开始设立的。1968年美国唐·欧·克努特教授开创了数据结构的最初体系，他编著的《计算机程序设计技巧》第一卷《基本算法》是第一本较系统地阐述数据的逻辑结构、存储结构及其操作的著作。“数据结构”在计算机科学中是一门综合性的专业基础课，它是介于数学、计算机硬件和计算机软件三者之间的一门核心课程。数据结构课程的内容不但是一般程序设计(尤其是非数值性程序设计）的基础，同时也是设计和实现编译程序、操作系统、数据库系统及其他系统程序的重要基础。

数据结构是计算机存储、组织数据的方式。通常情况下，精心选择的数据结构能够实现更高的运行效率或存储效率的算法。数据结构往往与高效的检索算法和索引技术有关。数据结构在计算机科学中是一门综合性的专业基础课，研究非数值计算程序设计问题中计算机的操作对象以及它们之间的关系和操作等问题。它和计算机的硬件，尤其是计算机软件的研究有着密切的关系，是介于数学、计算机硬件和软件三者之间的一门核心课程，无论是汇编程序还是操作系统，都涉及到研究数据元素在存储器中的分配问题。

数据结构不仅涉及到图论，表，树的理论，目前还扩充到网络、集合代数论、格、关系等方面。它对操作系统的发展有着推波助澜的作用，是操作系统的重要基础。而操作系统是配置在计算机硬件上的第一层软件，是一组程序的集合，其他所有的软件如汇编程序等等，都将依赖操作系统的支持，获取它的服务。同时，操作系统也离不开相关软件的支持，它的数据如何分配，软件所采取的数据结构、算法，都将大大影响到本系统的性能。既然许多学科都涉及到数据元素在存储器中的分配问题，因此不能把数据结构和操作系统视为两个独立无关的学科，而是应该把它们结合起来研究，融会贯通。

计算机解决一个实际问题时，大致需要经过下列几个步骤：首先要从具体问题中抽象出一个适当的、正确的数学模型，然后设计一个解决此数学模型的算法（Algorithm），最后编出程序、进行测试、调试直至得到最终解答。建立数学模型的实质是分析具体问题，从中提取要操作的对象，并找出这些操作对象之间包含的关系，然后运用数学的语言进行描述。计算机算法与数据的结构关系非常密切，算法必须依附于具体的数据结构，数据结构直接关系到算法的选择和运行效

率。既然运算是由计算机来完成，这就要设计与之相对应的插入、删除和修改等算法 。即数据结构还需要给出每一种结构类型所定义的各种运算的算法。

本论文是基于操作系统的角度，着眼于数据结构，分析研究其在大型软件系统中的重要地位，对系统中各类资源的表示与组织，不仅要能够最大限度的利用系统资源，同时还要使资源得到公平合理的利用[2]。在许多类型的程序设计中，数据结构的选择是一个最基本的设计考虑因素。很多大型系统的构造经验表明，系统实现的困难程度和系统构造的质量很大程度上都依赖于是否选用了最优化的数据结构。在解决实际问题的过程中，一旦确定了数据结构，算法就容易设计了。不过有些情况也会颠倒过来，需要根据特定算法来选择最相适应的数据结构。无论属于哪一类情况，选择正确合适的数据结构都是非常重要的，因为数据结构直接影响着算法的设计与执行效率。

本论文选择 Linux 作为研究对象。Linux 是一种能够在多种平台上运行、源代码公开、免费、功能强大、遵守 POSIX 标准、并且与 UNIX 兼容的操作系统。

Linux 最初版本是由 Linus Benedict Torvalds 编写的，为了能够使 Linux 更加完善，Torvalds 在网络上公开了 Linux 的源代码，邀请全世界的志愿者来参与 Linux 的完善与开发。在全世界爱好者的共同帮助下，Linux 得到不断的完善，并在短时期内迅速崛起。现在，Linux 内核还在以相当快的速度不断地发展着。事实证明，linux 是一个很有发展前途的操作系统，也是为数不多可以与 Microsoft 旗下操作系统相竞争的操作系统。[1]

我国的 IT 产业起步较晚，技术落后于西方经济发达国家。在我国，由于受知识产权的限制，无论是 PC 平台上使用的 Windows 桌面系统，还是使用应用于大中型机的 UNIX，都无法窥视到其内部结构。目前信息化社会的大环境，迫切需要有我们自己研发的安全操作系统。计算机体系中的安全包括：操作系统安全、数据库安全、网络安全和应用软件安全。操作系统是所有上层应用的接口、平台，因此如果没有安全的操作系统作为基础，那么所有的上层应用安全问题都不能得到真正的解决。既然要设计自己的安全操作系统，对现有操作系统内核的透彻研究是必不可少的工作。Linux 的开放源码为这一项工作提供了条件。

本论文将以操作系统的基本原理作为指导思想，以分析内核主要数据结构为核心，以理清数据结构间的关系为线索，以分析实现机制为前期基础工作的目的。深透理解 Linux 作为多用户任务操作系统的本质及其工作原理，思考优化改进的途径，以提高现有操作系统的效率。

第二章 linux 内核整体结构研究

2.1 Linux 内核的五大子系统分析

目前，操作系统内核的结构模式主要可分为整体式的单内核模式和层次式的微内核模式。Windows采用的是层次式的微内核模式，而Linux则是采用整体式的单内核模式，单内核模式的突出优点是内核代码结构紧凑，执行速度快，而不足之处则是层次结构性不强。[3]

在单内核模式系统中，操作系统的工作服务流程为：应用程序运行指定的参数执行系统调用指令（int x80），将 CPU 从用户态切换到核心态，然后系统根据参数值调用特定的系统调用服务程序，而这些服务程序则根据需要调用底层的支持函数以完成特定的功能。当完成了应用程序要求的服务后，操作系统又从核心态切换回用户态，回到应用程序中继续执行后续指令。因此，单内核模式的内核也可粗略地分为三层：调用服务的主程序层，执行系统调用的服务层和支持系统调用的底层函数，如图 2-1 所示：

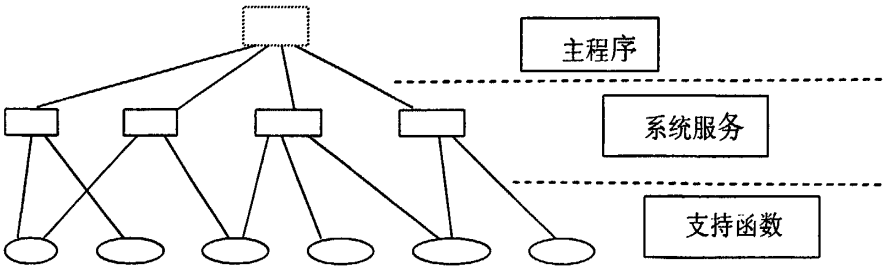

图 2-1 系统调用层次

Linux 内核由 5 个主要的子系统构成。这 5 个子系统分别是：进程调度(SCHED)、内存管理(MM)、进程间通信(IPC) 、虚拟文件系统(Virtual File System, VFS)和网络接口(NET) [3]。

进程调度控制着进程对 CPU 的占用。当需要调度下一个进程运行时，由调度程序选择最值得运行的进程。但运行进程实际上是仅等待使用 CPU 资源的进程，如果某个进程同时还在等待其它资源，则该进程是不可运行的。Linux 使用了比较简单的基于优先级和时间片的进程调度算法选择调度新的进程。

内存管理允许多个进程安全地共享主内存区域。Linux 的内存管理功能支持虚

拟内存，即在计算机中运行的程序，其代码、数据和堆栈的总量可以超过实际内存的大小，操作系统只需要将当前正在使用的程序块保留在内存中，其余的程序块保留在磁盘上。必要时，操作系统负责在磁盘和内存之间交换程序块。

内存管理从逻辑上可以分为硬件无关的部分和硬件相关的部分。硬件无关的部分提供了地址的映射和虚拟内存的对换；硬件相关的部分为内存管理硬件提供了虚拟接口。

虚拟文件系统隐藏了不同类型硬件的具体细节，为所有设备提供了统一的接口，虚拟文件系统还支持多达数十种不同的文件系统，这也是 Linux 较有特色的部分。

虚拟文件系统可分为逻辑文件系统和设备驱动程序。逻辑文件系统指 Linux 所支持的文件系统，如 ext2、fat 等，设备驱动程序指为每一种硬件控制器所编写的设备驱动程序模块。

网络接口提供了对各种网络标准的存取和各种网络硬件的支持。网络接口分为网络协议和网络驱动程序两部分。网络协议部分负责实现每一种可能的网络传输协议，网络设备驱动程序负责与硬件设备进行通信，每一种可能的硬件设备都有相应的设备驱动程序。

进程间通信支持进程间各种通信机制。

图 2-2 显示了上述五个子系统之间的关系：

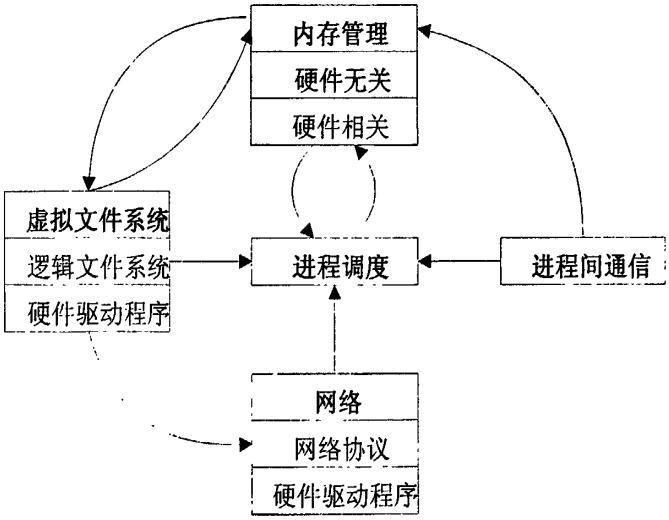

图 2-2　系统模块调用关系

各个子系统之间的依赖关系如下：

（1）进程调度与内存管理之间的关系

这两个子系统互相依赖。在多道程序环境下，某个程序要运行必须为之创建

进程，而创建进程的首要事件，就是要将程序和数据装入内存中，并且当需要让出 CPU 的控制权时，保存该进程此刻的相关信息（即进程上下文）到进程指定的内存单元中。

（2）进程间通信与内存管理之间的关系

进程间通信子系统要依赖内存管理支持共享内存通信机制，这种机制允许两个进程除了拥有自己的内存空间以外，还可对共同的内存区域进行访存。通过进程对内存区域的共享使用，来实现通信的目的。

（3）虚拟文件系统与网络接口之间的关系

虚拟文件系统通过网络接口支持网络文件系统(NFS)，也通过内存管理支持 RAMDISK 设备。

（4）内存管理与虚拟文件系统之间的关系

内存管理利用虚拟文件系统支持对换，对换进程定期地由调度程序调度，这也是内存管理依赖于进程调度的唯一原因。当某进程存取的内存映射被换出时，内存管理向文件系统发出请求，同时，挂起当前正在运行的进程。

在这些子系统中，进程调度子系统是保证其他子系统顺利工作的关键。无论是文件系统的系统进程还是网络子系统的服务进程都需要通过进程调度来获得相应的 CPU 时间来得以正常运行。

2.2 小结

本章对标准 Linux 内核整体结构进行了简单介绍，对其五大子系统各自的功能及其相互间依赖关系作了分析，为后面章节深入研究其内核源代码、实现机制等搭建了平台。

第三章 LINUX 中的数据结构及其算法研究

上一章简单介绍了 LINUX 的基本体系结构，本章针对 LINUX 中主要的数据结构，以及其某些相关的操作进行分析。

Linux 中包含了许多对象和数据结构，例如内存页面，进程和中断。如果操作系统要高效运行，那么如何及时地从多个对象中快速引用其中的一个对象是要关注的主要问题。Linux 使用链表和二叉搜索树（还有一组帮助例程）先将这些对象划分到不同的组[4]，之后再以有效的方式在对应的组中进行查找单个元素操作。

3.1 链表

在计算机科学中，链表（linked lists）是最常见的数据类型，并贯穿在整个 Linux 内核中。在 Linux 内核中，链表通常表现为循环双向链表的形式。因此，给定链表中的任一结点，均可以 O(1)的时间复杂度查找到其前驱结点和后继结点。图 3-1 是循环双向链表的示例图。

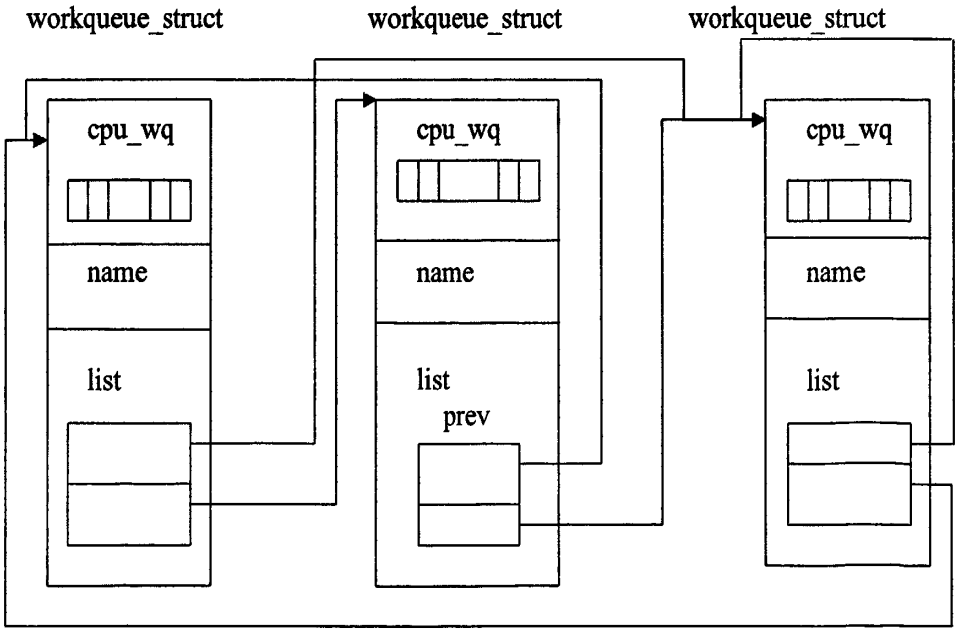

图 3-1 链表示例

有关 LINUX 链表定义的所有代码均在头文件 include/linux/list.h 中有定

义。本节对链表的主要特征做分析。

若现有一个数据结构 foo，并且需要维持一个这种数据结构的双链队列，最简单最常用的方法就是在这个数据结构的类型定义中加入两个指针，例如：

```
typedef struct foo
   {   struct foo *prev;
       struct foo *next;
        ......................
     }foo_t;
```

然后将这种数据结构套用于各种队列操作的子程序。由于用来维持队列的这两个指针的类型是固定不变的（都指向 foo 数据结构），这些子程序不能用于其他数据结构的队列操作。也就意味着，需要维持多少种数据结构的队列，就得需要多少套的队列操作子程序。对于那些使用队列较少的应用程序这或许不是个大问题，但对于使用大量队列情况的内核就存在问题了，队列操作子程序数量庞大。所以，linux 内核中采用了一套通用的、一般的，可以使用各种不同数据结构的队列操作。具体的办法是 linux 的开发人员把指针 prev 和 next 从具体的“宿主”数据结构中抽象出来独立成为一种数据结构 list_head，这种数据结构既可以“寄宿”在具体的宿主数据结构内部，成为该数据结构的一个“连接件”，也可以独立存在作为一个队列的头。这个数据结构的定义在 include/linux/list.h 中[1]。

```
struct list_head
 {
  struct list_head *next, *prev;
  };
```

这里的 list_head 没有数据域。在 Linux 内核链表中，数据并没有包含在链表结构中，而是在数据结构中包含链表节点。

如果需要有某种数据结构的队列，就在这种结构的内部放入一个 list_head 数据结构。以内存页面管理机制中的 page 数据结构为例，其定义为：

```
typedef struct page
{
 struct  list_head list;
  ..................
 struct  page *next_hash;
```

```
.....
struct  list_head lru;
.....
}mem_map_t;
```

在上面的定义中， page数据结构中寄宿了两个list_head结构，或者说包含了两个队列操作的连接件，因此 page 结构可以同时存在于两个双链队列中。此外，结构中还有个单链指针 next_hash，用来维持一个链的哈希队列。

对于宿主数据结构内部的每个 list_head 数据结构都要做初始化，可以通过一个宏操作 INTT_LIST_HEAD 进行[1]：

```
#define INIT_LIST_HEAD(ptr) do{\
    (ptr)->next=(ptr); (ptr)->prev=(ptr); \
  }while(0)
```

参数 ptr 为指向需要初始化的 list_head 结构。可见初始化以后两个指针都指向该 list_head 结构自身。要将一个 page 结构通过其“队列头”list 链入一个队列，可以使用 list_add（）函数，这是一个 inline 函数，其代码定义在 include/linux/list.h 中[4]：

```
static_inline_void list_head(struct list_head *new, struct list_head
*head)
 {
  _list_add(new,head,head->next);
 }
```

参数 new 指向欲链入队列的宿主数据结构内部的 list_head 数据结构。参数 head 指向链入点，同样也是个 list_head 结构，它可以是个独立的，真正意义上的队列头，也可以在另一个宿主数据结构（其至可以是不同类型的宿主结构）内部。该 inline 函数调用另一个 inline 函数_list_add()来完成操作：

```
[list_add( )->_list_add( )]
 Static_inline_void list_add(struct list_head *new, struct list_head
*prev, struct list_head *next)
  {
    next->prev=new;
    new->next=next;
    new->prev=prev;
```

```
    prev->next=new;
}
```

而从队列中脱离链的操作 list_del( )[1]定义如下：

```
Static_inline_void_list_del(struct list_head *entry)
    {
      _list_del(entry->prev,entry->next);
    }
```

类似的，这里也是调用另一个 inline 函数_list_del( )来完成操作：[list-del( )->_list_del( )]

```
Static_inline_void_list_del(struct list_head *prev,struct list_head
    *next)
    {
      next->prev=prev;
      prev->next=next;
    }
```

在_list_del()中的操作对象是队列中在 entry 之前和之后的两个 list_head 结构。如果 entry 是队列中的最后一项，则二者相同，就是队列的头，那也是一个 list_head 结构，但不在任何宿主结构内部。

在 linux 中，大量的队列操作都是通过 list_head 进行的，这仅仅是连接件，假如现在有一个宿主结构，能很容易知道它的某个 list_head 在哪里，从而以此为参数调用 list_add()或 list_del()；可是，反过来考虑，当我们顺着一个队列取得其中一项 list_head 结构时，如何找到其宿主结构呢？毕竟在 list_head 结构中并没有指向宿主结构的指针。而我们真正关心的是宿主结构，并非是连接件。这一问题在 linux 中是这样解决问题的，以下的一段代码：来自 mm/pasge_alloc.c 中的一行[4]：

```
page=memlist_entry(curr,struct page,list);
```

这里的 memlist_entry ( ) 将一个 list_head 指针 curr 换算成其宿主结构的起始地址，也就是取指向其宿主 page 结构的指针。而以下一行代码定义了 memlist_entry()[5]：

```
#define memlist_entry list_entry
```

所以原函数实际变成了 list_entry (curr,struct page,list)，而它的定义如下：

#define list_entry(ptr,type,member)\ container_of(ptr, type, member)

其中 ptr 是指向该数据中 list_head 成员的指针，也就是存储在链表中的地址值，type 是数据项的类型，member 则是数据项类型定义中 list_head 成员的变量名。

container_of 宏定义在[include/linux/kernel/h]中:

```
#define container_of(ptr, type, member)
( { const  typeof( ((type\ *)0 )->member ) *__mptr = (ptr);
 (type *)((char *)__mptr - offsetof(\type,member) ); })
```

offsetof 宏定义在[include/linux/stddef.h] [6]中:

```
#define offsetof(TYPE, MEMBER) ((size_t) &((TYPE *)0)->MEMBER)
```

这里使用了一个利用编译器技术的小技巧，即先求得结构成员在结构中的偏移量，然后根据成员变量的地址反过来求得宿主结构变量的地址。

将上面的 memlist_entry(curr,struct page,list) [6]替换一下，如下:

```
page= ((struct  page *) ((char *) (curr) - (unsigned long ((& ((struct
page *) 0) ->list))) ;
```

上面的curr是一个page结构内部成份list的地址,而问题所需要的却是page结构本身的地址，所以要从 curr 减去一个位移量。成份 list 在 page 内部的位移量究竟是多少呢？& ((struct page*) 0) ->list[7]就表示当结构 page 正好在地址 0 上时其成份 list 的地址，这就是所需要的位移量。

同样的道理，如果是在 page 结构的 lru 队列里，则传下来的 member 位 lru，一样能算出宿主结构的地址[5]。

综上所述，这一套操作方法既普遍适用，又保持了较高的效率。但是，阅读代码存在着一个不便之处，那就是光从代码中不容易看出一个 list_head 的宿主结构是 shenm，而从前的方法只要看一下指针 next 的类型就可以知道了。

3.2 树

树用于 Linux 内存管理中，能够有效地访问并操作数据。此时，衡量其有效性就是看存储及从若干个数据中检索指定数据的速度能有多快。本节将讨论基本树，重点是红黑树。而树在 Linux 下的实现方式及帮助例程，在 linux 的“文件系统”中占有重要位置。在计算机科学中，有根树由结点和边组成，结点代表数据元素，边代表结点之间的路径，第一个结点，或者说顶层结点，就是树的根结

点。结点之间的关系有双亲、孩子、兄弟三种。每个孩子结点有且仅有一个双亲结点（除了根结点），每个双亲结点可以有一个或多个孩子结点，兄弟结点拥有共同的双亲，终端结点没有孩子结点，又称为叶结点。树的高度是指从根结点到最远的叶结点之间的边数。树的每一行子孙称之为一层。查找给定兄弟集合中的某一数据元素时，有序树最左边的兄弟其值最小，而最右边的兄弟其值最大。树通常以链表和数组的形式实现，按某种次序在树中访问结点的过程即树的遍历。

3.2.1 二叉排序树

以前，采用线性查找的方式来查找关键字值，在每次循环中对关键值做比较。在有序表中，每次比较可以减少其中一半的结点。二叉排序树和链表不同，它是一种分层的数据结构，而不是线性的。在二叉排序树中，每个元素或结点指向一个左孩子或右孩子结点，每个孩子结点又指向一个左孩子或右孩子，依此类推。其结点之间排序规则是左孩子的关键值小于双亲，而右孩子的关键值大于或等于双亲。因此，对于一个给定结点的关键值，其左子树上所有结点的关键值均小于该结点，而其右子树上所有结点的关键值均大于或等于该结点。向二叉树中存放数据时，首先必须找到适当的插入位置，而每次循环均可减少一半待查找的数据个数。做算法时间复杂度分析，其性能（关于查找的次数）为 O(logn)，相比之下，线性查找的性能是 O(n)。

遍历二叉树的算法比较简单。对于每个结点而言，比较完该结点的关键值后就可以以相同方式遍历其左子树或右子树，二叉树的遍历可以很方便用递归来实现。下面将讨论其具体实现，辅助函数以及二叉树的类型。上文提到，二叉树中的结点可以有一个左孩子，或者一个右孩子，或者有左、右两个孩子，也可以没有孩子。二叉排序树的规则是，给定一个结点的值 x，其左孩子（包括所有子孙结点）的值小于 x，而其右孩子（包括所有子孙结点）的值大于 x。由此可知，如果将数据的有序集合插入到二叉树中，将形成一个线性列表，对于一个给定值，其查找速度就会变得和线性查找一样慢。例如，根据数据集[0,1,2,3,4,5,6]创建一颗二叉树时，0 是树根；1 比 0 大，是 0 的右孩子；2 比 1 大，是 1 的右孩子；而 3 是 2 的右孩子；依此类推，其实质是退化成为了线性查找 O(n)。

在均高二叉树[8]中，根到任意叶结点的距离都是最远的。结点添加到二叉树中后，为了保证查找的效率，必须进行平衡化处理，这可以通过旋转来实现。插入一个结点后，给定结点 e，如果它有一个比任何其他叶结点高两层的左子树，就必

须对 e 作右旋转。如图 2.4 所示，e 变成 h 的双亲，e 的右孩子则变成 h 的左孩子。若每次插入结点后都进行了平衡化处理，最多只需作一次旋转。满足平衡规则（某结点左右子树高度之差的绝对值不超过 1）的二叉树称为 AVL 树（这一术语最初是由 G.M.Adelson-Velskii 和 E.M.Landis 提出来的[9]）。

插入 d 对 e 作右旋转如图 3-2 所示：

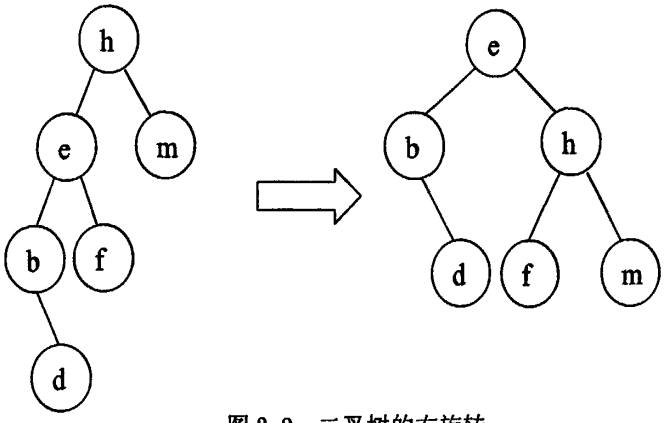

图 3-2　二叉树的右旋转

3.2.2 红黑树

红黑树类似于 AVL 树，主要用于 Linux 内存管理。红黑树的本质是二叉平衡树，其中每个结点都有红或黑的颜色属性。红黑树的规则如下：

（1）任何结点非红即黑；

（2）如果一个结点是红的，那么它的所有孩子都是黑的；

（3）所有叶结点都是黑的；

（4）从根向叶结点遍历时，每条遍历路径包含相同数量的黑色结点。

AVL 树和红黑树的查找时间复杂度都是 O log(n)，而根据插入（已排序的、未排序的）和查找数据的不同，每次都能得出不同的具体数值。前文已经提到，许多其他数据结构和相关的查找算法也应用于计算机科学中。

3.3 小结

本节的主要工作是通过分析理解 Linux 中常用数据结构及其算法，达到对内核深入透彻理解的目的。通过本节中对链表和树结构进行的分析，能更好地把握数据结构在相关算法实现中的重要作用，对后文中改进现有 Linux 内核起到了较好的理论支撑作用。

第四章 Linux 的进程结构及其相关组织分析

4.1 相关概念简述

4.1.1Linux 进程的四个要素

一般来说 Linux 系统的进程都具备下列诸要素：

（1）有一段程序供其执行。这段程序不一定是某个进程所专有，其他进程可以共享。

（2）有进程专用的内核空间堆栈。

（3）在内核中有一个 task_struct 数据结构，即 “进程控制块 PCB”（process control block）。这个数据结构是操作系统识别进程的唯一标识，进程才能成为内核调度的一个基本单位接受内核的调度[10]。同时，这个结构还记录着进程所占用的各项资源信息。

（4）有独立的存储空间，这意味着拥有专有的用户空间；同时还意味着除了内核空间堆栈外还有其专用的用户空间堆栈。但内核空间是不能独立的，任何进程都不可能直接（不通过系统调用）改变内核空间的内容（除其本身的内核空间堆栈以外）。

这四点都是必要条件，缺了任何一条都不能成为“进程”。如果只具备了前三条而缺第四条，就称为“线程”。如果完全没有用户空间，就称为“内核线程”（kernel thread）[11]；而如果共享用户空间则称为“用户线程”。二者往往都简称“线程”。事实上在 Linux 系统中，进程和线程的区分并不十分严格，许多进程在“诞生”之初都与其父进程共用同一个存储空间，所以严格说来还是线程；但是子进程可以建立其自己的存储空间，并与父进程分离，成为真正意义上的进程[4]。

4.1.2 task_struct 结构研究

task_struct 结构是向系统表明进程存在的唯一凭证，它包含了进程的全部信息。同时，也是进程为实现操作而取得必要资源的唯一途径。下面列出了 task_struct 结构的全部源码，在源码后面有对 task_struct 结构（参见

include\linux\sched.h[1]）各数据项的分类解释：

```
    struct task_struct {
        volatile long state;              /* 进程的运行状态*/
        struct thread_info *thread_info; /* 当前进程的一些运行环境信息*/
        atomic_t usage;
        unsigned long flags;              /*进程的状态标志*/
        unsigned long ptrace;
        int lock_depth;                   /* BKL lock depth */
    #ifdef CONFIG_SMP
    #ifdef __ARCH_WANT_UNLOCKED_CTXSW
        int oncpu;                        /*进程运行所在的cpu编号*/
    #endif
  #endif
    int load_weight;      /* 进程所在CPU负载平衡的信息，用于多处理器 */
    int prio, static_prio, normal_prio; /*进程优先极标示*/
    struct list_head  run_list;           /*进程队列链表*/
    struct prio_array *array;             /*进程所在的运行队列*/
    unsigned short ioprio;
  #ifdef CONFIG_BLK_DEV_IO_TRACE
    unsigned int btrace_seq;
  #endif
    unsigned long sleep_avg;              /* 进程的平均睡眠时间*/
    unsigned long long timestamp, last_ran; /*进程的运行的时间戳*/
    unsigned long long sched_time;        /* 运行锁定调度的时间值*/
    enum sleep_type sleep_type;           /*进程睡眠的状态*/
    unsigned long policy;
    cpumask_t cpus_allowed;               /*CPU的屏蔽标示*/
    unsigned int time_slice, first_time_slice; /*进程运行的时间片*/
  #if defined(CONFIG_SCHEDSTATS) || defined(CONFIG_TASK_DELAY_ACCT)
    struct sched_info sched_info;
  #endif
```

```
        struct list_head tasks;
        /* ptrace_list/ptrace_children forms the list of my children
         * that were stolen by a ptracer. */
        struct list_head ptrace_children;
        struct list_head ptrace_list;
        struct mm_struct *mm, *active_mm; /*进程的内存地址映射信息*/
    /* task state */
        struct linux_binfmt *binfmt; /*进程执行体的文件格式*/
        long exit_state; /*进程退出状态*/
        int exit_code, exit_signal;
    int pdeath_signal;  /*  The signal sent when the parent dies  */
        unsigned long personality;
        unsigned did_exec:1;
        pid_t pid;/*进程标示符*/
        pid_t tgid;
    #ifdef CONFIG_CC_STACKPROTECTOR
        /* Canary value for the -fstack-protector gcc feature */
        unsigned long stack_canary;
    #endif
    /* pointers to (original) parent process, youngest child, younger
sibling,older sibling, respectively.  (p->father can be replaced with
p->parent->pid)*/
    struct task_struct *real_parent; /* real parent process (when being
debugged) */
    struct task_struct *parent;     /* parent process */
        /*children/sibling forms the list of my children plus the
            tasks I'm ptracing. */
    struct list_head children;  /* list of my children */
    struct list_head sibling;   /* linkage in my parent's children list*/
struct task_struct *group_leader;  /* threadgroup leader */
         /* PID/PID hash table linkage. */
    struct pid_link pids[PIDTYPE_MAX];
```

```
    struct list_head thread_group;
    struct completion *vfork_done;      /* for vfork() */
    int __user *set_child_tid;          /* CLONE_CHILD_SETTID */
       int __user *clear_child_tid;     /* CLONE_CHILD_CLEARTID */
       unsigned long rt_priority;       /*实时进程优先级*/
       cputime_t utime, stime;
       unsigned long nvcsw, nivcsw;     /* 上下问切换次数*/
       struct timespec start_time;
    /* mm fault and swap info: this can arguably be seen as either
mm-specific or thread-specific */
       unsigned long min_flt, maj_flt;
       cputime_t it_prof_expires, it_virt_expires;
       unsigned long long it_sched_expires;
       struct list_head cpu_timers[3];/* process credentials */
       uid_t uid,euid,suid,fsuid;
       gid_t gid,egid,sgid,fsgid;
       struct group_info *group_info;
       kernel_cap_t   cap_effective, cap_inheritable, cap_permitted;
       unsigned keep_capabilities:1;
       struct user_struct *user;
    #ifdef CONFIG_KEYS
    struct key *request_key_auth;/*assumed request_key authority */
    struct key *thread_keyring;  /*keyring private to this thread */
    unsigned char jit_keyring; /*default keyring to attach requested keys
to*/
    #endif
          /* fpu_counter contains the number of consecutive context
*switches that the FPU is used. If this is over a threshold, *the lazy fpu
saving becomes unlazy to save the trap. This is *an unsigned char so that
after 256 times the counter wraps and *the behavior turns lazy again; this
to deal with bursty apps *that only use FPU for a short time */
    unsigned char fpu_counter;
```

```
int oomkilladj; /* OOM kill score adjustment (bit shift).*/
char comm[TASK_COMM_LEN]; /* executable name excluding path
                - access with [gs]et_task_comm (which lock
                  it with task_lock())
                - initialized normally by flush_old_exec */
/* file system info */
int link_count, total_link_count;
#ifdef CONFIG_SYSVIPC
/* ipc stuff */
   struct sysv_sem sysvsem;
#endif
/* CPU-specific state of this task */
struct thread_struct thread;
/* filesystem information */
struct fs_struct *fs;/*open file information */
   struct files_struct *files; /* namespaces */
   struct nsproxy *nsproxy;  /* signal handlers */
   struct signal_struct *signal;
   struct sighand_struct *sighand; /*信号处理结构体*/
   sigset_t blocked, real_blocked; /*信号屏蔽*/
   sigset_t saved_sigmask;         /*To be restored with
                                    TIF_RESTORE_SIGMASK */
   struct sigpending pending;     /*进程挂起的信号*/
   unsigned long sas_ss_sp;
   size_t sas_ss_size;
   int (*notifier)(void *priv);
   void *notifier_data;
   sigset_t *notifier_mask;
   void *security;
   struct audit_context *audit_context;
   seccomp_t seccomp;
/* Thread group tracking */
```

```
        u32 parent_exec_id;
        u32 self_exec_id;
    /* Protection of (de-)allocation: mm, files, fs, tty, keyrings */
        spinlock_t alloc_lock;
        /* Protection of the PI data structures: */
        spinlock_t pi_lock;
    #ifdef CONFIG_RT_MUTEXES
        /* PI waiters blocked on a rt_mutex held by this task */
        struct plist_head pi_waiters;
        /* Deadlock detection and priority inheritance handling */
        struct rt_mutex_waiter *pi_blocked_on;
    #endif
    #ifdef CONFIG_DEBUG_MUTEXES
        /* mutex deadlock detection */
        struct mutex_waiter *blocked_on;
    #endif
    #ifdef CONFIG_TRACE_IRQFLAGS
        unsigned int irq_events;
        int hardirqs_enabled;
        unsigned long hardirq_enable_ip;
        unsigned int hardirq_enable_event;
        unsigned long hardirq_disable_ip;
        unsigned int hardirq_disable_event;
        int softirqs_enabled;
        unsigned long softirq_disable_ip;
        unsigned int softirq_disable_event;
        unsigned long softirq_enable_ip;
        unsigned int softirq_enable_event;
        int hardirq_context;
        int softirq_context;
#endif
#ifdef CONFIG_LOCKDEP
```

```
# define MAX_LOCK_DEPTH 30UL
  u64 curr_chain_key;
  int lockdep_depth;
  struct held_lock held_locks[MAX_LOCK_DEPTH];
  unsigned int lockdep_recursion;
#endif
 /* journalling filesystem info */
void *journal_info;
 /* VM state */
  struct reclaim_state *reclaim_state;
      struct backing_dev_info *backing_dev_info;
      struct io_context *io_context;
      unsigned long ptrace_message;
      siginfo_t *last_siginfo;  /* For ptrace use. */
   /*current io wait handle: wait queue entry to use for io waits
   If this thread is processing aio, this points at the waitqueue
   inside the currently handled kiocb. It may be NULL (i.e. *default to
a stack based synchronous wait) if its doing sync *IO. */
      wait_queue_t *io_wait;
   /* i/o counters(bytes read/written, #syscalls */
      u64 rchar, wchar, syscr, syscw;
      struct task_io_accounting ioac;
   #if defined(CONFIG_TASK_XACCT)
      u64 acct_rss_meml;    /* accumulated rss usage */
      u64 acct_vm_meml; /* accumulated virtual memory usage */
      cputime_t acct_stimexpd;/* stime since last update */
   #endif
   #ifdef CONFIG_NUMA
      struct mempolicy *mempolicy;
      short il_next;
   #endif
   #ifdef CONFIG_CPUSETS
```

```
    struct cpuset *cpuset;
    nodemask_t mems_allowed;
    int cpuset_mems_generation;
    int cpuset_mem_spread_rotor;
#endif
    struct robust_list_head __user *robust_list;
#ifdef CONFIG_COMPAT
    struct compat_robust_list_head __user *compat_robust_list;
#endif
    struct list_head pi_state_list;
    struct futex_pi_state *pi_state_cache;
    atomic_t fs_excl;    /* holding fs exclusive resources */
    struct rcu_head rcu;
     /* cache last used pipe for splice */
    struct pipe_inode_info *splice_pipe;
#ifdef CONFIG_TASK_DELAY_ACCT
    struct task_delay_info *delays;
#endif
#ifdef CONFIG_FAULT_INJECTION
    int make_it_fail;
#endif
}
```

下面部分将阐述 task_struct 中相关的域。

（一）与进程属性相关的域

进程的属性分类是一种全方位的分类，我们为进程属性定义了进程的状态和标识，图 4-1 说明了 task_sturct 中进程的属性相关的域[2]：

(1)　volatile long state:

表示进程的当前状态。Linux 系统中进程有 6 种状态。

• TASK_RUNNING：正在运行或在就绪队列 run_queue 中准备运行的进程，实际参与进程调度。

• TASK_INTERRUPTIBLE：处于等待队列中的进程，待资源有效时唤醒，也可由其它进程通过信号(signal)或定时中断唤醒后进入就绪队列 run_queue。

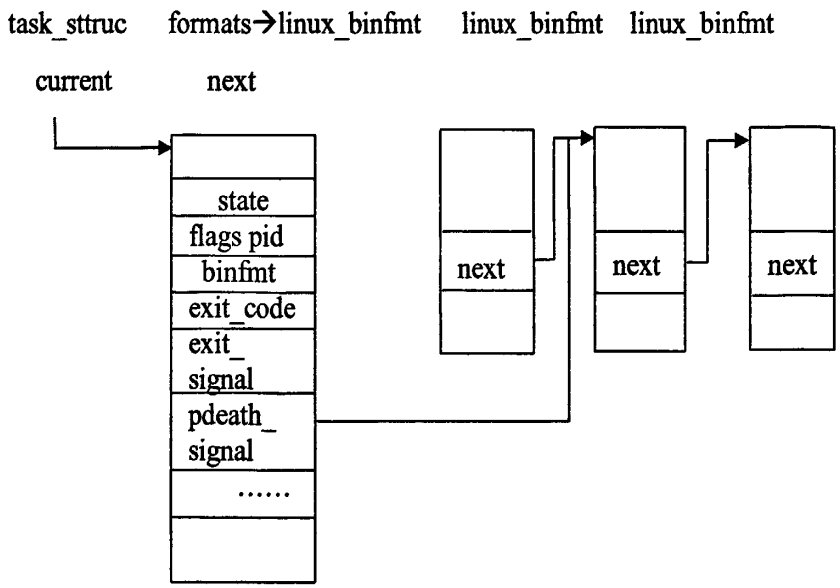

图 4-1 进程属性域

• TASK_UNINTERRUPTIBLE：处于等待队列中的进程，待资源有效时唤醒，不可由其它进程通过信号(signal)或定时中断唤醒。

• TASK_ZOMBIE：表示进程结束但尚未消亡的一种状态(僵死状态)。此时，进程已经结束运行且释放大部分资源，但尚未释放进程控制块。

• TASK_STOPPED：进程被暂停，通过其它进程的信号才能唤醒。导致这种状态的原因有两个，或者是对收到 SIGSTOP、SIGSTP、SIGTTIN 或 SIGTTOU 信号的反应，或者是受其它进程的 ptrace 系统调用的控制而暂时将 CPU 交给控制进程[12]。

• TASK_SWAPPING：进程页面被交换出内存的进程。

(2) unsigned long flags：

包含进程标志信息：

• PF_ALIGNWARN 打印“对齐”警告信息；
• PF_PTRACED 被 ptrace 系统调用监控；
• PF_TRACESYS 正在跟踪；
• PF_FORKNOEXEC 进程刚创建，但还没执行；
• PF_SUPERPRIV 超级用户特权；
• PF_DUMPCORE dumped core；
• PF_SIGNALED 进程被信号(signal)杀出；
• PF_STARTING 进程正被创建；

- PF_EXITING　　进程开始关闭；
- PF_USEDFPU　　该进程使用 FPU(SMP only)；
- PF_DTRACE　　delayed trace (used on m68k)。

(3)　pit_t pid：

Linux 中，每个进程都有唯一的进程标识符 pid(process identifier)，pid 位于 task_struct 结构体中，类型为 pid_t，其本质就是整型数据，因此，pid 默认的最大值是 32767（短整型数的最大可能值）[13]。

(4)　binfmt：

Linux 支持多种可执行文件格式，每种可执行文件格式都定义了一种数据结构，指明程序代码如何被载入内存。

(5)　exit_signal 和 exit_code：

exit_code 与 exit_signal 分别存放任务的退出值和终止信号(如果使用了该域的话)，这是将子进程的退出值传给其他进程的方式。

(6)　pdeath_signal：

pdeath_signal 是进程在消亡时设置的信号。

(7)　comm：

通常通过在命令行调用一个可执行程序来创建进程，当在命令行调用可执行程序时，comm 保存其名称。

(8) ptrace：

当进程因执行检测任务而调用 ptrace（）系统调用时设置此值。

当进程执行调度时包含了以下信息：

(二) 与进程调度相关的域

进程在运行时如同拥有自己独立的 cpu，但事实上是多个进程共享一个 cpu，为了在运行的进程之间进行切换，每个进程均与调度程序有着密切的联系。然而要理解上述的某些域，必须要理解调度的基本概念。当多个进程都已就绪等待运行时，由调度程序决定谁先运行和运行多长时间。调度程序通过给每个进程分配一个时间片（timeslice）和优先级（priority）来实现公平、高效地调度。时间片定义了一个进程被切换到另外一个进程之间允许运行的时间长度。进程的优先级则是一个数值，它代表进程在调度时被选择的相对顺序，相对其他就绪进程而言，优先级越高的进程会优先被调度运行。图 4-2 所使的域保存了调度所需的值。

(1)　prio：优先级，相当于 2.4 中 goodness() 的计算结果，在 0~MAX_PRIO-1

之间取值(MAX_PRIO 定义为 140)，其中 0~MAX_RT_PRIO-1（MAX_RT_PRIO 定义为 100)[14]属于实时进程范围，MAX_RT_PRIO~MX_PRIO-1 属于非实时进程。数值越大，表示进程优先级越小。

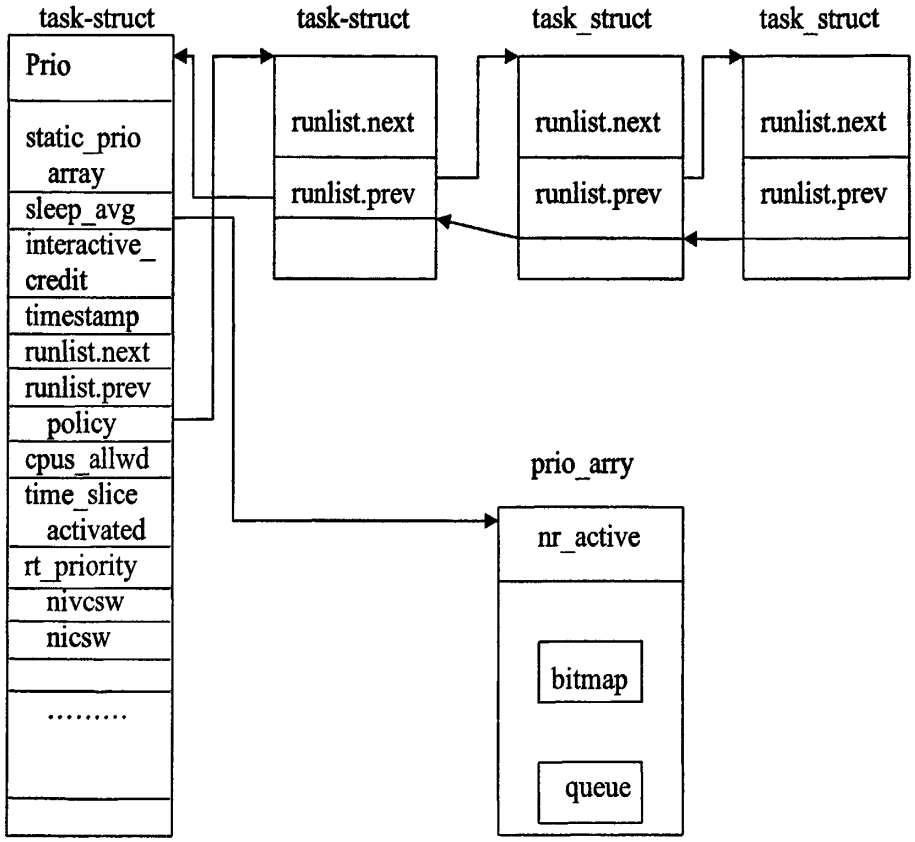

图 4-2 与调度相关的域

Linux2.6 中，动态优先级不再统一在调度器中计算和比较，而是独立计算，并存储在进程的 task_struct 中，再通过 prioty_array 结构自动排序。prio 的计算和很多因素相关。当进程被执行且时间片用完时，其动态优先级将在睡眠时刻被更新，其值 prio 与下面将说明的 static_prio 字段的值有关，prio 的值是在 static_prio 的基础上加/减 5，具体依进程的历史而变，如果睡眠了很长时间的话，那么它将得到+5 的时间延长；反之，如果运行了相当长的时间且用完了其时间片，那么它将得到-5 的时间缩减。[5]

(2) static_prio:

static_prio 与 nice 的值相关，其默认值为 MAX_PRIO-20，可调用 nice () 来修改进程的静态优先级。进程初始时间片的大小仅决定于进程的静态优先级，这一点不论是实时进程还是非实时进程都一样，所不同的是实时进程的

static_prio 不参与优先级计算。

(3) run_list：

run_list 是指运行队列，它是一个包含所有就绪状态进程的队列，关于其具体内容将在下文调度算法中给以具体分析。

(4) arry：

arry 是指运行队列的优先级数组，见下文详析。

(5) sleep_avg：

用于计算任务的有效优先级，它是任务睡眠过程中时钟节拍平均数。进程的平均等待时间（以 nanosecond 为单位），在 0 到 NS_MAX_SLEEP_AVG 之间取值，初值为 0，相当于进程等待时间与运行时间的差值。sleep_avg 所代表的含义比较丰富，既可用于评价该进程的"交互程度"，又可用于表示该进程需要运行的紧迫性。这个值是确定动态优先级的关键因子，sleep_avg 越大，计算出来的进程动态优先级也越高（数值越小）[15]。

(6) time_stamp：

进程发生调度事件的时间（单位是 nanosecond，见下）。包括以下几类：

- 被唤醒的时间（在 activate_task() 中设置）；
- 被切换下来的时间（schedule()）；
- 被切换上去的时间（schedule()）；
- 负载平衡相关的赋值（见"调度器相关的负载平衡"）。

从这个值与当前时间的差值中可以分别获得"在就绪队列中等待运行的时长"、"实际运行时长"等与优先级计算相关的信息. time_stamp（时间戳）用于当任务睡眠或放弃处理器时计算 sleep_avg 的值。

(7) interactive_credit ：

这个变量代表进程的"交互程度"，在 -CREDIT_LIMIT 到 CREDIT_LIMIT+1 之间取值。进程被创建出来时，初值为 0，而后根据不同的条件加 1 减 1，一旦超过 CREDIT_LIMIT（只可能等于 CREDIT_LIMIT+1），它就不会再降下来，表示进程已经通过了"交互式"测试，被认为是交互式进程了。interactive_credit 用于与 sleep_avg、activated 共同计算 sleep_avg 的指针。[16]

(8) policy：

该进程的进程调度策略。Linux 系统提供三种调度策略，根据任务的属性，调度策略有：

- SCHED_OTHER 0 非实时进程，基于优先级的轮转法(round robin)。

- SCHED_FIFO　　1　实时进程，先进先出算法。
- SCHED_RR　　2　实时进程，基于优先权的轮转法。

第一种是普通分时进程的调度策略，后面两种是软实时进程调度策略。

(9) time_slice (时间片):

time_slice 定义任务每次被调度时允许占用 CPU 的最长时间。初始值是根据进程的静态优先级来计算的。进程的 time_slice 值代表进程的运行时间片剩余数量，在进程创建时与父进程平分时间片，在运行过程中递减，一旦归 0，则按 static_prio 值重新赋予上述基准值，[17]并请求调度。时间片的递减和重置在时钟中断中进行 (sched_tick())，除此之外，time_slice 值的变化主要在创建进程和进程退出过程中。

(10) rt_priority:

rt_priority 时实时进程的优先级，是一个静态值，只能通过 shedule () 来更新，支持实时任务时必须用到该域。

(11) cpus_allowed:

cpus_allowed 指明由哪个 cpu 来处理该任务，可以通过这种方式在多处理系统上指定某以特定的任务在哪个 cpu 上运行。

(三) 其他一些重要的域

(1) mm 和 active_mm

在 linux 中，采用按需分页的策略解决进程的内存使用需求。task_struct 的数据成员 mm 指向关于存储管理的 mm_struct 结构。其中包含了一个虚存队列 mmap，指向由若干 vm_area_struct 描述的虚存块。[18]同时，为了提高访存速度，mm 中的 mmap_avl 维护了一个 AVL 树。在该树中，所有的 vm_area_struct 虚存块均由左指针指向相邻的低址虚存块，右指针指向相邻的高址虚存块，见图 4-3。其结构定义在 include/linux/sched.h 中。

(2) struct fs_struct *fs 和 struct files_struct *files

fs 保存了进程本身与 VFS 的关系消息，其中 root 指向根目录结点，pwd 指向当前目录结点，umask 给出新建文件的访问模式(可由系统调用 umask 更改)，count 是 Linux 保留的属性，如图 4-4 所示。结构定义在 include/linux/sched.h 中[1]。

files 包含了进程当前所打开的文件(struct file *fd[NR_OPEN])。在 Linux 中，一个进程最多只能同时打开 NR_OPEN 个文件。同时前三项分别预先设置为标

准输入、标准输出和出错消息输出文件。进程与文件的对应关系见图 4.4。

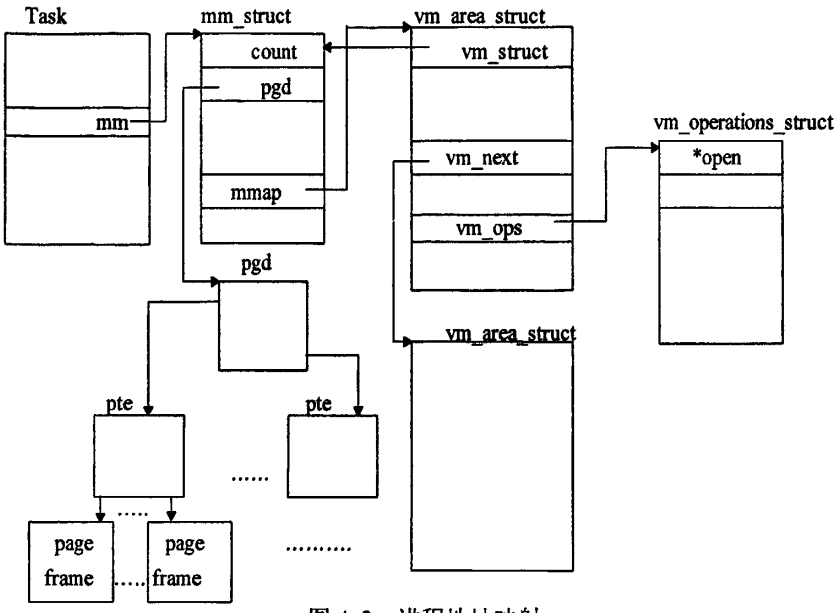

图 4-3 进程地址映射

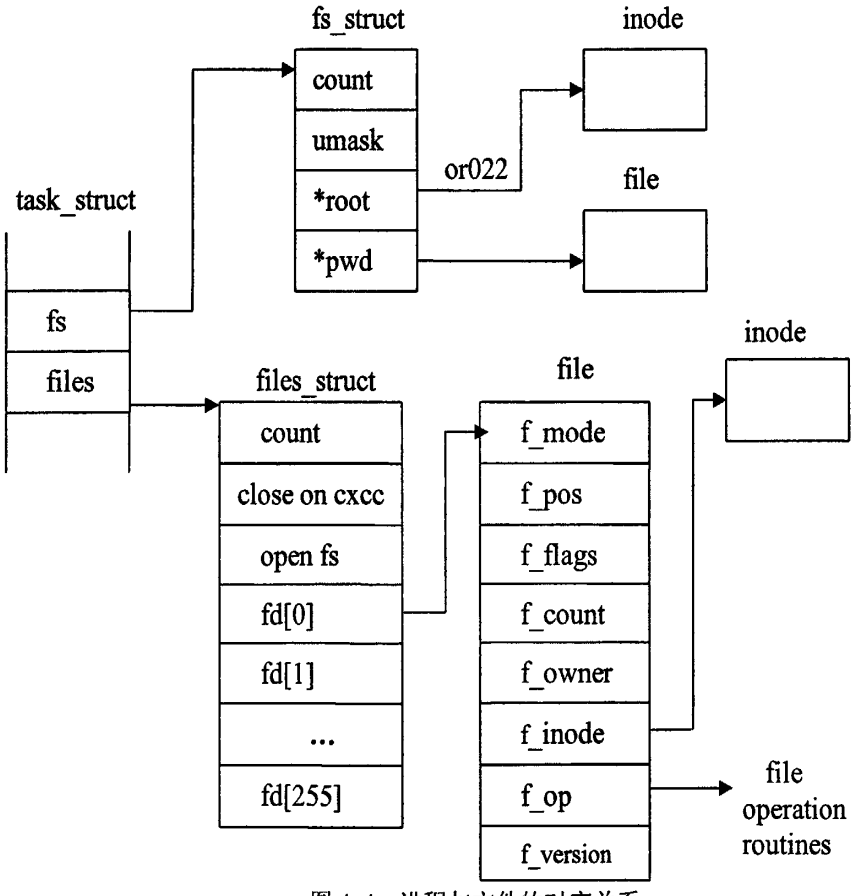

图 4-4 进程与文件的对应关系

由于 Linux 的进程数据结构是很庞大的，不仅包含进程所用的内存信息，与其相关文件的信息，还有一些进程对信号和中断的处理信息，本论文没有完全列出分析其所有的数据成员，只介绍了其一小部份。进程是操作系统中一个很重要的资源，在 Linux 中，调度以进程为基本单位，这也是和 windows 的一个区别。图 4-5 显示了进程的状态转换关系。

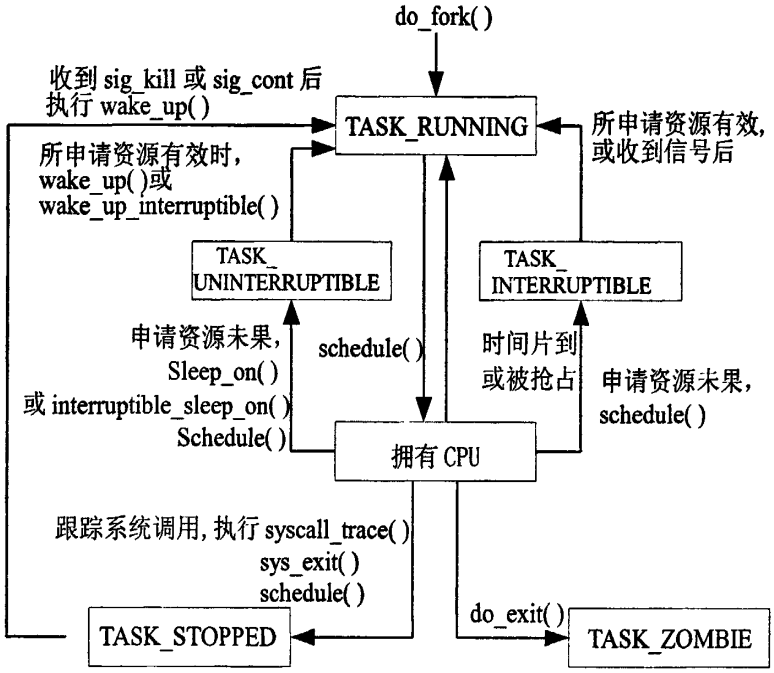

图 4-5 进程状态转换

论文本节主要对以进程为中心的状态和转换进行了分析，并对其中数据结构及其中重要的域定义进行了研究。

4.2 进程组织中数据结构的应用

4.2.1 等待队列

论文上节研究了 TASK_RUNNING 状态和 TASK_INTERRUPTIBLE 或 TASK_UNINTERRUPTIBLE 状态之间的转换。本节研究该状态转换的另一个结构。当进程等待一个外部事件发生时，就把它从运行队列删除并放到等待队列上。等待队列是 wait_queue_t 结构的双向链表。wait_queue_t 结构的设立是为了记录等待进程需要的所有信息。等待一个特定外部事件的所有进程被放到一个等待队列

中。当某个等待队列上的进程被唤醒时，该进程核实它所等待的条件，然后或者继续睡眠，或者从等待队列删除自己并把自己设置回 TASK_RUNNING。当父进程需要获知它所创建子进程的状态时，ys_wait4()系统调用对等待队列执行操作。等待外部事件的进程(因此不再处于运行队列中)可能处于 TASK_INTERRUOTIBLE 状态，也可能处于 TASK_UNINTERRUPTIBLE 状态[19]。

等待队列是wait_queue_t结构的双向链表。wait_queue_t结构中有指向阻塞进程 task 结构的指针。每个链表以 wait_queue_head_t 结构打头，该结构表示链表的头部，并存放有 wait_queue_t 的自旋锁，自旋锁可以防止 wait_queue_t 的额外竞争条件。

图 4-6 说明了等待队列的实现。

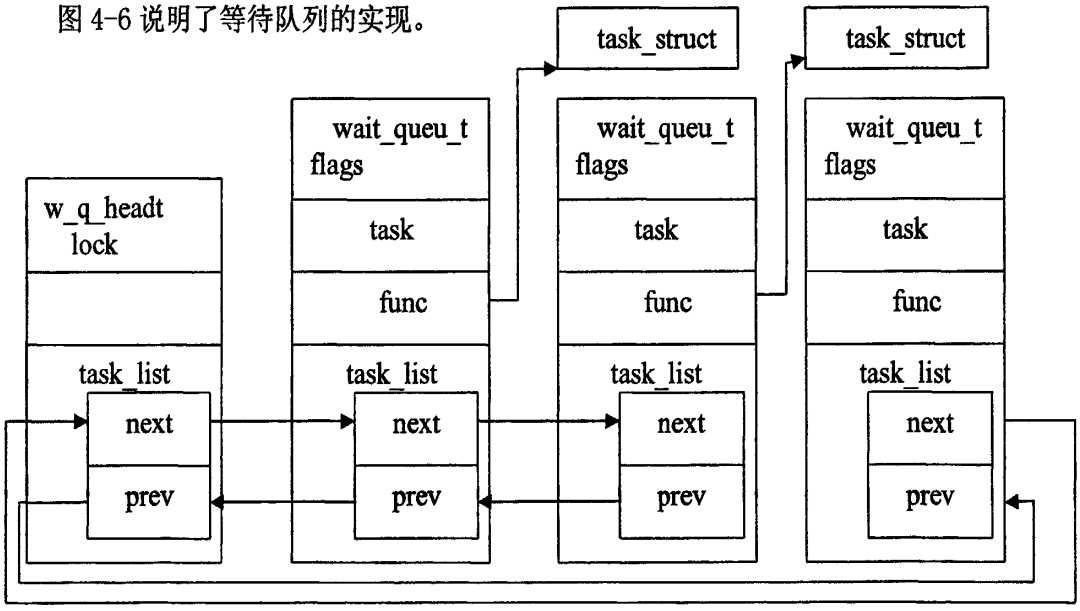

图 4-6 等待队列

以下是 wait_queue_t 和 wait_queue_head_t 结构的分析。

```
typedef struct _wait_queue wait_queue_t;
struct _wait_queue {
      unsingned  int  flags;
      #define WQ_FLAG_EXCLUSIVE 0X01
       struct task_struct  *task;
       wait_queue_func_t func;
       struct  list_head task_list;
```

```
};
struct  _wait_queue_head {
        spinlock_t lock;
        struct list_head task_list;
    };
typedefine struct  _wait_queue_head  wait_queue_head_t;
```

wait_queue_t 结构由下列域组成：

(1)flags:存放值 WQ_FLAG_EXCLUSIVE(值被设置为 1)或 ~WQ_FLAG_EXCLUSIVE(将会为 0)。WQ_FLAG_EXCLUSIVE 标志表示该进程属性为独占式进程。[20]

(2) task：一个指针，指向加入在等待队列上的进程其进程描述符。

(3) func：存放函数的一个结构，这个函数作用是唤醒等待队列上的进程。这个域的默认值为 wait_queue_func_t，定义如下：

```
typedef int (*wait_queue_func_t)(wait_queue_t *wait ,
                unsigned mode, int sync);
```

在此，wait 是指向等待队列的指针，而 mode 是 TASK_INTERRUOTIBLE 或者 TASK_UNINTERRUPTIBLE，sync 表示唤醒是否同步。

(4) task_list：这个结构包含了两个指针，分别指向等待队列中的前一个元素和后一个元素。

(5) _wait_queue_head：该结构是等待队列链表的表头，由下列域组成：

① Lock：每个链表有一个锁，这使向等待队列添加或删除数据项操作能够同步。

② task_list：用于指向等待队列中第一个元素和最后一个元素。

通常,进程让自己睡眠的途径包括对某一个 wait_event *宏的调用，或者通过执行下列步骤来实现：

① 通过声明等待队列，进程利用 DECLARE_WAITQUEUE_HEAD 继续睡眠。

② 该进程利用 add_wait_queue()或者 add_wait_queue_exclusive()把自己加入到等待队列。

③ 把进程状态变为 TASK_INTERRUPTIBLE 或 TASK_UNINTERRUPTIBLE。

④ 如果外部事件还没有发生，检测外部事件并调用 schedule()。

⑤ 当外部事件发生后，把进程状态设置为 TASK_RUNNING。

⑥ 该进程通过调用 remove_wait_queue()从等待队列删除自己。

下列代码执行对于 wait_queue_head_t 结构的初始化：

```
void init_waitqueue_head(wait_queue_head_t *q)
 {      spin_lock_init(&q->lock);
       INIT_LIST_HEAD(&q->task_list);
 }
```

调用一组 wake_up 宏中的其中一个就可以唤醒进程。这些宏唤醒某个等待队列上的所有进程，它把进程置为 TASK_RUNNING 状态，并放回运行队列。[21]

而调用函数 add_wait_queue()所完成的工作如下：

① 执行在等待队列中插入与删除操作。有两个不同的函数可以向等待队列添加睡眠进程，即函数 add_wait_queue()和 add_wait_queue_exclusive()。这两个函数添加两种类型的的睡眠进程。非独占式等待进程是指等待条件返回的进程，这个条件不能被其他等待进程共享。独占式等待进程是指等待一个其他进程可能正在等待的条件,这可能会产生一个竞争条件。

add_wait_queue()函数在等待队列中插入一个非独占式的进程。非独占式进程是指那些在任何条件下，当等待的事件完成时就被内核唤醒的进程。这个函数设置等待队列结构的 flags 域，0表示睡眠进程，同时设置等待队列锁，以避免访问同一个等待队列的中断产生竞争条件，之后把 wait_queue_head_t 结构加入到等待队列链表，并且从等待队列恢复锁，使得其他进程可以使用。

```
void add_wait_queue(wait_queue_head_t *q,wait_queue_t *wait)
{
   unsigned  long flags;
   wait->flags&=~WQ_FLAG_EXCLUSIVE;
   spin_lock_irqsave(&q->lock,flags);
   _add_wait_queue(q,wait);
Spin_unlock_irqrestore(&q->lock,flags);
}
```

对_add_wait_queue()的定义如下：

```
static inline void _add_wait_queue(wait_queue_head_t *head,
wait_queue_t *new)
{ list_add(&new->task_list, &head->task_list);
}
```

add_wait_queue_exclusive()函数向等待队列插入一个独占式进程。该函数将等待进程结构的 flags 域设置为 1，并且用与 add_wait_queue()相同的方式进行独占操作，但有一点例外，它执行插入进程操作时在队列末端进行。这就意味着在一个特定的等待队列里，非独占式进程在前面，独占式进程在末尾[22]。

```
void add_wait_queue_exclusive(wait_queue_head_t *q,wait_queue_t *wait)
{
    unsigned long flags;
    wait->flags|=WQ_FLAG_EXCLUSIVE;
    spin_lock_irqsave(&q->lock,flags);
     _add_wait_queue_tail(q,wait);
    spin_unlock_irqrestore(&q->lock,flags);
}
```

对于从队列中删除任务的代码如下：

```
void remove_wait_queue(wait_queue_head_t *q, wait_queue_t *wait)
{
    unsigned long flags;
    spin_lock_irqsave(&q->lock, flags);
    _remove_wait_queue(q, wait);
    spin_unlock_irqrestore(&q->lock, flags);
}
```

而_remove_wait_queue()的定义如下：

```
static inline void _remove_wait_queue(wait_queue_head_t *head,
wait_queue_t *old)
{
    list_del(&old->task_list);
```

}

4.2.2 等待事件

Linux 系统中，对于某个进程需要等到某个事件时,可通过 wait_event*()接口来完成。wait_event*() 接口包括了 wait_event()，wait_interruptible()和 want_event_interruptible_timeout()。[23]

图 4-7 显示了这些函数基本的调用轨迹。

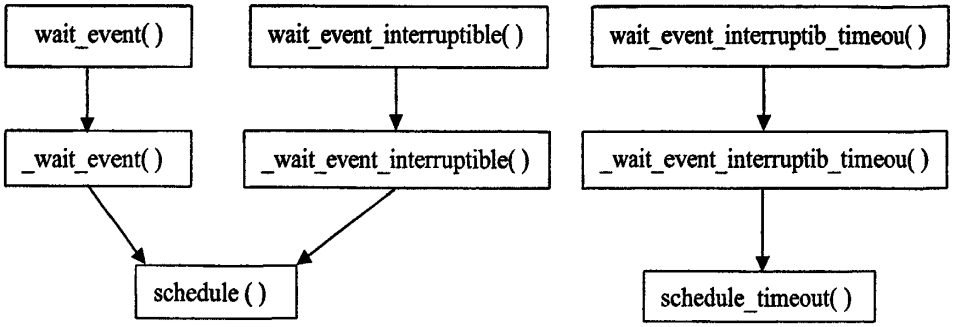

图 4-7 函数调用关系

wait_event()接口是对_wait_event( )的调用以无限循环的方式进行封装，仅当所等待的条件返回时，循环才被打破，wait_event_interruptible_timeout()传递的三个参数是 int 类型的返回值，它用于传递超时时间。

其相关定义如下：

```
#define wait_event(wq,condition)
   do{   if(condition) break;
         _wait_event(wq,condition);
      }while(0)
```

_wait_event()接口完成所有围绕进程状态改变和描述符操作的工作。

```
#define  _wait_event(wq,condition)
  do{ wait_queue_t  _wait;
      init_waitqueue_enttry(&_wait,current);
      add_wait_queue(&wq,&_wait);
      for( ;  ;  ){
          set_current_state(TASK_INTERRUPTIBLE);
```

```
if(condition)    break;
schedule();
}
Current->state=TASK_RUNNING;
Remove_wait_queue(&wq,&_wait);
}while(0)
```

在程序中首先为当前进程初始化等待队列描述符，并把该描述符添加到传递过来的等待队列中。然后设置一个无限循环，仅当条件成立时，才中断循环。进程在这个条件上阻塞，阻塞之前利用 set_current_state 宏把其状态设置为 TASK_INTERRUPTIBLE，这个宏引用了指向当前进程的指针，因此不必把进程信息传给它。一旦进程阻塞，它依靠对 schedule()的调用为别的进程让出 CPU。当条件成立时，进程被设置为 TASK_RUNNING(调度程序把它放入运行队列)。最后从等待队列删除这一元素。在删除该元素前，remove_wait_queue()函数为等待队列上锁，而在返回前开锁。

而对于 wait_event_interruptible()函数,它的作用是将进程的状态设置为 TASK_UNTERRUPTIBLE 并对当前进程等待一个 signal_pending 调用。

在 Linux 系统运行中，进程必须被唤醒来检查它的条件是否成立。进程可以让自己睡眠，但是不能唤醒自己。有一系列的函数以及宏可以用来唤醒处于等待队列中的进程。

常用的是_wake_up()。以下代码对其进行了详细解析。

```
void  fastcall  _wale_up(wait_queue_head_t *q, unsigned int mode,
int nr_exclusive)
 {   unsigned long flags;
     spin_lock_irqsave(&q->lock, flags);
     _wake_up_common(q,mode, nr_exclusive,0);
    spin_lock_irqrestore(&q->lock, flags);
 }
```

其中 q 是一个指向等待队列的指针，mode 是表示将要唤醒的进程的类型(由进程状态来标识)；nr_exclusive，表示它是独占式唤醒还是非独占式唤醒。独占

式唤醒(nr_exclusive=0 时)将唤醒等待队列中的所有进程(独占式和非独占式进程)，而非独占式唤醒只唤醒一个独占式进程和所有非独占式进程。[24]

在调用_wake_up_common()之前要设置等待队列的锁，以确保竞争条件不会发生。

wakeup 函数的大部分工作由函数_wake_up_common()完成：

```
static void _wake_up_common(wait_queue_head_t *q, unsigned int modeint
nr_exclusive, int sunc)
{
   struct list_head *tmp , *next;
   list_for_each_safe(tmp,next,&q->task_list)
   {
     wait_queue_t *curr;
     unsigned flags;
     curr=list_entry(tmp,wait_queue_t,task_list);
     flags=curr->flags;
     if(curr->func(currmode,sync)&&(flags&WQ_FLAG_EXCLUSIVE)&&!_nr_exc
lusive)
     break;
     }
}
```

其中的 sync 表示唤醒是否同步。该函数扫描等待队列中的每一项，调用 wait_queue_t 的 func 域[24]。在默认条件下，它调用下面所示的 default_wake_function()函数，如下面代码所示：

```
 int default_wake_function(wait_queue_t *curr,unsigned mode,int sync)
{ task_t *p=curr->task;
 return try_to_wake_up(p, mode, sync);
}
```

该函数在 wait_queue_t 结构所指向的 task 上调用函数 try_to_wake_up()，它完成唤醒进程大部分工作，包括把进程插入至运行队列。

在_wake_up_common()中,如果被唤醒的进程是第一个独占式的进程，则终止循环。若知道所有的独占式进程都排在等待队列的尾部，这为操作提供了方便,因为在遇到等待队列中的第一个独占式进程后，就可以判定剩余的所有进程也将是独占式的，因此不应该唤醒它们，而应终止循环(即不再遍历链表)，这样即可提高 Linux 的工作效率。

4.3 小结

本章深入分析了操作系统中最重要的基本单位进程的结构，对进程运行中各种状态及转换、各大功能模块实现所用到的数据结构做了透析，在此研究的基础上，将在论文下一章中重点针对 Linux 调度功能及调度实时性进行研究。

第五章 Linux 的进程调度研究

5.1 调度模块及数据结构的应用研究

本节将通过进程的调度模块来深入分析数据结构在操作系统中的应用，同时针对不同数据结构作算法的改进。主要是通过对 linux2.4 和 2.6 内核版本的不同（其调度模块所选用的数据结构有所差异）来分析其在调度算法上的改进。

一个多进程的操作系统，进程是独立的任务，拥有各自的权利和责任。如果一个进程崩溃，它不应当影响到系统的另一个进程随之崩溃。每一个独立的进程运行在自己的虚拟地址空间，除了通过安全的核心管理的机制之外无法影响其它进程。

在一个进程的生命周期中，进程会需要使用许多系统资源。例如占用系统的 CPU 执行它的指令，占用系统的物理内存来存储程序段和数据。或者打开和使用文件系统中的文件，直接或者间接使用系统的物理设备。如果一个进程独占了系统的大部分物理内存和 CPU，对于其他进程就是不公平的，资源的使用也不是均衡的。所以 Linux 必须跟踪进程本身和它使用的系统资源以便公平均衡的去管理系统中的进程。

系统最宝贵的资源就是 CPU[25]。目前常用的都是单 CPU 系统。Linux 作为一个多进程的操作系统，它的设计目标就是公平而充分的利用 CPU 让进程在系统中得到运行。一般情况都是进程数多于 CPU，因此其他的进程就必须等到当前进程将 CPU 释放以后才能运行。多进程的思想是：一个进程一直运行，直到它必须等待，通常是等待一些系统资源(也包括时间片)，当其拥有了所需资源以后才可以继续运行。在一个单进程的系统中，比如 DOS 系统中 CPU 被简单地设为空闲，这样等待资源的时间就会被浪费。而在一个多进程的系统中，同一时刻许多进程在内存中并发，当一个进程必须等待时，操作系统将 CPU 从当前进程切换到另一个满足调度的进程。

在 Linux 中，每个进程用一个 task_struct 的结构来表示，其中包含了进程的所有信息，Linux 用其管理系统中的进程。代表进程的数据结构指针形成一个数组，数组的大小即允许并发的进程的数量。调度程序维持 current 指针来指向当前运行的进程。

linux 内核是多任务的内核，这意味着多个进程可以同时运行，就好像它们是系统中唯一存在的进程一样。通过调度程序来控制操作系统在特定时刻选择哪一个进程获得系统的 cpu。

调度程序负责在不同的进程之间进行 cpu 的切换，并决定进程以怎样的次序获得对 cpu 的访问。Linux 与大多数操作系统一样，通过时钟中断触发调度程序，当时钟中断发生时，内核必须决定当前进程是否要为另外的进程让出 cpu，让出之后接下来哪一个进程应该获得 cpu。两个时钟中断之间的时间间隔称为时间片。

系统的进程往往分为两种类型：交互式的和非交互式的。交互式进程非常依赖于 I/O。因此，通常使用不完自己的部分时间片，而是把 cpu 让给其他进程。非交互式进程非常依赖于 cpu，通常使用自己大部分的时间片。调度程序必须平衡这两种类型进程的需求，并且努力确保每个进程能获得足够的时间以完成它的任务，而且还不能对其他进程的执行有不利影响。

Linux 还区别另一种类型的进程：实时进程。实时进程必须实时地执行，Linux 支持实时进程，但它们处于调度逻辑的外部，即 linux 的调度程序把标记为实时的所有进程视为比其他任何进程的优先级都高。实时进程的开发者确保这些进程不会贪婪地占有 cpu，并确保它们最终放弃 cpu。

调度程序通常使用某种类型的进程队列管理系统中进程的执行，在 linux 中，这个进程队列称为运行队列。运行队列的定义与调度程序有着密切的关系。在 Linux 以往的版本中，调度算法的复杂度与进程的数量呈线性关系，即 O(n)，在那些版本的内核中，就绪进程队列是一个全局数据结构，调度器对它的所有操作都会因全局自旋锁而导致系统各个处理机之间的等待，使得就绪队列成为一个明显的瓶颈。linux2.4 的就绪队列就是一个简单的以 runqueue_head 为表头的双向链表。但在 linux 2.6 的版本时，采用了一种新的数据结构，使得算法的时间复杂度提高到了 O（1）。这其中的改进很大程度上与在此内核版本中对数据结构的选定有关，在 Linux2.6 中，就绪队列定义为一个复杂得多的数据结构 struct runqueue，并且，尤为关键的改进点是，每一个 CPU 都将维护一个自己的就绪队列，这将大大减小竞争。Linux2.6 内核的 O(1)算法中很多关键技术都与 runqueue 有关。本节将详细分析该数据结构。

5.1.1 Linux2.6 进程调度中的数据结构

调度程序操作的对象是一个称为运行队列(RUN QUEUE)的结构。系统中每个CPU 都包含自己的运行队列。 运行队列中的核心数据结构是两个按优先权排列的数组。其中一个包含了运行进程；另一个包含了到期的进程。通常，一个活跃的进程运行一段固定的时间(时间片长度或 0 时间片)之后被插入到期数组去等待更多的 CPU 时间，当活跃数组为空时，调度程序通过交换活跃指针和到期数组的指针来交换这两个数组。然后，调度程序开始执行新活跃数组中的进程。图 5-1 说明了该运行队列的基本结构。

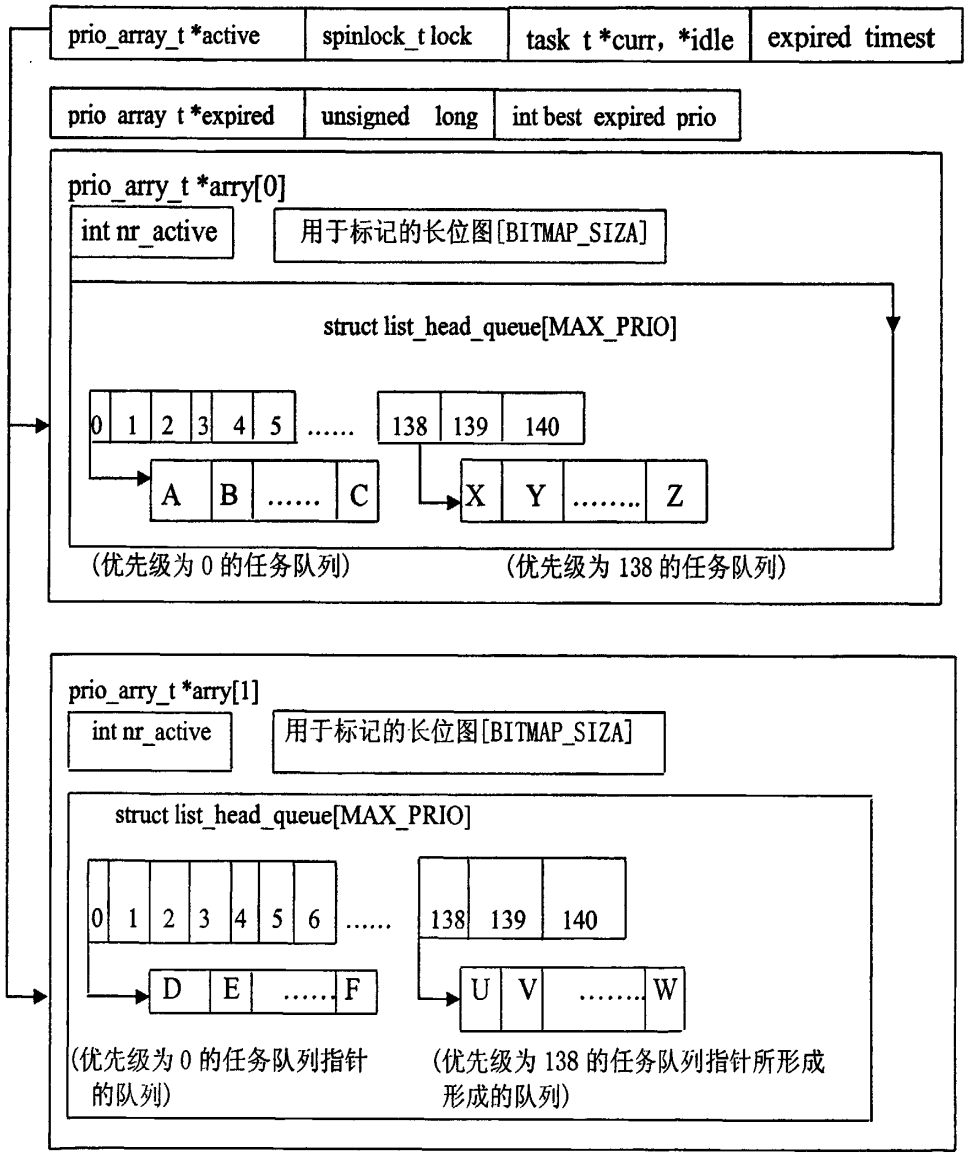

图 5-1 运行队列结构

这个图只标出了运行队列，即 run_queue 的某些数据成员。图 5-1 说明了运

行队列的优先权数组，优先权数组结构定义如下：

```
struct prio_array
{   int  nr_active;
    unsigned long bitmap[BITMAP_SIZE];
    struct list_head queue[MAX_PRIO];   };
```

Prio_array 结构的字段如下所定义：

① nr_active 计数器，记录优先权数组中的进程数。

② Bitmap 以位图映射的方式记录数组中相应优先权的进程队列。bitmp 的实际长度依赖于系统中无符号长整型的大小。它始终足够存放 MAX_PRIO 个位。但这样，bitmap 可能会更长。

③queue 存储进程链表的数组。每个链表含有特定优先权的进程。这样，queue[0]就保存所有优先权为 0 的进程的链表；queue[1]就保存了所有优先权为 1 的进程的链表等。这里的 queue 并不是 task_struct 结构指针，而是 task_struct::run_list，就是应用了上述链表的一个小技巧。

另外的一些数据成员介绍如下：

① spinlock_t lock：runqueue 的自旋锁，当需要对 runqueue 进行操作时，仍然应该锁定，但这个锁定操作只影响一个 CPU 上的就绪队列，因此，竞争发生的概率要小多了。

② task_t *curr ，*idle：curr 表示本 CPU 正在运行的进程，idle 指向 CPU 的 idle 进程，相当于 Linux2.4 中 init_tasks[this_cpu()] 的作用。

③ int best_expired_prio：记录 expired 就绪进程组中的最高优先级（数值最小）。该变量在进程进入 expired 队列的时候保存（schedule_tick()）。

④ unsigned long expired_timestamp：当新一轮的时间片递减开始后，这一变量记录着最早发生的进程耗完时间片事件的时间（jiffies 的绝对值，在 schedule_tick() 中赋），它用来表示 expired 中就绪进程的最长等待时间。它的使用体现在 EXPIRED_STARVING(rq) 宏上。

本论文前面提到，每个 CPU 上维护了两个就绪队列，active（运行队列）和 expired（到期队列）。一般情况下，时间片用完的进程应该从 active 队列转移到 expired 队列中（schedule_tick()），但如果该进程是交互式进程（即 I/O 繁忙型）调度器会让其继续保持在 active 队列上以提高它的响应速度。这种措施不应该让其他就绪进程等待过长时间，也就是说，如果 expired 队列中的进程已经等待了足够长时间了，即使是交互式进程也应该转移到 expired 队列上

来，排空 active 队列。这个阀值就体现在 EXPIRED_STARVING(rq)上。在 expired_timestamp 和 STARVATION_LIMIT 都不等于 0 的前提下，如果下列两个条件都满足，则 EXPIRED_STARVING()返回真：

(i)（当前绝对时间 - expired_timestamp） ≥ （STARVATION_LIMIT * 队列中所有就绪进程总数 + 1），也就是说 expired 队列中至少有一个进程已经等待了足够长的时间；

(ii)正在运行的进程的静态优先级比 expired 队列中最高优先级要低（best_expired_prio，数值要大），此时当然应该尽快排空 active 切换到 expired 队列上来。

⑤ unsigned long nr_running：本 CPU 上的就绪进程数，该数值是 active 和 expired 两个队列中进程数的总和，是说明本 CPU 负载情况的重要参数。

理解进程可以从 CPU 系统上的调度程序来入手。在阅读其源码以前，本论文下面部分将研究 Linux2.6 版本的调度算法。

普通进程的调度选择算法基于进程的优先级，拥有最高优先级的进程被调度器选中。Linux2.4 中，时间片 counter 同时也表示了一个进程的优先级。Linux2.6 中时间片用任务描述符中的 time_slice 域表示，而优先级用 prio（普通进程）或者 rt_priority（实时进程）表示。

调度器为每一个 CPU 维护了两个进程队列数组：active 队列和 expire 队列。数组中的元素是保存某一优先级的进程队列指针。系统一共有 140 个不同的优先级，因此这两个数组大小都是 140。当需要选择当前最高优先级的进程时，2.6 调度器不用遍历整个 runqueue，而是直接从 active 数组中选择当前最高优先级队列中的第一个进程。假设当前所有进程中最高优先级为 50(即系统中没有任何进程的优先级小于 50)。则调度器直接读取 active[49]，得到优先级为 50 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，这种算法的复杂度为 O(1)，从而解决了 2.4 调度器的扩展性问题。为了实现上述算法 active 数组维护了一个 bitmap，当某个优先级别上有进程被插入列表时，相应的比特位就被置位。Sched_find_first_bit()函数查询该 bitmap，返回当前被置位的最高优先级的数组下标。在上例中 sched_find_first_bit 函数将返回 49。在 IA 处理器上可以通过 bsfl 等指令实现。为了提高交互式进程的响应时间，O(1)调度器不仅动态地提高该类进程的优先级，还采用以下方法：

每次时钟 tick 中断中，进程的时间片(time_slice)被减 1。当 time_slice 为 0 时，调度器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 队列。如果不是交互式进程则从 active 队列中移到 expired 队列。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 队列中，否则进入 expire 队列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者交互式进程也会被移到 expire 队列中。当 active 队列中的所有进程都被移到 expire 队列中后，调度器交换 active 队列和 expire 队列。当进程被移入 expire 队列时，调度器会重置其时间片，因此新的 active 队列又恢复了初始情况，而 expire 队列为空，从而开始新的一轮调度。

从高层来看，调度程序仅仅是是对特定数据结构进行操作的一组函数。实现调度程序的所有代码几乎都能在 kernel/sched.c 和 include/linux/sched.h 中找到。在调度程序中,任务(或进程)是数据结构和控制流程的一个集合。调度程序的代码也引用了 task_struct，它是 linux 内核用来记录进程信息的数据结构。

5.1.2 Linux2.6 进程调度机制的改进分析

（一）选择下一个进程

进程被初始化并放到运行队列后，在某个时刻，它获得对 cpu 的访问。负责把 cpu 的控制权传递到不同进程的两个函数是 schedule()和 schedule_tick()。scheduler_tick()是一个由内核周期性调用的系统定时器，它把进程标记为需要重新调度。定时时件发生时，当前的进程就被保存起来，linux 内核接管对 cpu 的控制。定时事件完成后，linux 内核通常把控制权传回被保存的进程。然而,当所保存的进程被标记为需要重新调度时，内核调用 schdule()来选择需激活哪一个进程，但并不一定选择接管控制前正在执行的哪个进程。在内核接管控制前处于执行状态的进程称为当前进程。为了避免情况太复杂，在某些条件下，内核可以从内核获得控制权，称为内核抢占。本论文下面将分析调度程序是如何决定来选择下一个进程。

首先是 schedul()，linux 内核利用它决定哪个进程接下来要执行，其次是 scheduler_tick()，内核利用它决定哪个进程必须让出 cpu。这两个函数组合的结果说明了调度程序中的控制流程。

```
[kernel/schedu.c]
asmlinkage void __sched schedule(void)
{
```

```
    struct task_struct *prev, *next;
    struct prio_array *array;
    struct list_head *queue;
    unsigned long long now;
    unsigned long run_time;
    int cpu, idx, new_prio;
       long *switch_count;
       struct runqueue_t *rq;
    /*
     * Test if we are atomic.  Since do_exit() needs to call into
     * schedule() atomically, we ignore that path for now.
 * Otherwise, whine if we are scheduling when we should not
*be. */
    if (unlikely(in_atomic() && !current->exit_state))
{
        printk(KERN_ERR "BUG: scheduling while atomic: "
            "%s/0x%08x/%d\n",
            current->comm, preempt_count(), current->pid);
        debug_show_held_locks(current);
        if (irqs_disabled())
            print_irqtrace_events(current);
        dump_stack();
    }
    profile_hit(SCHED_PROFILING, __builtin_return_address(0));
need_resched:
    preempt_disable();
    prev = current;
    release_kernel_lock(prev);
need_resched_nonpreemptible:
    rq = this_rq();
/* The idle thread is not allowed to schedule!
     * Remove this check after it has been exercised a bit.
```

```
        */
       if (unlikely(prev == rq->idle) && prev->state != TASK_RUNNING)
    {
       printk(KERN_ERR "bad: scheduling from the idle thread!\n");
       dump_stack();
    }
    schedstat_inc(rq, sched_cnt);
       now = sched_clock();
       if (likely((long long)(now - prev->timestamp) < NS_MAX_SLEEP_AVG))
   {
       run_time = now - prev->timestamp;
       if (unlikely((long long)(now - prev->timestamp) < 0))
              run_time = 0;
       } else
          run_time = NS_MAX_SLEEP_AVG;
       /*
        * Tasks charged proportionately less run_time at high sleep_avg
to
        * delay them losing their interactive status
        */
       run_time /= (CURRENT_BONUS(prev) ? : 1);
```

在以上代码中，计算了进程在调度程序上激活的时间长度。如果激活的时间长度大于最大的平均睡眠时间(NS_MAX_SLEEP_AVG)，就把其运行时间设置为最大的平均睡眠时间。

这就是 linux 内核代码在其他代码片段中定义的一个时间片(timeslice)。时间片既指调度程序的中断间隔时间，又指进程使用 cpu 的时间。如果进程耗尽了它的时间片，它就到期并不再激活，时间戳(timestamp)是一个绝对值，它是用来确定进程使用 cpu 时间长度的。调度程序利用时间戳来减小已使用过 cpu 的进程的时间片。

例如：假使进程 A 有一个 50 个时钟周期的时间片，它使用了 5 个时钟周期的 CPU，然后为其他进程让出 cpu。内核利用时间戳来决定进程 A 的时间片还剩 45 个周期。在此段的最后两行代码是用于交互式进程，此类进程是那些花费大量时间

用于等待输入的进程，键盘控制器就是一个很好的例子，它的大多数时间在等待输入，但当它有任务要执行，用户希望它拥有高优先权，以便快速的响应。交互式进程是指那些交互信用超过 100(默认值)的进程，它们有效 run_time 被(sleep_avg/max_sleep_avg*MAX_BOUNS(10))除。

```
[kernel/sched.c]
        spin_lock_irq(&rq->lock);
        switch_count = &prev->nivcsw;
      if (prev->state && !(preempt_count() & PREEMPT_ACTIVE))
{
         switch_count = &prev->nvcsw;
    if (unlikely((prev->state & TASK_INTERRUPTIBLE) &&
        unlikely(signal_pending(prev))))
        prev->state = TASK_RUNNING;
    else {
           if (prev->state == TASK_UNINTERRUPTIBLE)
           rq->nr_uninterruptible++;
           deactivate_task(prev, rq);
         }
}
```

在以上代码部分中，函数首先获得运行队列锁，因为要修改它，判断这是否是内核抢占，如果由于内核抢占上一个进程而进入schedule()，那么，若有信号正被挂起，我们就离开上一个运行的进程，这意味着内核紧接着抢占了正常的处理；因此，代码被两个 unlikely()语句包含，如没有更进一步的抢占，我们就从运行队列删除被抢占的进程并继续选择下一个要运行的进程。

```
[kernel/sched.c]
        cpu = smp_processor_id();
        if (unlikely(!rq->nr_running))
{     idle_balance(cpu, rq);
```

```
        if (!rq->nr_running)
          {    next = rq->idle;
               rq->expired_timestamp = 0;
               wake_sleeping_dependent(cpu);
               goto switch_tasks;
          }
        }
            array = rq->active;
            if (unlikely(!array->nr_active)) {
            /*
             * Switch the active and expired arrays.
             */
                schedstat_inc(rq, sched_switch);
            rq->active = rq->expired;
            rq->expired = array;
            array = rq->active;
            rq->expired_timestamp = 0;
            rq->best_expired_prio = MAX_PRIO;
    }
```

在这段代码中，用 smp_processor_id()获得当前 cpu 的标识符，接着查看运行队列，如果运行队列上没有进程，就设置下一个进程为 idle 进程，并且把运行队列的到期时间戳重置为 0。在多处理机系统中，首先检查是否有进程在其它 CPU 上运行而本 CPU 可以抓取，为提高效率和资源利用，在系统中的所有 CPU 上装载了用于平衡的处理机制。仅当没有进程能够从其他 CPU 移出时，才把 idle 设置为运行队列的下一个进程并重置到期的时间戳。如果运行队列的活跃数组为空，在选择一个新进程运行之前，将交换活跃数组指针和到期数组指针。

[kermel/sched.c]

```
 idx = sched_find_first_bit(array->bitmap);
  queue = array->queue + idx;
  next = list_entry(queue->next, struct task_struct, run_list);
if(!rt_task(next)&&interactive_sleep(next->sleep_type))
{     unsigned long long delta = now - next->timestamp;
    if (unlikely((long long)(now - next->timestamp) < 0))
        delta = 0;
    if (next->sleep_type == SLEEP_INTERACTIVE)
        delta = delta * (ON_RUNQUEUE_WEIGHT * 128 / 100) / 128;
    array = next->array;
    new_prio = recalc_task_prio(next, next->timestamp + delta);
    if (unlikely(next->prio != new_prio)) {
        dequeue_task(next, array);
        next->prio = new_prio;
        enqueue_task(next, array);
    }
}
next->sleep_type = SLEEP_NORMAL;
if (rq->nr_running == 1 && dependent_sleeper(cpu, rq, next))
    next = rq->idle;
```

这里，调度程序利用 sched_find_first_bit( )找到优先权最高的一个进程运行，然后让 queue 指向某个链表，这个链表在指定的位置存放优先权数组。next 被初始化为 queue 中的第一个进程。如果将要激活的进程依赖于正在睡眠的兄弟进程，就选择下一个新的进程激活，并且跳转到 switch_tasks ( ) 处继续执行调度函数。

假设现有进程 A，它派生出进程 B 从设备读取数据，进程 A 要等待进程 B 完成后才能继续执行。如果调度程序选择进程 A 激活，这段代码 dependent_sleepr( ) 会发现进程 A 正在等待进程 B，并会选择一个全新的进程激活。

```
[kernel/sched.c]
    switch_tasks:
        if (next == rq->idle)
        schedstat_inc(rq, sched_goidle);
        prefetch(next);
        prefetch_stack(next);
        clear_tsk_need_resched(prev);
        rcu_qsctr_inc(task_cpu(prev));
        update_cpu_clock(prev, rq, now);
      prev->sleep_avg =run_time;
          if ((long)prev->sleep_avg <= 0)
            prev->sleep_avg = 0;
        prev->timestamp = prev->last_ran = now;
        sched_info_switch(prev, next);
    if (likely(prev != next)) {
        next->timestamp = next->last_ran = now;
        rq->nr_switches++;
        rq->curr = next;
        ++*switch_count;
        prepare_task_switch(rq, next);
        prev = context_switch(rq, prev, next);
        barrier();
/* this_rq must be evaluated again because prev may have *moved CPUs since
it called schedule(), thus the 'rq' on its stack frame will be invalid.*/
        finish_task_switch(this_rq(), prev);
    }
  else    spin_unlock_irq(&rq->lock);
      prev = current;
    if (unlikely(reacquire_kernel_lock(prev) < 0))
```

```
        goto need_resched_nonpreemptible;
    preempt_enable_no_resched();
    if (unlikely(test_thread_flag(TIF_NEED_RESCHED)))
        goto need_resched;
}
```

在第一行,尝试着把新进程 task 结构的内容预取到CPU的L1高速缓存里。接下来，由于正在进行上下文切换，所以必须通知当前 CPU，系统正在做此事。这使得多 CPU 的设备能够确保正在被其他 CPU 共享的数据被独占地访问，这个过程称为“读-拷贝更新”。

系统根据前一个进程运行的时间减小其 sleep_avg 属性，并对负值进行调整，如果进程既不是交互式也不是非交互式，它的交互信用就处于低信用和高信用之间，由于它的睡眠平均值较底，我们减小它的交互信用，并更新它的时间戳为当前时间。这个操作有助于调度程序了解给定进程已经使用了多少 CPU 时间，并有助于估算进程以后将使用多少CPU时间。

如果我们没有选择同一个进程，就设置新进程的时间戳，增加运行队列的计数器，并把新进程设置为当前进程。在重新获得内核锁后，激活抢占，并判断时候需要立刻重新调度。如果是,则回到schedule()的开头。其中context_switch()是用汇编语言编写的,它完成对进程上下文的切换,执行完 context_switch()后，我们可能需要重新调度，scheduler_tick()也许已经把新进程标记为需要重新调度了。或者,当我们激活抢占时,新进程已被标记。我们不断地重新调度进程(并对上下文切换)，直到找到一个不需要重新调度的进程。离开 schedule()的进程成为这个CPU上执行的新进程。

（二）被动让出 CPU

通过调用程序 schedule( )可使进程自动放弃CPU。在想要睡眠或等待信号发生的内核代码及设备驱动程序中，对这个函数的调用非常普遍。若其他进程想要连续地使用 CPU，系统定时器必须告诉它们让出 CPU。Linux 内核定时地夺取CPU，如此强制停止活跃进程，然后执行基于定时的许多任务。其中，这些任务之一就是 schedule_tick( )，它是内核强迫一个进程放弃CPU的函数。如果一个进程运行时间过长，内核就不把 CPU 的控制权返回给这个进程，而是改为选择另外一个进程。关于这个函数的详细源代码可在 linux 内核源文件的/kernel/sched.c里面找到。

上文已经谈到进程的调度对象是一个由进程组成的运行队列，如果一个进程是因为自己的运行时间到期而结束运行，则它将被移到运行队列的到期数组中，以便下一轮的再次调度。但是，当进程运行结束或者由于某些原因使得进程处于运行态以外的某种状态，我们应该将此进程从运行队列中删除。那进程是怎样从队列中删除或者插入，并能方便的反映给调度程序的？下面将分析从运行队列中删除的情况。

主要有两种方法可以从运行队列中删除进程：

（1） 进程被内核抢占并且它的状态不是运行状态，进程没有挂起的信号；

（2） 在 SMP 机器中，进程能够从一个运行队列删除并被放到另外一个运行队列上。

第一种情形通常发生在当进程把自己放到等待队列上睡眠之后，schedule() 被调用的时候。进程把自己标记为非运行状态(TASK_INTERRUPTIBLE. TASK_UNINTERRUPTIBLE,TASK_STOPPED 等)，并且通过从运行队列删除它，内核不再允许这个进程对CPU的访问。现在，系统对如何通过deactivate_task()从运行队列删除进程进行跟踪：

```
[kernel/sched.c]
static void deactivate_task(struct task_struct *p,runqueue_t *rq)
{
  rq->nr_running--;
  if(p->state==TASK_UNINTERRUPTIBLE)
   rq->nr_uninterruptible+=;
   dequeue_task(p,,p->array);
   p->array=NULL;
}
```

由于进程 P 不再运行，调度程序首先减小它的运行进程数。如果进程是不可中断的，那么还会增加运行队列中不可中断的进程数，相应地在不可中断的进程醒来时减少操作。接着，程序再更新运行队列的统计信息。通过这样的方式，系统实际上从运行队列删除了该进程。内核利用 p->array 域检测是否有进程正在运行以及是否是在某一运行队列上。因为 p->array 不再是这两种情况，被置为了空。

除此以外还有一些运行队列的处理要完成，下面是 dequeue_task()的细节：

```
[kernel/sched.c]
 static void dequeue_task(struct task_struct *p,prio_array_t *array)
  {
    array->nr_active--;
    list_del(&p->run_list);
    if(list_empty(array->queue+p->prio))
       _clear_bit(p->prio,array->bitmap);
  }
```

在这段代码中，我们调整优先权数组上的活跃进程数，进程 p 若不在到期数组上，就在活跃数组上。系统根据 p 的优先权从优先权数组中的进程链表中删除进程。如果因删除而导致链表为空，则必须清除优先权数组的位图中的相应位，以此表示再也没有任何进程的优先权为 p->prio()。

由于 p->run_list 是一个 list_head 的结构，list_del()仅用一步就能完成所有的删除操作。从运行队列删除进程从而使进程完全不再活跃的地步，如果这个进程处于 TASK_INTERRUBTIBLE 或 TASK_UNINTERRUPTIBLE 状态，它可能被唤醒并被放回运行队列。如果进程处于 TASK_STOPPED，TASK_ZPMBLE 或 TASK_DEAD 状态，则它的所有结构被删除并废弃。

5.1.3 Linux2.6 调度算法小结

上一节详细分析了 Linux 进程的调度结构，并且研究了主要的源代码。在此简单总结一下：Linux2.6 调度器改进了前任调度器 Linux2.4 的可扩展性问题，schedule()函数的时间复杂度由线性阶 O(n)变为常数阶 O(1)。这取决于两个改进：

(1) Pick next 算法借助于 active 数组，不再需要遍历 runqueue；

(2) 取消了定期更新所有进程 counter 的操作，动态优先级的修改分布在进程切换，时钟 tick 中断以及其它一些内核函数中进行。

和 2.4 的调度器相比，2.6 的 schedule()函数要更加简单一些，减少了锁操作，优先级计算也放在调度器外进行了。为减少进程在 cpu 间频繁跳跃，2.4

中将被切换下来的进程重新调度到另一个 cpu 上的动作也省略了。调度器的基本流程仍然可以概括为相同的五步：

清理当前运行中的进程（prev）—选择下一个投入运行的进程（next）—设置新进程的运行环境—执行进程上下文切换—后期整理 。

2.6 的调度器工作流程保留了很多 2.4 系统中的动作，进程切换的细节也与 2.4 基本相同（由 context_switch() 开始）。下面从三点观察两种内核版本的不同：

（1）相关锁

主要是因为就绪队列分布到各个 cpu 上了，2.6 调度器中仅涉及两个锁的操作：就绪队列锁 runqueue::lock，全局核心锁 kernel_flag。对就绪队列锁的操作保证了就绪队列的操作唯一性，核心锁的意义与 2.4 中相同：调度器在执行切换之前应将核心锁解开（release_kernel_lock()），完成调度后恢复锁状态（ reacquire_kernel_lock() ） 。 进 程 的 锁 状 态 依 然 保 存 在 task_struct::lock_depth 属性中。

因为调度器中没有任何全局的锁操作，2.6 调度器本身的运行障碍就几乎不存在了。

（2）prev

调度器主要影响 prev 进程的两个属性：

①sleep_avg 减去了本进程的运行时间（详见"进程平均等待时间 sleep_avg"的"被切换下来的进程"）；

②timestamp 更新为当前时间，记录被切换下去的时间，用于计算进程等待时间。

prev 被切换下来后，即使修改了 sleep_avg，它在就绪队列中的位置也不会改变，它将一直以此优先级参加调度直至发生状态改变（比如休眠）。

（3）next

在前面介绍 runqueue 数据结构的时候，已经分析了 active/expired 两个按优先级排序的就绪进程队列的功能，2.6 的调度器对候选进程的定位有三种可能：

①active 就绪队列中优先级最高且等待时间最久的进程；

②当前 runqueue 中没有就绪进程了，则启动负载平衡从别的 cpu 上转移进程，再进行挑选（可查阅"调度器相关的负载平衡"）；

③如果仍然没有就绪进程，则将本 cpu 的 IDLE 进程设为候选。

在挑选出 next 之后，如果发现 next 是从 TASK_INTERRUPTIBLE 休眠中醒来后第一次被调度到（activated>0），调度器将根据 next 在就绪队列上等待的时长重新调整进程的优先级（并存入就绪队列中新的位置，详见"进程平均等待时间 sleep_avg"）。

除了 sleep_avg 和 prio 的更新外，next 的 timestamp 也更新为当前时间，用于下一次被切换下来时计算运行时长。

Linux2.6 的调度器，除核心应用主动调用调度器之外，核心还在应用不完全感知的情况下在以下三种时机中启动调度器工作：

①从中断或系统调用中返回；

②进程重新允许抢占（preempt_enable()调用 preempt_schedule()）；

③主动进入休眠（例如 wait_event_interruptible()接口）。

Linux2.6 内核调度系统相比 Linux2.4 而言有两点新特性对实时应用至关重要：内核抢占和 O(1) 调度。其中内核抢占机制使得任何没有锁保护的代码都能够被中断，抢占其 CPU，使得后到达的高优先级进程能够及时获得运行。这两点改进保证了实时进程能在可预计的时间内得到响应。这种"限时响应"的特点符合软实时（soft realtime）的要求，离"立即响应"的硬实时（hard realtime）还有一定距离。并且，Linux2.6 调度系统仍然没有提供除 cpu 以外的其他资源的剥夺运行，同时其进程优先级的设置也比并非完全能够表示真正的轻重缓急程度，因此，尽管较 Linux2.4 有改进，但它的实时性并没有得到突破性的进步。论文下一节将提出自己的解决方案。

5.2 Linux2.6 进程调度的实时性改进

本章前面分析了 Linux2.6 版本较 2.4 版本，在内核主体中通过采用可抢占内核和更加有效的调度算法来提高中断性能和改进调度响应时间，但由于 Linux 内核设计仍然以吞吐量为主要考虑，因此该系统在实时调度方面仍然存在着较多的不足之处。主要有以下几个突出的方面：

(1) 频繁关中断会造成中断丢失，因此由中断触发的那些实时任务不能立即被调度而获得执行。在 Linux 系统调用中涉及很多内核数据的处理，为了保护内核数据会执行大量的互斥操作。因此为了保护临界资源，中断会长时间被关闭。若是低优先级的进程关闭了中断，那么即使发生了高优先级实时进程的中断，系统也无法响应。这样的情况在实时系统中是不允许出现的。

（2）Linux 并非完全根据进程的优先级来进行调度。对于系统中的所有进程，无论它们的优先级大小，终会在某个时间获得一个时间片得以运行。那么，当低优先级的进程获得时间片运行时，高优先级的进程必须等到低优先级进程的时间片结束后方可获得 CPU。在实时系统中，不能够出现高优先级进程等待低优先级进程的现象。

（3）只要高优先级进程的中断是打开状态，则可能会被任意的中断打断。若该中断的中断服务程序（ISR）执行时间太长，将会影响到该高优先级进程的执行时间。

（4）粗糙的时钟粒度不能够提供精确的定时以满足实时应用微秒级的响应需求。[25]很多中断需要系统在几十μs 内作出响应，但 Linux2.6 版本内核中设计的时间频率是 100Hz，即中断是 1-10ms 一次，很显然，ms 级的时钟粒度不能满足实时应用中响应的μs 级需求。

本节将针对 Linux2.6 在实时调度方面的不足之处提出一种解决方案，包含两种策略，通过修改的数据结构加以改进。

5.2.1 提高时钟精度的策略

开源的 Linux 对改进时钟精度提供了一些解决方案，主流的有 KURT-Linux、RT-Linux 和 Monta-Vista Linux 等。[26]

KURT-Linux 由 Kansas 大学研制开发，它将原来时钟中断的固定频率设置为单次触发模式（one shot mode），即每次为时钟芯片设置一个超时时间，然后在超时时间发生时，在时钟中断处理程序中再次根据需要为时钟芯片设置一个超时时间（μs）。KURT-Linux 利用 Pentium 架构 CPU 提供的 time stamp clock（TSC）跟踪系统时间，以到期 TSC 时标和当前 TSC 时标之差值作为芯片的时钟精度，可达微秒级。

RT-Linux 由新墨西哥工学院开发，在提高时钟精确度方面的策略类似于 KURT-Linux 的思想：通过将系统的实时时钟设置为单次触发状态，然后利用 CPU 的计数寄存器提供与 CPU 时钟频率相匹配的定时精确度。尤其是使 Intel8354 定时器芯片工作在 interrupt-on-termina-count 模式。这样可以将中断调度提高至 1 微秒左右的精确度。

MontaVista Linux(前身为 HardHat Linux)采用高精确度定时器 HRT（High Resolution POSIX Tim-ers），使得定时器可以产生任何微秒级的中断，不再需要

每个微秒级都产生中断。不再采用传统的周期中断 CPU 的方法，仅在最需要调度时间的那一刻中断CPU，即one-shot模式，这种解决思路也与KURT-Linux类似。MontaVista Linux实现了高定时器精确度，但缺乏Linux传统间隔定时，对Linux的一般性应用有一定的影响。

Linux—SRT 是剑桥大学 David Ingram 的博士论文项目，它简单地修改了Linux内核中HZ的定义，将Linux的时钟频率由每秒100次提高到了1024次。仅仅单纯的提高时钟中断频率，将使时钟中断产生的更频繁，这必定会引起调度负载的增加，还会频繁打乱处理器的高速缓存。[26]

本论文借鉴了KURT的思路，使用一个实时核心时钟处理系统，该系统与标准Linux核心时钟并行运行，并区别于原有的Linux核心时钟。这个实时核心时钟处理系统刻度精密，专门用来对实时任务进行定时。通过这项解决策略，能有效提高系统的稳定性和效率，同时维护这个独立的核心时钟也更加容易。

本论文利用Pentium架构CPU中time stamp clock（TSC）跟踪系统时间提供的精度标准，对于系统中所增加的此高精度定时器，需要维护两个与时钟有关的中断请求队列：系统时钟中断请求队列（timer_irq）和 HRT 中断请求队列（hrt_timer_irq），它们分别对应于各自的中断服务程序。[26] 并且还需将CONFIG_HRT_REQ 宏定义的添加在内核中，作用是控制该高精度时钟系统的运行。在该项策略中，必须要在内核中配置 CONFIG_HRT_REQ 高精度定时器机制方能有效。这里关键的数据结构是 hrt_timer.sub_ex-pires。该数据结构用于指示在一个高精度定时器到期的时候处理 jiffy 值之外的机器指令周期数，并提供高精度的判断标准。

对hrt_timer.sub_expires数据结构描述如下：

```
struct hrt_timer{
struct rb_node node;                    /*内嵌的红黑树节点*/
ktime_t sub_expires;                    /*期满时刻*/
enum hrt_timer_state state;             /*定时器状态*/
int(*function)(struct hrt_timer*);
struct hrtimer_base*base;               /*定时器基准*/
#ifdef CONFIG_HRT_REQ
int mode;                               /*定时器模式*/
struct list_head cd_entry;              /*回调函数队列*/
#endif
```

};

同时还需要对与定时器相关的函数做修改。如初始化定时器数据结构init_timer()、激活定时器的函数 add_timer ()、更改已激活的定时器超时时间函数 mod_timer( )等等。

5.2.2 实时调度算法的改进

由于Linux设计的初衷是基于通用目的，公平性和吞吐量是其主要考虑的因素。普通Linux提供两种软实时中断和一种普通分时进程调度策略。经典常用的实时调度算法有：

① 基于优先级的调度算法PD (priority driven scheduling)，调度器以优先级作为寻求下一个任务执行的依据。

② 基于时间驱动的调度算法TD (time driven schedu-ring)，该算法本质上是一种设计时就确定下来的离线的静态调度方法，在系统设计阶段，在明确系统中所有处理的情况下，对于各个任务的开始、切换以及结束时间等事先组出明确的安排和设计。

③ 基于比例共享的调度算法SD (share drivenscheduling)，该算法是按照一定的权重对一组需要调度的任务进行调度，使其执行时间与权重完全成正比。[26]

但在实际应用中的实时系统中，实时进程的优先级并不能简单的由一个权值来确定，因为需要综合考虑各种因素。比如多任务多用户不同的需求、截止时间、任务紧迫程度等，还会随着时间的推进发生变化，因此常用的典型调度算法并不能很好的满足实时需求。

本论文考虑的改进策略引入了常用的两种基于优先级的实时调度算法：基于静态优先级的速率单调调度算法(Rate Monotonic Analysis，RM) [25]和基于动态优先级的最早截止期优先算法(EarliestDeadline First，EDF) [25]，采用动态模块LKM (LinuxLoadable Kernel Mod2ule) [25]方式加载。

RM调度算法是一种经典的静态优先级抢占式调度算法。其优先级通过任务运行的周期来进行设置，周期越短，截止时间越紧，优先级越高。因此在RM算法中，优先级最高的是周期最短的进程。多任务条件下，周期最短的进程优先获得CPU。

RM算法需要三个条件：①由独立的周期性任务组成，每个任务具有自己的静态优先级；②优先级的确定策略是某个任务的周期越短，其优先级越高；③采用

可抢占的优先级调度方式。RM算法保证只要所有任务对CPU的利用率低于某一界限，那么所有任务都将满足它们的时限，而不用管某个任务应该在什么时候运行，即使发生了瞬时过载，只要CPU利用率仍在适当的界限值内，就能够保证关键任务的时限。下列条件表明了多任务条件下CPU的利用率，只要满足即可由RM调度算法所调度。

$$V = \sum_{i=1}^{n} C_i/T_i \leqslant n(2^{1/n}-1) \quad (n>1)$$

其中，C_i是第i个进程在任务周期的最大执行时间，T_i为第i个进程的运行周期，C_i/T_i是CPU的利用率。

所修改的RM调度器数据结构如下：

(1)在头文件sched. h中增加RMS的宏定义：

```
#define RMS 3;
```

(2)在进程控制块task_struct中增加采用RM调度策略的任务的属性：

```
struct RMS.  struct
{
unsigned long period;
unsigned long ready_time;
unsigned long service_time;
unsigned long time_serviced;
};
```

在该结构加入task_struct之前，对其进行初始化：

```
struct RMS_struct init_rms(struct RMS_struct init_rms)
{
Init_rms.period = -1;
Init_rms.ready_time = 0;
Init_rms.service_time = 1;
Init_rms.time_serviced = 0;
};
```

(3)同时需要设置扩展调度参数结构sched_params；

```
struct sched_params
{
int sched_priority;
```

```
unsigned long period;
unsigned long service_time;
};
```

本论文引入的第二种调度算法是EDF，也称作截止时间驱动调度算法(DDS)，是一种动态调度算法。EDF调度算法在调度时，任务的优先级按照任务的截止时间动态分配。截止时间越短的任务，优先级越高。EDF调度算法已被证明是动态最优调度，而且是充要条件。处理机利用率最大可达100%。[25]但若发生瞬时过载时，系统行为则不可预测，可能会发生多米诺骨牌现象，即一个任务丢失时将会引起一连串的任务接连丢失。另外，它的在线调度开销比RMS大[25]。

要应用EDF调度算法，需满足以下条件：

$$V = \sum_{i=1}^{n} C_i/T_i < 1 \quad (n>1)$$

其中，C_i是第i个进程在任务周期的最大执行时间，T_i为第i个进程的运行周期，C_i/T_i是CPU的利用率。

所修改的EDF调度器的数据结构如下：

1)在头文件sched. h中增加EDF的宏定义：

```
#define EDF 4;
```

2)在进程控制块task_struct中增加采用EDF调度策略的任务的属性：

```
struct EDF_struct{
unsigned long deadline;
unsigned long period;
unsigned long ready_time;
unsigned long service_time;
unsigned long time_serviced;
};
```

5.2.3 实验测试及结果分析

对于此改进方案中的策略二——实时调度模块的改进设计，我使用普通的家用电脑进行了实验测试。

测试条件：硬件平台CPU为1.7G，内存1G；软件环境为redhat9.0，对比测试标准Linux2.6.18和自己所修改过的Linux内核。测试工具为LTT(linux trace toolkit)0.9.5。

Linux trace toolkit是由opersys公司与lineo公司合作开发的Linux跟踪系统。LTT工具包主要通过跨越固定时间段重构系统行为，提供给用户精确事件信息。以图形方式观察系统动态，并为内存空间运行的实时Linux任务提供追踪能力[25]。该工具是一个用于跟踪系统详细运行状态和流程的工具，跟踪记录系统中的特定事件。LTT的具体结构见图5-2。

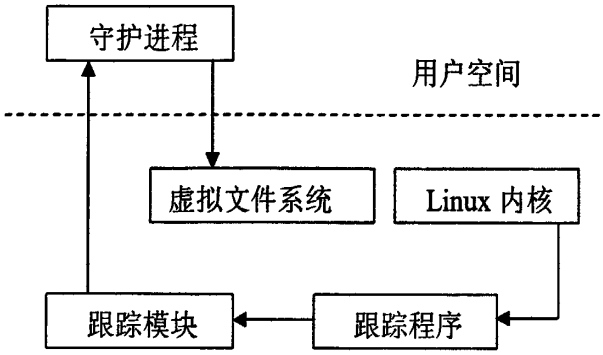

图5-2 LTT结构示意图

操作系统实时性主要可由上下文切换时间、中断延迟时间和任务响应时间等几个重要参数来衡量。其中上下文切换时间包括保存一个进程状态时间、恢复另一个进程状态的时间，是组成任务响应时间的主要组成部分。除此以外，操作系统的实时性还受硬件条件的影响，并且进行性能数据测试的环境差异也好影响测试结果。表5-1是针对普通Linux2.6.18内核与修改过的Linux内核做上下文切换时间和任务响应时间的对比参照。

表5-1 测试数据

测试对象	上下文切换时间	任务相应时间
标准Linux2.6.18	12ms	24ms
修改后的Linux	276μs	403μs

从测试数据来看，基于Linux2.6修改后的内核在上下文切换和任务响应时间值上都有较大程度的提升，较好的改进了实时调度方面存在的不足。

5.3 小结

本论文此章节基于前面几章中对于Linux源代码所作的分析，特别针对Linux2.6在实时调度这一方面所存在的不足之处做了研究，提出了基于提高时钟精度和改进的RM、EDF调度算法的改进策略，较详细的描述了改进后的主要数据结构在其中的应用。

第六章 结束语

我用了三个月的时间详细的阅读并注释 Linux 内核源代码，尤其对其中进程调度和进程控制部分的源代码作了较深入的分析。近年来，Linux 以其优秀的稳定性、灵活的架构、丰富的设备驱动、开放的源码以及良好的伸展性等特点，在桌面系统、低端服务器、高端服务器以及嵌入式系统等各方面都表现出越来越强的竞争力，对于一个仍然处于"集市式"开放开发模式的操作系统来说，能做到这一点简直就是一个奇迹。这也是我之所以选择 linux 作为研究对象的原因。

但从调度系统的实现上，通过分析，我认为 Linux 的长项仍然在桌面系统上，它仍然保持着早年开发时"利己主义"的特点，即自由软件开发者的开发动力，很大程度上来自于改变现有系统对自己"不好用"的现状，很大程度是为了为自己的某一些需求服务。尽管出于种种动机和动力，Linux 表现出与 Windows 等商用操作系统竞争的强势，但从开发者角度来看，这种愿望与自由软件的开发特点是有矛盾的。对于 Linux 的市场来说，最紧迫、最活跃的需要在于嵌入式系统。但至少从调度系统来看，Linux2.6 并没有在这方面下很大功夫，研究证明它并不能满足某些嵌入式系统的特殊要求，比如嵌入式系统在实时反馈及控制的要求。从市场的角度来看，由于 Linux 在真正商品的研发过程中具有比较大的难度，从而使得 linux 的应用在市场中占的比例还相对较少。

从学术意义上，通过对 Linux 代码的深入学习，加深了对操作系统原理的理性认识，并在较深层次上理解了操作系统的代码实现过程。通过对内核代码的分析，使我对 Linux 进程调度和进程控制部分的实现过程有了一个清晰的思路，也从中认识了数据结构的选择对操作系统性能的影响，从而也更深刻的意识到数据结构在大型软件系统中所扮演的角色。所有这些都必将对我日后的教学及科研工作起到积极的作用。同时，我在阅读内核代码的过程中深为这些代码精妙的实现方法而折服，这些代码实现的技巧以及对资源数据的组织会对本人从事目前的教学科研工作带来极大的助益。同时，对内核代码进行阅读、总结的过程也提高了我分析问题、解决问题的能力，而这种能力是无论做什么工作都需要的。

不管怎么说，linux 在嵌入式开发的领域有着广大的前景，而在如今的嵌入式产品中，对实时性的要求也越来越高，本人通过对 linux 调度器的研究学习，针对目前 Linux2.6 在实时调度方面的不足之处提出了改进的策略，并且对其中的

重要数据结构的应用也一并做了较详细的阐述。

在这次对 linux 内核代码的研读过程中，我也从中认识到了 LINUX 作为一个通用操作系统在应用中的不足。现仅谈谈自己的一点看法：在 linux 的调度算法中，当某一进程的时间片用完时，会被调度程序移动到等待运行的队列中（对于非实时进程和非交互式进程）等待下一次切换运行队列而再次得到运行。这样一来，就造成了一个很大的瓶颈，在某一负载比较大的系统中，比较高的优先级运行完了自己的时间片，并排队在等待队列中。当系统中不断有进程转入运行状态，并加入运行队列时，即使这些进程的优先级都比较低，也会在较高的优先级且已进入等待运行队列的进程运行结束前得到运行，这使的优先级较高的进程得不到快速及时的响应。在我看来，针对这一点，可在运行队列中得到改进，可以在某一进程运行完自己的时间片时，并不是把进程加入运行队列的等待队列中，而是将进程的加入运行队列的下一优先级的运行队列中，并且可以延长其下次运行的时间片，这样在保证了同级的优先得到时间片运行后，这个进程可在下一级的优先级中参与 CPU 的竞争而得到运行。

再者，在 linux 内核中，由于要计算进程的平均睡眠时间，这样，在内核中，就不得不在时间中断的程序中，累计进程的睡眠时间值，这样就必须在时间中断中遍历运行队列中的每一个进程，这样的花销是非常大的。当然这和进程选用时间片运行机制有一定的关系，关于它的解决方案还待以后的研究。在最近的版本中，有人也提出采用树的结构来组织运行队列，不再跟踪进程的睡眠时间，也不再区分交互式进程，对所有进程统一公平对待。Linux 内核机制在实时性方面也还欠缺，由于进程的切换的开销比较大（尽管在 linux 内核中是用汇编语言来实现这一功能，并且支持了内核态抢占），但线程相对就比较小，尽管 Linux 系统中有对线程的支持，但是那是通过加入软件库来实现的，而并不是在内核中对线程的支持。如果将 Linux 的调度算法以进程为单位转换为以线程为单位，这样可大大改变切换的速度，可快速的响应刚被选中的进程（或线程）。我想上述的一些问题是由于 Linux 的开发模式，尽管它是开源的，每个人都可以加入或者修改内核中的源代码，但是也正是这种比较个人式的开发，缺乏团对的紧密合作，使的 linux 内核还存在诸多待改进的地方。

由于本人的知识有限，所以对 Linux 内核的分析总是有这样或那样的不足。同时，受时间和精力的限制，并对代码的分析也没有达到非常完美的地步。通过对 Linux 内核的分析，深感自身操作系统理论的匮乏，今后还应不断加深自身的理论水平，同时注重理论与实践的结合。

致谢

感谢指导老师吴跃教授对本文作者无私的帮助。作者在分析 Linux 源码的过程中得到了吴教授耐心、细致的指导，并积极提供相应的资料，这些指导工作对作者深入理解 Linux 内核起了关键性的作用。吴教授认真审阅了作者的论文，并提出了十分中肯的修改意见，使作者能够较顺利地完成毕业论文。本人对吴教授严谨的治学态度深感钦佩。

感谢梁山伟同学为作者提供了相关资料和许多有建设性的意见，并共同探讨了部分关键问题。

感谢我的父母，他们在精神上给予的坚定支持使我能够充满信心地完成毕业论文。

感谢为作者顺利完成本学位论文工作提供帮助的所有人们。

参考文献

[1] Linux 内核源码 v2.6.18 http:// www.kernel.org

[2] Claudia Salzberg Rodriguz著.陈莉君等译.Linux内核编程.北京:机械工业出版社,2006,78-92

[3] 郭玉东,王非非著.Linux 操作系统结构分析.西安：西安电子科技大学出版社，2004，116-139

[4] Kernel Traffic . http://www.kerneltraffic.org

[5] 毛德操,胡希明.Linu 内核源代码情景分析（上）.杭州：浙江大学出版社，2006，231-246

[6] Uresh Vahalia 著.聊鸿斌译.UNIX 高级教程-系统技术内幕.北京：清华大学出版社，1999，165-166

[7] 博韦.西斯特著.陈莉君,张琼声译.深入理解 linux 内核（第二版）.北京：中国电力出版社，2004，76-106

[8] Uresh Vahalia 著.UNIX 内核新特性.北京：人民邮电出版社，2005，92-93

[9] 科波特著.魏永明,钟书毅译.Linux 设备驱动程序.北京：中国电力出版社，2005，379-381

[10] 贺永红.基于 linux 的嵌入式实时研究与改进：[硕士学位论文].贵阳：贵州大学，2007

[11] 赖娟.linux 内核分析及实时性改造：[硕士学位论文].成都：电子科技大学，2007

[12] 杜传业.嵌入式 Linux 内核解析：[硕士学位论文].天津：河北工业大学，2007

[13] 冯伟.实时嵌入式 Linux 操作系统的研究与实现：[硕士学位论文].北京：北京邮电大学，2006

[14] 胡炜，尤晋元.嵌入式 Linux 内核挂起案例调试与解析：[硕士学位论文].上海：上海交通大学，2007

[15] 林涛，孙鹤旭，云立君等.Linux 在嵌入式系统中的实现.微计算机信息, 2005，21(07):27-28.

[16] 王勇，杨勇.嵌入式 Linux 操作系统的应用移植.测控技术，2006，25(10):57-64.

[17] 王成，刘金刚.基于 Linux 的嵌入式操作系统的研究现状及发展展望.微型机与应用，2004，(05):4-6.

[18] 杨卫辉.嵌入式系统中 Linux 内核的实时化分析与实现.吉林工程技术师范学院学报，2005，21(12):14-17.

[19] 刘文峰，李程远，李善平.嵌入式 Linux 操作系统的研究.浙江大学学报，2004,38(04):447-452.

[20] 罗奕.Linux 操作系统的定制和精简.计算机时代，2005，(05):38-39.

[21] 满春涛，李鹏.嵌入式 Linux 操作系统实时性的分析与研究.自动化技术与应用，

2005,24(05):40-42.

[22] 王亚军，刘金刚.Linux 运用于嵌入式系统的技术分析.计算机应用研究，2005，(05):102-104.

[23] 赵明富，李太福，陈鸿雁等.Linux 嵌入式系统的实时性分析.计算机工程，2003，29(18):89-91.

[24] 赵明富，李太福，罗松.Linux 嵌入式系统实时性分析与实时化改进.计算机应用研究，2004,(04):200-203.

[25] 范剑英，吴岩.linux2.6 内核实时性分析与改进方案.哈尔滨理工大学学报，2008，13(1):24-28.

[26] 周鹏，周明天．Linux 内核中一种高精度定时器的设计与实现．计算机技术与发展，2006,16(4):73-78.