第 3 章 内核组件

本章将对一些驱动开发相关的内核组件进行讲解。我们首先以内核线程开始，它类似于用户空间的进程，通常用于并发处理。

另外，内核还提供了一些接口，使用它们可以简化代码、消除冗余、增强代码可读性并有利于代码的长期维护。本章会学习链表、哈希链表、工作队列、通知链(notifier chain)、完成以及错误处理辅助接口等。这些辅助接口经过了优化，而且清除了 bug，因此你的驱动可以继承这些优点。

内核线程

内核线程是一种在内核空间实现后台任务的方式。该任务可以是繁忙地处理异步事务，也可以睡眠等待某事件的发生。内核线程与用户进程相似，唯一的不同是内核线程位于内核空间可以访问内核函数和数据结构。和用户进程相似，由于可抢占调度的存在，内核现在看起来也在独占 CPU。很多设备驱动都使用了内核线程以完成辅助任务。例如，USB 设备驱动核心的 khubd 内核线程的作用就是监控 USB 集线器，并在 USB 被热插拔的时候配置 USB 设备。

创建内核线程

让我们用一个例子老学习内核线程的知识。当我们在开发这个例子线程的时候，你也会学习到进程状态、等待队列的概念，并接触到用户模式辅助函数。当你熟悉内核线程以后，你可以使用它作为在内核中进行各种各样实验的媒介。

假定我们的线程要完成这样的工作：一旦它检测到某一关键的内核数据结构的健康状态极度恶化（譬如，网络接受缓冲区的空闲内存低于警戒水位），就激活一个用户模式程序给你发送一封 email 或发出一个呼机警告。

该任务比较适合用内核线程来实现，原因如下：

（1）它是一个等待异步事件的后台任务；

（2）由于实际的事件侦测由内核的其他部分完成，本任务也需要访问内核数据结构；

（3）它必须激活一个用户模式的辅助程序，这比较耗费时间。

内建的内核线程

使用 ps 命令可以查看系统中正在运行的内核线程（也称为内核进程）。内核线程的名字被一个方括号括起来了：

```
bash> ps -ef
UID        PID  PPID  C STIME TTY          TIME CMD
root         1     0  0 22:36 ?        00:00:00 init [3]
root         2     0  0 22:36 ?        00:00:00 [kthreadd]
root         3     2  0 22:36 ?        00:00:00 [ksoftirqd/0]
root         4     2  0 22:36 ?        00:00:00 [events/0]
root        38     2  0 22:36 ?        00:00:00 [pdflush]
root        39     2  0 22:36 ?        00:00:00 [pdflush]
root        29     2  0 22:36 ?        00:00:00 [khubd]
root       695     2  0 22:36 ?        00:00:00 [kjournald]
...
root      3914     2  0 22:37 ?        00:00:00 [nfsd]
root      3915     2  0 22:37 ?        00:00:00 [nfsd]
...
root      4015  3364  0 22:55 tty3     00:00:00 -bash
root      4066  4015  0 22:59 tty3     00:00:00 ps -ef
```

[ksoftirqd/0]内核线程是实现软中断的助手。软中断是由中断发起的可以被延后执行的底半部。在第 4 章《打下基础》将对底半部和软中断进行详细的分析，这里的基本理论是让中断处理程序中的代码越少越好。中断处理时间越小，系统屏蔽中断的时间会越短，这会降低时延。Ksoftirqd 的工作是确保高负荷情况下，软中断既不会饥饿，又不至于压垮系统。在对称多处理器（SMP）及其上，多个线程实例可以并行地运行在不同的处理器上，为了提高吞吐率，系统为每个 CPU 都创建了一个 ksoftirqd 线程（ksoftirqd/n，其中 n 代表了 CPU 序号）。

events/n（其中 n 代表了 CPU 序号）实现了工作队列，它是另一种在内核中延后执行的手段。内核中期待延迟执行工作的程序可以创建自己的工作队列，或者使用缺省的 events/n 工作者线程。第 4 章也对工作队列进行了深入分析。

pdflush 内核线程的任务是对页高速缓冲中的脏页进行写回（flush out）。页高速缓冲会对磁盘数据进行缓存，为了提供性能，实际的磁盘写操作会一直延迟到 pdflush 后台程序将脏数据写回磁盘才进行。当系统中可用的空闲内存低于门限，或者页变成脏页后一段时间。在 2.4 内核中，这 2 个任务分配被 bdflush 和 kupdated 这 2 个单独的线程完成。你可能会注意到 ps 的输出中有 2 个 pdflush 的实例，如果内核感觉到现存的实例已经在满负荷运转，它会创建 1 个新的实例以服务磁盘队列。当你的系统有多个磁盘，而且要频繁访问它们的时候，这种方式会提高吞吐率。

在以前的章节中我们已经看到，kjournald 是通用内核日志线程，它被 EXT3 等文件系统使用。

Linux 网络文件系统（NFS）通过一套名为 nfsd 的内核线程实现。

在被内核中负责监控我们感兴趣的数据结构的任务唤醒之前，我们的例子线程一直会放弃 CPU。在被唤醒后，它激活一个用户模式辅助程序并将恰当的身份代码传递给它。

使用 kernel_thread()可以创建内核线程：

```
ret = kernel_thread(mykthread, NULL,
        CLONE_FS | CLONE_FILES | CLONE_SIGHAND | SIGCHLD);
```

标记参数定义了父子之间要共享的资源。CLONE_FILES 意味着打开的文件要被贡献，CLONE_SIGHAND 意味着信号处理被共享。

清单 3.1 显示了例子的实现。由于内核线程通常是设备驱动的助手，它们往往在驱动初始化的时候被创建。但是，本例的内核线程可以在任意合适的位置被创建，例如 init/main.c。

这个线程开始的时候调用daemonize()，它会执行初始的家务工作，之后将本线程的父亲线程改为kthreadd。每个Linux线程有一个父亲。如果某个父进程在没有等待其所有子进程都退出的时候就死掉了，它的所有子进程都会成为僵死进程（zombie process），仍然消耗资源。将父亲重新定义为kthreadd可以避免这种情况，并且确保线程退出的时候能进行恰当的清理工作[1]。

[1]在 2.6.21 及更早的内核中，daemonize()会通过调用 reparent_to_init()将本线程的父亲置为 init 任务。

由于 daemonize()在默认情况下会阻止所有的信号，因此，你的线程如果想处理某个信号，应该调用 allow_signal()来使能它。在内核中没有信号处理函数，因此我们使用 signal_pending()来检查信号的存在并采取适当的行动。出于调试目的，清单 3.1 中的代码使能了 SIGKILL 的传递，在收到该信号后，本线程会寿终正寝。

面对更高层次的 kthread API（其目的在于超越 kernel_thread()），kernel_thread()的地位下降了。以后我们会分析 kthreads。

清单 3.1 实现一个内核线程

```
static DECLARE_WAIT_QUEUE_HEAD(myevent_waitqueue);
rwlock_t myevent_lock;
extern unsigned int myevent_id;  /* Holds the identity of the
                                    troubled data structure.
                                    Populated later on */
static int mykthread(void *unused)
{
  unsigned int event_id = 0;
  DECLARE_WAITQUEUE(wait, current);
  /* Become a kernel thread without attached user resources */
  daemonize("mykthread");

  /* Request delivery of SIGKILL */
  allow_signal(SIGKILL);

  /* The thread sleeps on this wait queue until it's
     woken up by parts of the kernel in charge of sensing
```

```
    the health of data structures of interest */
  add_wait_queue(&myevent_waitqueue, &wait);


  for (;;) {
   /* Relinquish the processor until the event occurs */
   set_current_state(TASK_INTERRUPTIBLE);
   schedule();  /* Allow other parts of the kernel to run */
   /* Die if I receive SIGKILL */
   if (signal_pending(current)) break;
   /* Control gets here when the thread is woken up */
   read_lock(&myevent_lock);      /* Critical section starts */
   if (myevent_id) { /* Guard against spurious wakeups */
     event_id = myevent_id;
     read_unlock(&myevent_lock); /* Critical section ends */
     /* Invoke the registered user mode helper and
        pass the identity code in its environment */
     run_umode_handler(event_id); /* Expanded later on */
   } else {
     read_unlock(&myevent_lock);
   }
  }


  set_current_state(TASK_RUNNING);
  remove_wait_queue(&myevent_waitqueue, &wait);
  return 0;
}
```

将其编译入内核并运行，在 ps 命令的输出中，你将看到这个线程 mykthread：

```
bash> ps -ef
   UID        PID  PPID C STIME TTY      TIME CMD
   root         1     0 0 21:56 ?    00:00:00 init [3]
   root         2     1 0 22:36 ?    00:00:00 [ksoftirqd/0]
   ...
   root       111   1 0 21:56 ?     00:00:00 [mykthread]
   ...
```

在我们深入探究线程的实现之前，先写一段代码，让它监控我们感兴趣的数据结构的“健康状态”，一旦发生问题就唤醒 mykthread：

```
/* Executed by parts of the kernel that own the
   data structures whose health you want to monitor */
/* ... */

if (my_key_datastructure looks troubled) {
  write_lock(&myevent_lock);  /* Serialize */
  /* Fill in the identity of the data structure */
  myevent_id = datastructure_id;

  write_unlock(&myevent_lock);

  /* Wake up mykthread */
  wake_up_interruptible(&myevent_waitqueue);
}

/* ... */
```

清单 3.1 运行在进程上下文，而上面的代码即可以运行于进程上下文，又可以运行于中断上下文。进程和中断上下文通过内核数据结构通信。在我们的例子中用于通信的是 myevent_id 和 myevent_waitqueue。myevent_id 包含了有问题的数据结构的身份信息，对它的访问通过加锁进行了串行处理。

要注意的是只有在编译时配置了 CONFIG_PREEMPT 的情况下，内核线程才是可抢占的。如果 CONFIG_PREEMPT 被关闭，如果你运行在没有抢占补丁的 2.4 内核上，如果你的线程不进入睡眠状态，它将使系统冻结。如果你注释掉清单 3.1 中的 schedule ()，并且在内核配置时关闭了 CONFIG_PREEMPT 选项，你的系统将被锁住。

在第 19 章《用户空间的设备驱动》讨论调度策略时，你将学会怎样从内核线程获得软实时响应。

进程状态和等待队列

下面的语句是清单 3.1 中将 mykthread 置于睡眠状态并等待时间的代码片段：

```
add_wait_queue(&myevent_waitqueue, &wait);
for (;;) {

 /* ... */
 set_current_state(TASK_INTERRUPTIBLE);
 schedule();    /* Relinquish the processor */
 /* Point A */

 /* ... */
}
set_current_state(TASK_RUNNING);
remove_wait_queue(&myevent_waitqueue, &wait);
```

上面代码片段的操作依靠了 2 个概念:等待队列和进程状态。

等待队列用于存放需要等待事件和系统资源的线程。在被负责侦测事件的中断服务程序或另一个线程唤醒之前，位于等待队列的线程会处于睡眠。入列和出列的操作分别通过调用 add_wait_queue()和 remove_wait_queue()完成，而唤醒队列中的任务则通过 wake_up_interruptible()完成。

一个内核线程（或一个常规的进程）可以处于如下状态中的一种：运行（running）、可被打断的睡眠（interruptible）、不可被打断的睡眠（uninterruptible）、僵死（zombie）、停止（stopped）、追踪（traced）和 dead。这些状态的定义位于 include/linux/sched.h：

（1）处于运行状态（TASK_RUNNING）的进程位于调度器的运行队列（run queue）中，等待调度器将 CPU 时间分给它执行；

（2）处于可被打断的睡眠状态（TASK_INTERRUPTIBLE）的进程正在等待一个事件的发生，不在调度器的运行队列之中。当它等待的事件发生后，或者它被信号打断，它将重新进入运行队列；

（3）处于不可被打断的睡眠状态（TASK_INTERRUPTIBLE）的进程与处于可被打断的睡眠状态的进程的行为相似，唯一的区别是信号的发生不会导致该进程被重新放入运行队列；

（4）处于停止状态（TASK_STOPPED）的进程由于收到了某些信号已经停止执行；

（5）如果 1 个应用程序（如 strace）正在使用内核的 ptrace 支持以拦截一个进程，该进程将处于追踪状态（TASK_TRACED）；

（6）处于僵死状态(EXIT_ZOMBIE)的进程已经被终止，但是其父进程并未等待它完成。一个退出后的进程要么处于 EXIT_ZOMBIE 状态，要么处于死亡状态（EXIT_DEAD）。

你可以使用 set_current_state()来设置内核线程的运转状态。

现在回到前面的代码片段。mykthread 在等待队列 myevent_waitqueue 上睡眠，并将它的状态修改为 TASK_INTERRUPTIBLE，表明了它不想进入调度器运行队列的愿望。对 schedule()的调用将导致调度器从运行队列中选择一个新的任务投入运行。当负责健康状况检测的代码通过 wake_up_interruptible(&myevent_waitqueue)唤醒 mykthread 后，该进程将重新回到调度器运行队列。而与此同时，进程的状态也改变为 TAS

K_RUNNING，因此，以便唤醒的动作发生在设置。另外，如果 SIGKILL 被传递给了该线程，它也会返回运行队列。之后，一旦调度器从运行队列中选择了 mykthread 线程，它将从 Point A 开始恢复执行。

用户模式辅助函数

清单 3.1 中的 mykthread 会通过调用 run_umode_handler()向用户空间通告被侦测到的事件：

```
/* Called from Listing 3.1 */
static void
run_umode_handler(int event_id)
{
  int i = 0;
  char *argv[2], *envp[4], *buffer = NULL;
  int value;

  argv[i++] = myevent_handler; /* Defined in
                          kernel/sysctl.c */

  /* Fill in the id corresponding to the data structure
     in trouble */
  if (!(buffer = kmalloc(32, GFP_KERNEL))) return;
  sprintf(buffer, "TROUBLED_DS=%d", event_id);

  /* If no user mode handlers are found, return */
  if (!argv[0]) return; argv[i] = 0;

  /* Prepare the environment for /path/to/helper */
```

```
    i = 0;
    envp[i++] = "HOME=/";
    envp[i++] = "PATH=/sbin:/usr/sbin:/bin:/usr/bin";
    envp[i++] = buffer; envp[i]  = 0;

    /* Execute the user mode program, /path/to/helper */
    value = call_usermodehelper(argv[0], argv, envp, 0);

    /* Check return values */
    kfree(buffer);
}
```

内核支持这种机制：向用户模式的程序发出请求，让其执行某些程序。run_umode_handler()通过调用 call_usermodehelper()使用了这种机制。你必须通过/proc/sys/目录中的一个结点来注册 run_umode_handler()要激活的用户模式程序。为了完成此项工作，必须确保 CONFIG_SYSCTL（/proc/sys/目录中的文件全部都被看作 sysctl 接口）配置选项在内核配置时已经使能，并且在 kernel/sysctl.c 的 kern_table 数组中添加一个入口：

```
{
    .ctl_name    = KERN_MYEVENT_HANDLER, /* Define in
                                    include/linux/sysctl.h */
    .procname    = "myevent_handler",
    .data        = &myevent_handler,
    .maxlen      = 256,
    .mode        = 0644,
    .proc_handler = &proc_dostring,
    .strategy    = &sysctl_string,
},
```

上述代码会导致 proc 文件系统中产生新的/proc/sys/kernel/myevent_handler 结点。为了注册用户模式辅助程序，运行如下命令：

```
bash> echo /path/to/helper > /proc/sys/kernel/myevent_handler
```

当 mykthread 调用 run_umode_handler()时，/path/to/helper 程序将开始执行。

mykthread 通过 TROUBLED_DS 环境变量将有问题的内核数据结构的身份信息传递用户模式辅助程序。该辅助程序可以是一段简单的脚本，它发送给你一封包含了从环境变量搜集到的信息的邮件警报：

```
bash> cat /path/to/helper

#!/bin/bash

echo Kernel datastructure $TROUBLED_DS is in trouble | mail -s Alert root
```

对 call_usermodehelper()的调用必须发生在进程上下文，而且以 root 权限运行。它借用下文很快就要讨论的工作队列（work queue）得以实现。

辅助接口

内核中存在一些有用的辅助接口，这些接口可以有效地减轻驱动开发人员的负担。其中的一个例子就是双向链表库的实现。许多设备驱动需要维护和操作链表数据结构，内核的链表接口函数消除了管理链表指针的需要，也使得开发人员无需调试与链表维护相关的繁琐问题。本节我们将学会怎样使用链表（list）、哈希链表（hlist）、工作队列（work queue）、完成函数（completion function）、通知块（notifier block）和 kthreads。

我们可以等效的方式去完成辅助接口提供的功能。譬如，你可以不使用链表库，而是使用自己实现的链表操作函数，你也可以不使用工作队列而使用内核线程来进行延后的工作。但是，使用标准的内核辅助接口的好处是，它可以简化你的代码、消除冗余、增强代码的可读性，并对长期维护有利。

由于内核非常庞大，你总是能找到没有利用这些辅助机制优点的代码，因此，更新这些代码也许是一种好的开始给内核开发贡献代码的方式。

链表

为了组成双向链表，可以使用 include/linux/list.h 文件中提供的函数。首先，你需要在你的数据结构中嵌套一个 list_head 结构体：

```
#include <linux/list.h>


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


struct mydatastructure {
  struct list_head mylist; /* Embed */
  /* ... */             /* Actual Fields */
};
```

mylist 是用于链接 mydatastructure 多个实例的链表。如果你在 mydatastructure 数据结构内嵌入了多个链表头，mydatastructure 就可以被链接到多个链表中，每个 list_head 用于其中的一个链表。你可以使用链表库函数来在链表中增加和删除元素。

在进入细节的讨论之前，我们先总结以下链表库说提供的链表操作接口，如表 3.1 所示。

表 3.1 链表操作函数

函数	作用
INIT_LIST_HEAD()	初始化表头
list_add()	在表头后增加一个元素
list_add_tail()	在链表尾部增加一个元素

函数	作用
list_del()	从链表中删除一个元素
list_replace()	用另一个元素替代链表中的某一元素
list_entry()	遍历链表中的所有结点
list_for_each_entry()/ list_for_each_entry_safe()	被简化的链表递归接口
list_empty()	检查链表是否为空
list_splice()	将 2 个链表联合起来

为了论证链表的用法，我们来实现一个实例。这个实例也可以为理解下一节要讨论的工作队列的概念打下基础。假设你的内核驱动程序需要从某个入口点开始执行一个艰巨的任务，譬如让当前线程进入睡眠等待状态。在该任务完成之前，你的驱动不想被阻塞，因为这会降低依赖于它的应用程序的响应速度。因此，当驱动需要执行这种工作的时候，它将相应的函数加入一个工作函数的链表，并延后执行它。而实际的工作则在一个内核线程中执行，该线程会遍历链表，并在后台执行这些工作函数。驱动将工作函数放入链表的尾部，而内核则从链表的头部取元素，因此，则包装了这些放入队列的工作会以先进先出的原则执行。当然，驱动的剩余部分需要被修改以适应这种延后执行的策略。在理解这个例子之前，你首先要意识到在清单 3.5 中，我们将使用工作队列（work queue）接口来完成相同的工作，而且用工作队列的方式会显得更加简单。

首先看看本例中使用的关键的数据结构：

```
static struct _mydrv_wq {
  struct list_head mydrv_worklist; /* Work List */
  spinlock_t lock;              /* Protect the list */
```

```
  wait_queue_head_t todo;        /* Synchronize submitter
                                    and worker */
} mydrv_wq;



struct _mydrv_work {
  struct list_head mydrv_workitem; /* The work chain */
  void (*worker_func)(void *);    /* Work to perform */
  void *worker_data;             /* Argument to worker_func */
  /* ... */                      /* Other fields */
} mydrv_work;
```

mydrv_wq 是一个针对所有工作发布者的全局变量。其成员包括一个指向工作链表头部的指针、一个用于在发起工作的驱动函数和执行该工作的线程之间进行通信的等待队列。链表辅助函数不会对链表成员的访问进行保护，因此，你需要使用并发机制以串行化同时发生的指针引用。mydrv_wq 的另一个成员——自旋锁用于此目的。清单 3.2 中的驱动初始化函数 mydrv_init()会初始化自旋锁、链表头、等待队列，并启动工作者线程。

清单 3.2 初始化数据结构

```
static int __init
mydrv_init(void)
{
  /* Initialize the lock to protect against
     concurrent list access */
  spin_lock_init(&mydrv_wq.lock);

  /* Initialize the wait queue for communication
     between the submitter and the worker */
```

```
    init_waitqueue_head(&mydrv_wq.todo);



    /* Initialize the list head */

    INIT_LIST_HEAD(&mydrv_wq.mydrv_worklist);



    /* Start the worker thread. See Listing 3.4 */

    kernel_thread(mydrv_worker, NULL,

                CLONE_FS | CLONE_FILES | CLONE_SIGHAND | SIGCHLD);

    return 0;

}
```

在查看工作者线程（执行被提交的工作）之前，我们先看看提交者本身。清单 3.3 给出一个函数，内核的其他部分可以利用它来提交工作。该函数调用 list_add_tail()来将一个工作函数添加到链表的尾部，图 3.1 显示了工作队列的物理结构。

图 3.1 工作函数链表

<!--[if !vml]-->

<!--[endif]-->

清单 3.3 提交延后执行的工作

```
int
```

```
submit_work(void (*func)(void *data), void *data)
{
  struct _mydrv_work *mydrv_work;

  /* Allocate the work structure */
  mydrv_work = kmalloc(sizeof(struct _mydrv_work), GFP_ATOMIC);
  if (!mydrv_work) return -1;

  /* Populate the work structure */
  mydrv_work->worker_func = func; /* Work function */
  mydrv_work->worker_data = data; /* Argument to pass */
  spin_lock(&mydrv_wq.lock);      /* Protect the list */

  /* Add your work to the tail of the list */
  list_add_tail(&mydrv_work->mydrv_workitem,
                &mydrv_wq.mydrv_worklist);

  /* Wake up the worker thread */
  wake_up(&mydrv_wq.todo);

  spin_unlock(&mydrv_wq.lock);
  return 0;
}
```

下面的代码用于从一个驱动的入口点提交一个 void job(void *)工作函数：

```
submit_work(job, NULL);
```

在提交这个工作函数之后，清单 3.3 将唤醒工作者线程。清单 3.4 中工作者线程的结构与上一节讨论的标准的内核线程结构相似。该线程调用 list_entry()以遍历链表中的所有结点，list_entry()返回包含链表结点的容器的数据结构。仔细查看清单 3.4 中如下一行：

```
mydrv_work = list_entry(mydrv_wq.mydrv_worklist.next,
                        struct _mydrv_work, mydrv_workitem);
```

mydrv_workitem 被包含在 mydrv_work 之中，因此 list_entry()返回相应的 mydrv_work 结构体的指针。传递给 list_entry()的参数是被嵌入链表结点的地址、容器结构体的类型、嵌入的链表结点字段的名字。

在执行一个被提交的工作函数之后，工作者线程将通过 list_del()删除链表中的相应结点。注意当工作函数被执行之前，mydrv_wq.lock 被释放，执行完后又被重新获得。这是因为工作函数可以睡眠，而且新的被调度执行的代码可能要获取相同的自旋锁，则可能导致潜在的死锁问题。

清单 3.4 工作者线程

```
static int
mydrv_worker(void *unused)
{

  DECLARE_WAITQUEUE(wait, current);
  void (*worker_func)(void *);
  void *worker_data;
  struct _mydrv_work *mydrv_work;

  set_current_state(TASK_INTERRUPTIBLE);

  /* Spin until asked to die */
  while (!asked_to_die()) {
```

```
add_wait_queue(&mydrv_wq.todo, &wait);


if (list_empty(&mydrv_wq.mydrv_worklist)) {
  schedule();
  /* Woken up by the submitter */
} else {
  set_current_state(TASK_RUNNING);
}
remove_wait_queue(&mydrv_wq.todo, &wait);


/* Protect concurrent access to the list */
spin_lock(&mydrv_wq.lock);


/* Traverse the list and plough through the work functions
   present in each node */
while (!list_empty(&mydrv_wq.mydrv_worklist)) {


  /* Get the first entry in the list */
  mydrv_work = list_entry(mydrv_wq.mydrv_worklist.next,
                    struct _mydrv_work, mydrv_workitem);
  worker_func = mydrv_work->worker_func;
  worker_data = mydrv_work->worker_data;


  /* This node has been processed. Throw it
     out of the list */
  list_del(mydrv_wq.mydrv_worklist.next);
```

```
        kfree(mydrv_work);   /* Free the node */


        /* Execute the work function in this node */
        spin_unlock(&mydrv_wq.lock);  /* Release lock */
        worker_func(worker_data);
        spin_lock(&mydrv_wq.lock);    /* Re-acquire lock */
      }
      spin_unlock(&mydrv_wq.lock);
      set_current_state(TASK_INTERRUPTIBLE);
    }


    set_current_state(TASK_RUNNING);
    return 0;
  }
```

处于代码简洁的目的，上述例子代码并不执行错误处理。例如，如果清单 2 调用的 kernel_thread()执行失败，你需要释放相应的工作结构体的内存。另外，清单 3.4 中的 asked_to_die()也没有完成。在侦测到信号或收到了模块卸载时 release()函数发出的信息后，它都会导致循环终止。

在本节结束之前，我们看一下另一个有用的链表库方法 list_for_each_entry()。使用这个宏，遍历操作就变地更为简洁，可读性也会更好，因为我们不必在循环内部调用 list_entry()。如果你想在循环内部删除链表元素，需要调用 list_for_each_entry_safe()。因为，我们可以将清单 3.4 中的下列代码：

```
  while (!list_empty(&mydrv_wq.mydrv_worklist)) {
    mydrv_work = list_entry(mydrv_wq.mydrv_worklist.next,
                    struct _mydrv_work, mydrv_workitem);
    /* ... */
  }
```

替换为：

```
struct _mydrv_work *temp;
list_for_each_entry_safe(mydrv_work, temp,
                &mydrv_wq.mydrv_worklist,
                mydrv_workitem) {
 /* ... */
}
```

在本例中，你不能使用 list_for_each_entry()，因为你需要在清单 3.4 的循环内部删除由 mydrv_work 指向的入口。而 list_for_each_entry_safe()则通过传入的第 2 个参数即临时变量 temp 解决了这个问题，其中，temp 用户保存链表中下一个入口的地址。

哈希链表

如果你需要实现哈希表链接数据结构，前文介绍的双向链表实现并非一种优化的方案。这是因为哈希表仅仅需要一个包含单一指针的链表头。为了减少此类应用的内存开销，内核提供哈希链表（hlist），它是链表的变体。链表对链表头和结点使用了相同的结构体，但是哈希链表则不一样，它针对链表头和链表结点有不同的定义：

```
struct hlist_head {
 struct hlist_node *first;
};

struct hlist_node {
 struct hlist_node *next, **pprev;
};
```

为了适应单一的哈希链表头指针这一策略，哈希链表结点会维护其前一个结点的地址，而不是它本身的指针。

哈希表的实现利用了 hlist_head 数组，每个 hlist_head 牵引着一个双向的 hlist_node 链表。而一个哈希函数则被用来定位目标结点在 hlist_head 数组中的索引，此后，

便可以使用 hlist 辅助函数（也定义在 include/linux/list.h 文件中）操作与选择的索引对应的 hlist_node 链表。fs/dcache.c 文件中目录高速缓冲（dcache）的实现可以作为一个例子。

工作队列

工作队列是内核中用于进行延后工作的一种方式[2]。延后工作在无数场景下都有用，例如：

[2] 软中断和 tasklet 是内核中用于延后工作的另外 2 种机制。第 4 章的表 4.1 对软中断、taskle 和工作队列进行了对比分析。

（1）1 个错误中断发生后，触发网络适配器重新启动；

（2）同步磁盘缓冲区的文件系统任务；

（3）给磁盘发送一个命令，并跟踪存储协议状态机。

工作队列的作用与清单 3.2 和 3.4 中的例子相似，但是，工作队列可以让你以更简洁的风格完成同样的工作。

工作队列辅助库向用户呈现了 2 个接口数据接口：workqueue_struct 和 work_struct，使用工作队列的步骤如下：

1. 创建一个工作队列（或一个 workqueue_struct），该工作队列与一个或多个内核线程关联。可以使用 create_singlethread_workqueue()创建一个服务于 workqueue_struct 的内核线程。为了在系统中的每个 CPU 上创建一个工作者线程，可以使用 create_workqueue()变体。另外，内核中也存在缺省的针对每个 CPU 的工作者线程（events/n，n 是 CPU 序号），可以分时共享这个线程而不是创建一个单独的工作者线程。根据具体应用的不同，如果你没有定义专用的工作者线程，可能会遭遇性能问题。

2. 创建一个工作元素（或者一个 work_struct）。 使用 INIT_WORK()可以初始化一个 work_struct，填充它的工作函数的地址和参数。

3. 将工作元素提交给工作队列。可以通过 queue_work()将 work_struct 提交给一个专用的 work_struct，或者通过 schedule_work()提交给缺省的内核工作

者线程。

接下来我们重新编写清单 3.2 和 3.4，以利用工作队列接口的优点，相关的代码为清单 3.5。原先的内核线程连同自旋锁、等待队列，在使用工作队列接口后都随风飘散了。如果你正在使用缺省的工作者线程的话，对 create_singlethread_workqueue()的调用都可以不需要。

清单 3.5 使用工作队列进行延后工作

```
#include <linux/workqueue.h>


struct workqueue_struct *wq;


/* Driver Initialization */
static int __init
mydrv_init(void)
{
 /* ... */
 wq = create_singlethread_workqueue("mydrv");
 return 0;
}
 /* Work Submission. The first argument is the work function, and
    the second argument is the argument to the work function */
 int
 submit_work(void (*func)(void *data), void *data)
 {
  struct work_struct *hardwork;


  hardwork = kmalloc(sizeof(struct work_struct), GFP_KERNEL);
```

```
    /* Init the work structure */
    INIT_WORK(hardwork, func, data);

    /* Enqueue Work */
    queue_work(wq, hardwork);
    return 0;
}
```

如果你使用了工作队列，你必须将对应模块设为 GPL 许可证，否则会出现连接错误。这是因为，内核仅仅将这些函数导出给 GPL 授权的代码。如果你查看内核工作队列的实现代码，你将发现如下的限制表达式：

```
EXPORT_SYMBOL_GPL(queue_work);
```

下列语句可用于宣布你的模块使用 GPL copyleft：

```
MODULE_LICENSE("GPL");
```

通知链

通知链（Notifier chains）可用于将状态改变信息发送给请求这些改变的代码段。与硬编码不同，notifier 提供了一种在感兴趣的事件产生时获得警告的技术。Notifier 的初始目的是将网络事件传递给内核中感兴趣的部分，但是现在也可用于许多其他目的。内核已经为主要的事件预先定义了 notifier。这样的通知的实例包括：

（1）死亡通知。当内核触发了一个陷阱和错误（由 oops、缺页或断点命中引发）时被发送。例如，如果你正在为一个医疗等级卡编写设备驱动，你可能需要注册自身接受死亡通知，这样，当内核恐慌发生时，你可以关闭医疗电子。

（2）网络设备通知。当一个网络接口卡启动和关闭的时候被发送。

（3）CPU 频率通知。当处理器的频率发生跳变的时候，会分发这一通知。

（4）Internet 地址通知。当侦测到网络接口卡的 IP 地址发送改变的时候，会发送此通知。

Notifier 的应用实例是 drivers/net/wan/hdlc.c 中的高级数据链路控制（HDLC）协议驱动，它会注册自己到网络设备通知链，以侦测载波状态的改变。

为了将你的代码与某通知链关联，你必须注册一个相关链的时间处理函数。当相应的事件发生时，事件 ID 和与通知相关的参数会传递给该处理函数。为了实现一个自定义的通知链，你必须另外实现底层结构，以便当事件被侦测到时，链会被激活。

清单 3.6 给出了使用预定义的通知和用户自定义通知的例子，表 3.2 则对清单 3.6 中的通知链和它们传递的事件进行了简要的描述，因此，可以对照查看表 3.2 和清单 3.6。

表 3.2 通知链和它们传送的事件

通知链	描述

Die Notifier Chain (die_chain)	通过register_die_notifier()，my_die_event_handler()被依附于die_chain死亡通知链。为了触发my_die_event_handler()的发生，代码中引入了一个冗余的引用，即：
	int *q = 0;
	*q = 1;
	当这段代码被执行的时候，my_die_event_handler()将被调用，你将看到这样的信息：
	my_die_event_handler: OOPs! at EIP=f00350e7
	死亡事件通知将die_args结构体传给被注册的事件处理函数。该参数包括一个指向regs结构体的指针（在发生缺陷的时候，用于存放处理器的寄存器）。my_die_event_handler()中打印了指令指针寄存器的内容。

Netdevice Notifier Chain(netdev_chain)	通过register_netdevice_notifier()，my_dev_event_handler()被依附于网络设备通知链netdev_chain。通过改变网络接口设备（如以太网ethX和回环设备lo）的状态可以产生此事件：

表 3.2 通知链和它们传送的事件

通知链	描述
	bash> ifconfig eth0 up 它会导致my_dev_event_handler()的执行。 net_device结构体的指针被传给该处理函数作为参数，它包含了网络接口的名字，my_dev_event_handler()打印出了该信息： my_dev_event_handler: Val=1, Interface=eth0 Val=1 意味着NETDEV_UP事件，其定义在include/linux/notifier.h文件中。
User-Defined Notifier Chain	清单 3.6 也实现了一个用户自定义的通知链my_noti_chain。假定你希望当用户读取proc文件系统中一个特定的文件的时候该事件被产生，可以在相关的procfs读函数中加入如下代码： blocking_notifier_call_chain(&my_noti_chain, 100, NULL); 当你读取相应的/proc文件时，my_event_handler()将被调用，如下信息被打印出来： my_event_handler: Val=100 Val包含了产生事件的ID，本例中为 100。该函数的参数没有被使用。

清单 3.6 通知事件处理函数

```
#include <linux/notifier.h>
#include <asm/kdebug.h>
#include <linux/netdevice.h>
#include <linux/inetdevice.h>
```

```
/* Die Notifier Definition */
static struct notifier_block my_die_notifier = {
  .notifier_call = my_die_event_handler,
};
/* Die notification event handler */
int
my_die_event_handler(struct notifier_block *self,
                     unsigned long val, void *data)
{
  struct die_args *args = (struct die_args *)data;

  if (val == 1) { /* '1' corresponds to an "oops" */
    printk("my_die_event: OOPs! at EIP=%lx\n", args->regs->eip);
  } /* else ignore */
  return 0;
}



/* Net Device notifier definition */
static struct notifier_block my_dev_notifier = {
  .notifier_call = my_dev_event_handler,
};



/* Net Device notification event handler */
```

```
int my_dev_event_handler(struct notifier_block *self,
                         unsigned long val, void *data)
{
  printk("my_dev_event: Val=%ld, Interface=%s\n", val,
         ((struct net_device *) data)->name);
  return 0;
}



/* User-defined notifier chain implementation */
static BLOCKING_NOTIFIER_HEAD(my_noti_chain);

static struct notifier_block my_notifier = {
  .notifier_call = my_event_handler,
};

/* User-defined notification event handler */
int my_event_handler(struct notifier_block *self,
                     unsigned long val, void *data)
{
  printk("my_event: Val=%ld\n", val);
  return 0;
}
/* Driver Initialization */
static int __init
my_init(void)
```

```
{
 /* … */

 /* Register Die Notifier */
 register_die_notifier(&my_die_notifier);

 /* Register Net Device Notifier */
 register_netdevice_notifier(&my_dev_notifier);

 /* Register a user-defined Notifier */
 blocking_notifier_chain_register(&my_noti_chain, &my_notifier);

 /* … */
}
```

通过 BLOCKING_NOTIFIER_HEAD()，清单 3.6 中的 my_noti_chain 被定义为一个阻塞通知，经由对 blocking_notifier_chain_register()函数的调用，它被注册。这意味着该通知事件处理函数总是在进程上下文被调用，也允许睡眠。如果你的通知处理函数允许从中断上下文调用，你应该使用 ATOMIC_NOTIFIER_HEAD()定义该通知链并使用 atomic_notifier_chain_register()注册它。

老的通知接口

早于 2.6.17 的内核版本仅支持一个通用目的的通知链。通知链注册函数 notifier_chain_register()内部使用自旋锁保护，但游走于通知链以分发事件给通知处理函数的函数notifier_call_chain()确是无锁的。不加锁的原因是事件处理函数可能会睡眠、在运行中注销自己或在中断上下文中被调用。但是无锁的实现却引入了竞态，而新的通知API则建立于老的接口之上，其设计中包含了克服此限制的意图。

完成接口

内核中的许多地方会激发一个单独的执行线索，之后等待它的完成。完成接口是一个充分的且简单的此类编码的实现模式。

一些使用场景的例子包括：

（1）你的驱动模块中包含了一个辅助内核线程。当你卸载这个模块时，在模块的代码从内核空间被移除之前，release()函数将被调用。release 函数中要求内核线程杀死自身，它一直阻塞等待线程的退出。清单 3.7 实现了这个例子。

（2）你正在编写块设备驱动（第 14 章《块设备驱动》讨论）中将设备读请求排队的部分。这激活了以单独线程或工作队列方式实现的一个状态机的变更，而驱动本身想一直等到该操作完成前才执行下一次操作。drivers/block/floppy.c 就是这样的一个例子。

（3）一个应用请求模拟/数字转换（ADC）驱动完成一次数据采样。该驱动初始化一个转换请求，接下来一直等待转换完成的中断产生，并返回转换后的数据。

清单 3.7 使用完成接口进行同步

```
static DECLARE_COMPLETION(my_thread_exit);      /* Completion */
static DECLARE_WAIT_QUEUE_HEAD(my_thread_wait); /* Wait Queue */
int pink_slip = 0;                              /* Exit Flag */


/* Helper thread */
static int
my_thread(void *unused)
{
  DECLARE_WAITQUEUE(wait, current);


  daemonize("my_thread");
  add_wait_queue(&my_thread_wait, &wait);


  while (1) {
```

```
    /* Relinquish processor until event occurs */
    set_current_state(TASK_INTERRUPTIBLE);
    schedule();
    /* Control gets here when the thread is woken
       up from the my_thread_wait wait queue */

    /* Quit if let go */
    if (pink_slip) {
      break;
    }
    /* Do the real work */
    /* ... */

  }

  /* Bail out of the wait queue */
  __set_current_state(TASK_RUNNING);
  remove_wait_queue(&my_thread_wait, &wait);

  /* Atomically signal completion and exit */
  complete_and_exit(&my_thread_exit, 0);
}

/* Module Initialization */
static int __init
my_init(void)
```

```
{
  /* ... */

  /* Kick start the thread */
  kernel_thread(my_thread, NULL,
                CLONE_FS | CLONE_FILES | CLONE_SIGHAND | SIGCHLD);

  /* ... */
}

/* Module Release */
static void __exit
my_release(void)
{
  /* ... */
  pink_slip = 1;                    /* my_thread must go */
  wake_up(&my_thread_wait);         /* Activate my_thread */
  wait_for_completion(&my_thread_exit); /* Wait until my_thread
                                    quits */
  /* ... */
}
```

可以使用 DECLARE_COMPLETION()静态地定义一个完成实例，或者使用 init_completion()动态地创建之。而一个执行线索可以使用 complete()或 complete_all()来标识一个完成。调用者自身则通过 wait_for_completion()等待完成。

在清单 3.7 中的 my_release()函数中，在唤醒 my_thread()之前，它通过 pink_slip 设置了一个退出请求标志。接下来，它调用 wait_for_completion()等待 my_thread()完成其退出。my_thread()函数醒来后，发现 pink_slip 被设置，它进行如下工作：

（1）向 my_release()函数通知完成；

（2）杀死自身

my_thread()使用 complete_and_exit()函数原子性地完成了这 2 个步骤。complete_and_exit()关闭了模块退出和线程退出之间的那扇窗，而如果使用 complete()和 exit()函数 2 步操作的话，此窗口则是开放的。

在第 11 章中，开发一个遥测设备驱动的时候，我们会使用完成接口。

Kthread辅助接口

Kthread 为原始的线程创建函数添加了一层外衣由此简化了线程管理的任务。

清单 3.8 使用 kthread 接口重写了清单 3.7。my_init()现在调用 kthread_create()而不是 kernel_thread()，你可以将线程的名字传入 kthread_create()，而不再需要明确地在线程内调用 daemonize()。

Kthread 允许你自由地调用内建的由完成接口所实现的退出同步机制。因此，如清单 3.8 中 my_release()函数所为，你可以直接调用 kthread_stop()而不再需要设置 pink_slip、唤醒 my_thread()并使用 wait_for_completion()等待它的完成。相似地，my_thread()可以进行一个简洁的对 kthread_should_stop()的调用以确认其是否应该退出。

清单 3.8 使用Kthread辅助接口完成同步

```
/* '+' and '-' show the differences from Listing 3.7 */


#include <linux/kthread.h>


/* Assistant Thread */
static int
my_thread(void *unused)
{
  DECLARE_WAITQUEUE(wait, current);
```

```
-   daemonize("my_thread");


-   while (1) {
+   /* Continue work if no other thread has
+    * invoked kthread_stop() */
+   while (!kthread_should_stop()) {
      /* ... */
-     /* Quit if let go */
-     if (pink_slip) {
-       break;
-     }
      /* ... */
    }
    __set_current_state(TASK_RUNNING);
    remove_wait_queue(&my_thread_wait, &wait);


-   complete_and_exit(&my_thread_exit, 0);
+   return 0;
 }


+   struct task_struct *my_task;


 /* Module Initialization */
 static int __init
 my_init(void)
 {
```

```
    /* ... */
-   kernel_thread(my_thread, NULL,
-                 CLONE_FS | CLONE_FILES | CLONE_SIGHAND |
                  SIGCHLD);
+   my_task = kthread_create(my_thread, NULL, "%s", "my_thread");
+   if (my_task) wake_up_process(my_task);

    /* ... */
 }

 /* Module Release */
 static void __exit
 my_release(void)
 {
    /* ... */
-   pink_slip = 1;
-   wake_up(&my_thread_wait);
-   wait_for_completion(&my_thread_exit);
+   kthread_stop(my_task);

    /* ... */
 }
```

代替使用 kthread_create()创建线程，接下来再使用 wake_up_process()激活它，你可以使用下面的一次调用达到目的：

```
kthread_run(my_thread, NULL, "%s", "my_thread");
```

错误处理助手

数个内核函数返回指针值。调用者通常将返回值与 NULL 对比以检查是否失败，但是它们很可能需要更多的信息以分析出确切的错误发生原因。由于内核地址有冗余比特，可以覆盖它以包含错误语义信息。一套辅助函数完成了此功能，清单 3.9 给出了一个简单的例子。

```
#include <linux/err.h>


char *
collect_data(char *userbuffer)
{


 char *buffer;


 /* ... */
 buffer = kmalloc(100, GFP_KERNEL);
 if (!buffer) { /* Out of memory */
  return ERR_PTR(-ENOMEM);
 }


 /* ... */
 if (copy_from_user(buffer, userbuffer, 100)) {
  return ERR_PTR(-EFAULT);
 }
 /* ... */


 return(buffer);
}
```

```
int
my_function(char *userbuffer)
{
  char *buf;

  /* ... */
  buf = collect_data(userbuffer);
  if (IS_ERR(buf)) {
    printk("Error returned is %d!\n", PTR_ERR(buf));
  }
  /* ... */

}
```

在清单 3.9 中，如果 collect_data()中的 kmalloc()失败，你将获得如下信息：

Error returned is -12!

但是，如果 collect_data()执行成功，它将返回一个数据缓冲区的指针。

再来一个例子，我们给清单 3.8 中的线程创建代码添加错误处理（使用 IS_ERR()和 PTR_ERR()）：

```
my_task = kthread_create(my_thread, NULL, "%s", "mydrv");

+  if (!IS_ERR(my_task)) {
+    /* Success */
     wake_up_process(my_task);
```

```
+ } else {
+   /* Failure */
+   printk("Error value returned=%d\n", PTR_ERR(my_task));
+ }
```

查看源代码

ksoftirqd、pdflush 和 khubd 内核线程代码分别在 kernel/softirq.c, mm/pdflush.c 和 drivers/usb/core/hub.c 文件中。

kernel/exit.c 可以找到 daemonize()，以用户模式助手的实现见于 kernel/kmod.c 文件。

list 和 hlist 库函数位于 include/linux/list.h。在整个类型中都有对它们的使用，因此在大多数子目录中，都能找到例子。其中的一个例子是 include/linux/blkdev.h 中定义的 request_queue 结构体，它存放磁盘 I/O 请求的链表。在第 14 章中我们会分析此数据结构。

查看www.ussg.iu.edu/hypermail/linux/kernel/0007.3/0805.html可以跟踪到Torvalds和Andi Kleen之间关于使用hlist实现list库的利弊的争论。

内核工作队列的实现位于 kernel/workqueue.c 文件，为了理解工作队列的用法，可以查看 drivers/net/wireless/ipw2200.c 中 PRO/Wireless 2200 网卡驱动。

内核通知链的实现位于 kernel/sys.c 和 include/linux/notifier.h 文件。查看 kernel/sched.c 和 include/linux/completion.h 文件可以挖掘完成接口的实现机理。kernel/kthread.c 包含了 kthread 辅助接口的源代码，include/linux/err.h 则包含了错误处理接口的定义。

表 3.3 给出了本章中所使用的主要的数据结构及其源代码路径的总结。表 3.4 列出了本章中使用的主要内核编程接口及其源代码路径。

表 3.3 数据结构总结

数据结构	路径	描述

数据结构	路径	描述
wait_queue_t	include/linux/wait.h	内核线程欲等待某事件或系统资源时使用
list_head	include/linux/list.h	用于构造双向链表数据结构的内核结构体
hlist_head	include/linux/list.h	用于实现哈希表的的内核结构体
work_struct	include/linux/workqueue.h	实现工作队列，它是一种在内核中进行延后工作的方式
notifier_block	include/linux/notifier.h	实现通知链，用于将状态变更信息发生给请求此变更的代码段
completion	include/linux/completion.h	用于开始某线程活动并等待它们完成

表 3.4 内核编程接口总结

内核接口	路径	描述
DECLARE_WAITQUEUE()	include/linux/wait.h	定义一个等待队列
add_wait_queue()	kernel/wait.c	将一个任务加入一个等待队列。该任务进入睡眠状态，知道它被另一个线程或中断处理函数唤醒。
remove_wait_queue()	kernel/wait.c	从等待队列中删除一个任务。

内核接口	路径	描述
wake_up_interruptible()	include/linux/wait.h kernel/sched.c	唤醒一个正在等待队列中睡眠的任务，将其返回调度器的运行队列。
schedule()	kernel/sched.c	放弃 CPU，让内核的其他部分运行。
set_current_state()	include/linux/sched.h	设置一个进程的运行状态，可以是如下状态中的一种：TASK_RUNNING、TASK_INTERRUPTIBLE、TASK_UNINTERRUPTIBLE、TASK_STOPPED、TASK_TRACED、EXIT_ZOMBIE或EXIT_DEAD.
kernel_thread()	arch/your-arch/kernel/process.c	创建一个内核线程
daemonize()	kernel/exit.c	激活一个内核线程（未依附于用户资源），并将调用线程的父线程改为kthreadd。
allow_signal()	kernel/exit.c	使能某指定信号的发起
signal_pending()	include/linux/sched.h	检查是否有信号已经被传输。在内核中没有信号处

内核接口	路径	描述
		理函数，因此，你不得不显示地检查信号的发起
call_usermodehelper()	include/linux/kmod.h kernel/kmod.c	执行一个用户模式的程序
Linked list library functions	include/linux/list.h	看表 3.1
register_die_notifier()	arch/your-arch/kernel/traps.c	注册一个 die 通知
register_netdevice_notifier()	net/core/dev.c	注册一个 netdevice 通知
register_inetaddr_notifier()	net/ipv4/devinet.c	注册一个 inetaddr 通知
BLOCKING_NOTIFIER_HEAD()	include/linux/notifier.h	创建一个用户自定义的阻塞性的通知
blocking_notifier_chain_register()	kernel/sys.c	注册一个阻塞性的通知
blocking_notifier_call_chain()	kernel/sys.c	将事件分发给一个阻塞性的通知链
ATOMIC_NOTIFIER_HEAD()	include/linux/notifier.h	创建一个原子性的通知
atomic_notifier_chain_register()	kernel/sys.c	注册一个原子性的通知
DECLARE_COMPLETION()	include/linux/completion.h	静态定义一个完成实例

内核接口	路径	描述
init_completion()	include/linux/completion.h	动态定义一个完成实例
complete()	kernel/sched.c	宣布完成
wait_for_completion()	kernel/sched.c	一直等待完成实例的完成
complete_and_exit()	kernel/exit.c	原子性的通知完成并退出
kthread_create()	kernel/kthread.c	创建一个内核线程
kthread_stop()	kernel/kthread.c	让一个内核线程停止
kthread_should_stop()	kernel/kthread.c	内核线程可以使用该函数轮询是否其他的执行单元已经调用kthread_stop()让其停止
IS_ERR()	include/linux/err.h	查看返回值是否是一个出错码