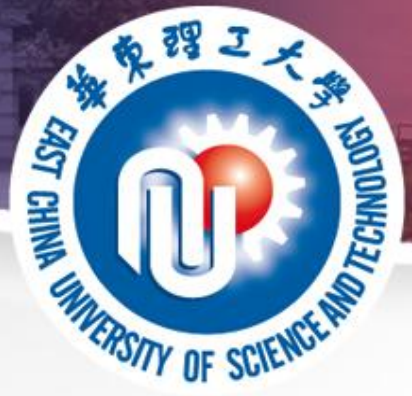


第九章　设备驱动

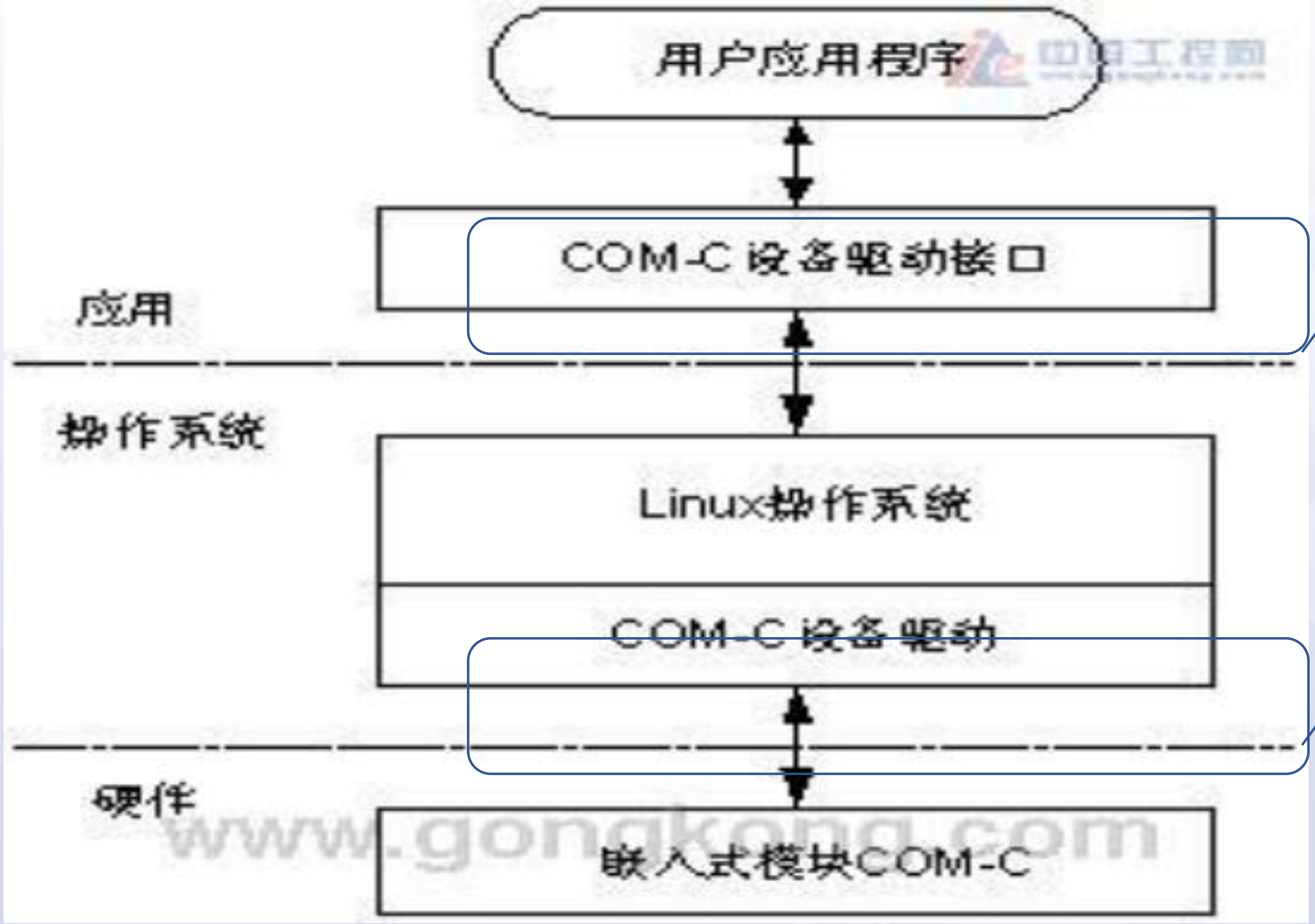
华东理工大学计算机系

罗　飞

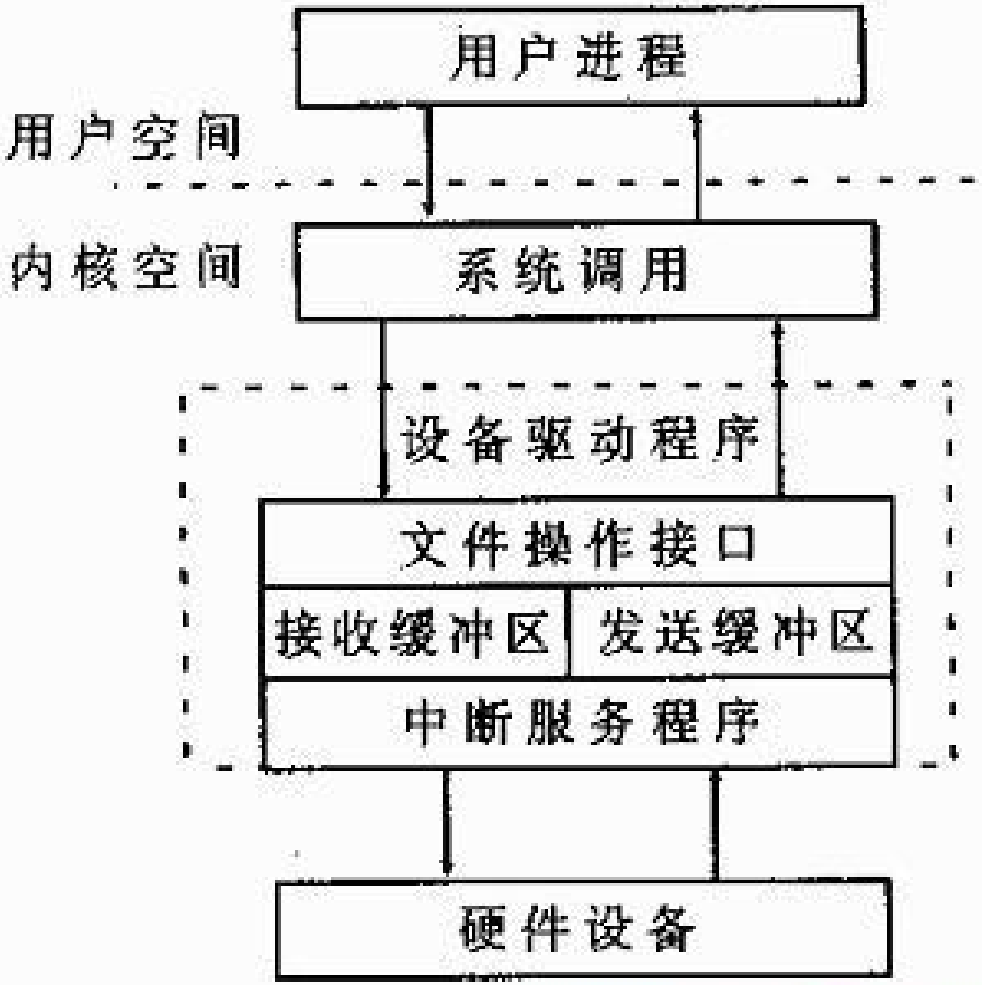
Content

- Linux设备驱动简介
- Linux驱动相关内核机制

Linux设备驱动模块



驱动接口



Linux 系统设备驱动程序的结构

Linux驱动程序概述

- 操作系统内核和机器硬件之间的接口
- 为应用程序屏蔽了硬件的细节
- 硬件设备→设备文件→普通文件
  - 设备操作

Linux设备驱动程序简介

- 设备驱动功能
- 设备分类
- 设备文件和设备号
- 代码分布

设备驱动功能

- 设备驱动程序是内核的一部分
  - 对设备的初始化和释放
  - 把数据从内核传到硬件/从硬件读数据到内核
  - 读取应用程序传送给设备文件的数据和回送应用程序请求的数据。这需要在用户空间，内核空间，总线以及外设之间传输数据
  - 检测和处理设备出现的错误

设备分类

- Linux支持三类硬件设备
  - 字符设备：无需缓冲直接读写
  - 块设备
    - 通过buffer或cache进行读写
    - 支持随机访问
  - 网络设备：通过BSD套接口访问

设备文件和设备号

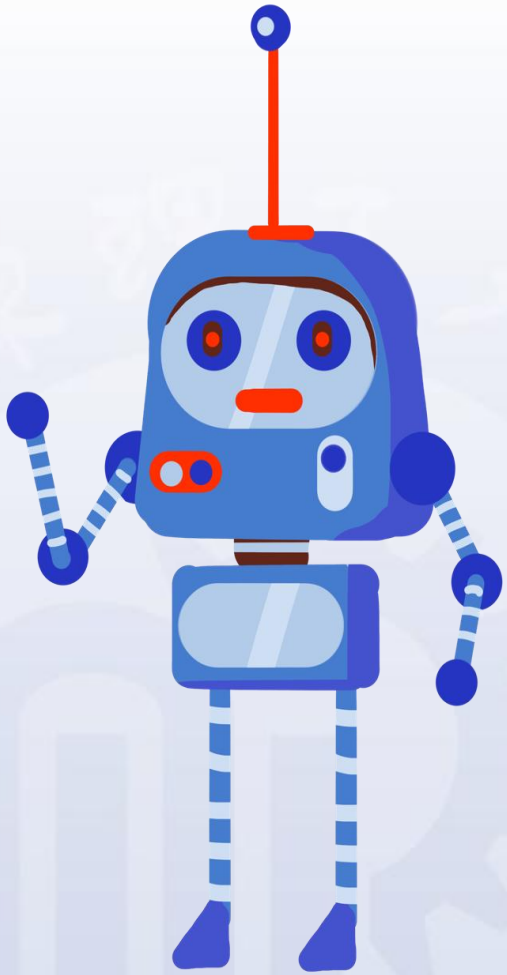
- 设备文件
  - Linux抽象了对硬件设备的访问，可作为普通文件一样访问，使用和操作文件相同的、标准的系统调用接口来完成打开、关闭、读写和I/O控制操作
- 每个设备文件都对应有两个设备号
  - 主设备号，标识设备的种类，使用的驱动程序
  - 次设备号，标识使用同一设备驱动程序的不同硬件设备

代码分布

- Drivers
  - block
  - char
  - cdrom
  - pci
  - scsi
  - net
  - sound

Linux驱动相关内核机制

- 设备模型概述
- sysfs
- 设备驱动模型
- 关键数据结构
- 同步机制
- 工作队列
- 异步IO



设备模型概述

- 设备模型提供独立的机制表示设备及其在系统中的拓扑结构
  - 代码重复最小
  - 提供如引用计数这样的统一管理机制
  - 例举系统中所有设备，观察其状态，查看其连接总线
  - 用树的形式将全部设备结构完整、有效地展现，包括所有总线和内部连接
  - 将设备和对应驱动联系起来
  - 将设备按类型分类
  - 从树的叶子向根的方向依次遍历，确保以正确顺序关闭各个设备的电源

sysfs（1）

- sysfs是设备拓扑结构的文件系统表现
  - block：每个子目录分别对应系统中的一个块设备，每个目录又都包含该块设备的所有分区
  - bus：内核设备按总线类型分层放置的目录结构，devices中的所有设备都是连接于某种总线之下可以找到每一个具体设备的符号链接
  - class：包含以高层功能逻辑组织起来的系统设备视图
  - dev：维护一个按字符设备和块设备的主次号码（major/minor）链接到真实的设备（devices下）的符号链接

sysfs（2）

- devices：系统设备拓扑结构视图，直接映射出内核中设备结构体的组织层次
- firmware：包含一些如ACPI，EDD，EFI等底层子系统的特殊树
- fs：存放的已挂载点，但目前只有fuse，gfs2等少数文件系统支持sysfs接口，传统的虚拟文件系统（VFS）层次控制参数仍然在sysctl（/proc/sys/fs）接口
- kernel：新式的slab分配器等几项较新的设计在使用它，其它内核可调整参数仍然位于sysctl（/proc/sys/kernel）接口中
- module：系统中所有模块的信息，不管这些模块是以内联（inlined）方式编译到内核映像文件（vmlinuz）还是编译到外部模块（ko文件）

设备模型基本元素

- 设备类结构classes
- 总线结构bus
- 设备结构devices
- 驱动结构drivers

Linux统一设备模型的基本结构

类型	说明	对应内核数据结构	对应/sys项
总线类型（Bus Types）	系统中用于连接设备的总线	struct bus_type	/sys/bus/*/
设备（Devices）	内核识别的所有设备，依照连接它们的总线进行组织	struct device	/sys/devices/*/*/../
设备类别（Device Classes）	系统中设备的类型（声卡，网卡，显卡，输入设备等），同一类中包含的设备可能连接不同的总线	struct class	/sys/class/*/
设备驱动（Device Drivers）	在一个系统中安装多个相同设备，只需要一份驱动程序的支持	struct device_driver	/sys/bus/pic/drivers/*/

关键数据结构——kobject

```
struct kobject{
    const char *name;/*短名字*/
    struct kobject *parent;/*表示对象的层次关系*/
    struct sysfs_dirent  *sd;/*表示sysfs中的一个目录项*/
    struct kref kref;/*提供一个统一的计数系统*/
    struct list_head entry;
    struct kset *kset;
    struct kobj_type *ktype;
};
```

关键数据结构——kref

- 定义

```
struct kref{
        atomic_t refcount;
}
```

- 初始化

```
void kref_init(struct kref *kref)
```

- 增加计数

```
struct kobject* kobject_get(struct kobject *kobj)
```

- 减少计数

```
void kobject_put(struct kobject *kobj)
```

关键数据结构——ktype

```
struct kobj_type{
    void (*release)(struct kobject *kobj);  /*析构函数*/
    struct sysfs_ops *sysfs_ops;
        /*包含对属性进行操作的读写函数的结构体*/
    struct attribute **default_attrs;
        /*定义了kobject相关的默认属性*/
}
```

关键数据结构——kset

```
struct kset{
    struct list_head list;
        /*在该kset下的所有kobject对象*/
    spinlock_t list_lock;
        /*在kobject上进行迭代时用到的锁*/
    struct kobject kobj;
    /*该指针指向的kobject对象代表了该集合的基类*/
    struct kset_uevent_ops *uevent_ops; /*指向一个用于处理集合中kobject对象的热插拔结构操作的结构体*/
}
```

同步机制

- 同步锁
- 信号量
- 读写信号量
- 原子操作
- 完成事件（completion）
- 时间

同步锁 — 自旋锁Spinlock（1）

- 自旋锁被别的执行单元保持，调用者就一直循环，看是否该自旋锁的保持者已经释放了锁。

- 自旋锁和互斥锁的区别是，自旋锁不会引起调用者睡眠，自旋锁使用者一般保持锁事件非常短，所以选择自旋而不是睡眠，效率会高于互斥锁。

同步锁 — 自旋锁Spinlock（2）

- **自旋锁的类型** spinlock_t
- **初始化**

  spinlock_t my_spinlock =
  SPIN_LOCK_UNLOCKED;
  void spin_lock_init(spinlock_t *lock);
- **获得锁** void spin_lock(spinlock_t *lock);
- **释放锁** void spin_unlock(spinlock_t *lock);
- **非阻塞版本**

  int spin_trylock(spinlock_t *lock);
  int spin_trylock_bh(spinlock_t *lock);

同步锁 — 读写锁（1）

- rmlock是内核提供的一个自旋锁的读者/写者形式。
- 读者/写者形式的锁允许任意数目的读者同时进入临界区，但是写者必须是排他的。
- 读写锁类型是rwlock_t，在<linux/spinlock.h>

同步锁 — 读写锁（2）

- **初始化**

  rwlock_t my_rwlock=RW_LOCK_UNLOCKED;/*静态*/

  rwlock_init(&my_rwlock); /* 动态 */

- **读者锁的获取和释放**

  void read_lock(rwlock_t *lock);

  void read_unlock(rwlock_t *lock);

- **写者锁的获取和释放**

  void write_lock(rwlock_t *lock);

  int write_trylock(rwlock_t *lock);

  void write_unlock_bh(rwlock_t *lock);

同步锁 — RCU锁（1）

- RCU（Read-Copy Update）锁机制是Linux2.6内核中新的锁机制
- 高性能的锁机制RCU克服了获得锁的开销和访问内存速度挂钩的问题
- RCU是改进的读写锁
  - 读者锁基本上没有同步开销，不需要锁，不使用原子指令，死锁问题也不用考虑
  - 写者锁同步开销相对较大，因为它需要延迟数据结构的释放，复制被修改的数据结构，也必须用某种锁机制同步并行的其它写者的修改操作

同步锁 — RCU锁（2）

- 加锁

  rcu_read_lock()

- 释放锁

  rcu_read_unlock()

- 同步RCU锁

  synchronize_rcu()

此函数由RCU写端调用，会阻塞写者，直到所有读者完成读端临界区，写者才能进行后续操作。

同步锁 — seqlock（1）

- seqlock是2.6内核包含的一对新机制，能够快速地、无锁地存取一个共享资源
- seqlock实现原理是依赖一个序列计数器
  - 当写者写入数据的时候，会得到一把锁，并且把序列值增加1。当读者读取数据之前和之后，这个序列号都会被读取，如果两次读取的序列号相同，则说明写没有发生
  - 如果表明发生过写事件，则放弃已经进行的操作，重新循环一次，一直到成功。

同步锁 — seqlock（2）

- 读者锁

```
unsigned int seq;
do {
    seq = read_seqbegin(&the_lock);
    /* Do what you need to do */
} while read_seqretry(&the_lock, seq);
```

- 写者锁由自旋锁实现

```
void write_seqlock(seqlock_t *lock);
void write_sequnlock(seqlock_t *lock);
```

信号量（1）

- Linux内核的信号量在概念和原理上与用户态的IPC机制信号量是一样的，是一种睡眠锁
  - 当一个任务试图获得已被占用的信号量时，会进入一个等待队列，然后睡眠
  - 当持有该信号量的进程释放信号量后，位于等待队列的一个任务就会被唤醒，这个任务获得信号量
- 信号量不会禁止内核抢占
- 信号量适用于锁会被长期持有的情况，而自旋锁比较适合被短期持有
- 信号量允许有多个持有者，而自旋锁任何时候只能有一个持有者

信号量（2）

- 宏声明一个信号量
  - DECLARE_MUTEX(name)
  - DECLARE_MUTEX_LOCKED(name)
- 声明和初始化互斥锁
  - void sema_init(struct semaphore *sem, int val);
  - void init_MUTEX(struct semaphore *sem);
  - void init_MUTEX_LOCKED(struct semaphore *sem);
- 获得/释放信号量
  - void down(struct semaphore *sem);
  - int down_trylock(struct semaphore *sem);
  - void up(struct semaphore *sem);

读写信号量（1）

- 读写信号量的访问者被细分为两类，一种是读者，另一种是写者。
  - 读者在拥有读写信号量期间，对该读写信号量保护的共享资源只能进行读访问
  - 如果某个任务同时需要读和写，则被归类为写者，它在对共享资源访问之前须先获得写者身份，写者在不需要写访问的情况下将被降级为读者。
- 可以有任意多个读者同时拥有一个读写信号量
  - 适合于读多写少
- 写者具有排他性和独占性

读写信号量（2）

- 声明和初始化
  DELARE_RWSEM(sem)
  void init_rwsem(struct rm_semaphore *sem);
- 读操作
  void down_read(struct rw_semaphore *sem);
  int down_read_trylock(struct rw_semaphore *sem);
- 写操作
  void down_write(struct rw_semaphore *sem);
  int down_write_trylock(struct rw_semaphore *sem);
  void up_write(struct rw_semaphore *sem);
- 写者降级为读者
  void downgrade_write(struct rw_semaphore *sem);

原子操作（1）

- 原子操作是指该操作在执行完毕前绝不会被任何其他任务或时间打断，是最小的执行单位
- 原子操作和架构有关，需要硬件的支持，它的API和原子类型的定义都在内核源码树include/asm/atomic.h文件中，使用汇编语言实现
- **主要用在资源计数**，很多应用计数（refcnt）就是通过原子操作实现的

原子操作（2）

- 原子操作类型的定义

  typedef struct {volatile int counter;}atomic_t;

- 原子操作API

  atomic_read(atomic_t *v);

  atomic_set(atomic_t *v,int i);

  void atomic_add(int I, atomic_t *v);

  void atomic_sub(int I,atomic_t *v);

  void atomic_inc(atomic_t *v);

  void atomic_dec(atomic_t *v);

  …

完成事件（1）

- 完成事件是一种简单的同步机制，表示“things may proceed”
  - **适用于需要睡眠和唤醒的情景**
  - 如果要在任务中实现简单睡眠直到其它进程完成某些处理过程为止，可采用完成事件，不会引起资源竞争

- 如果要使用completion，需要包含<linux/completion.h>，同时创建类型为struct completion的变量

完成事件（2）

- 静态声明和初始化

  DECLARE_COMPLETION(my_completion);

- 动态声明和初始化

  struct completion my_completion;

  init_completion(&my_compleiton);

- 等待某个过程的完成

  void wait_for_completion(struct completion *comp);

- 唤醒等待该事件的进程

  void complete(struct completion *comp);

  void complete_all(struct completion *comp);

同步机制—时间

- 测量时间流失（jiffies）
- 获知当前时间
- 延后执行
- 内核定时器

jiffies

- 通过定时器中断跟踪时间的流动
  - 根据Hz值来编程
- jiffies计数器
  - 64位变量

获知当前时间

- 通过查看jiffies的值来获取当前时间
- 通过jiffies的当前值来计算时间之间的时间间隔

延后执行

- 长延时
  - 多于一个时钟
  - 监控jiffies计数器的循环
- 短延时
  - 几个毫秒
  - 内核函数：ndelay, udelay, mdelay
    - 具体实现与硬件有关
  - msleep, ssleep, msleep_interruptible

内核定时器

- 告诉内核在用户定义的时间点
  - 使用用户定义的参数来执行用户定义的函数

- timer_list结构

内存映射和管理--物理地址映射到虚拟地址

- 每一种外设的访问都是通过读写设备上的寄存器来进行
  - 包控制寄存器、状态寄存器和数据寄存器三大类
- CPU对I/O端口的编址方式有两种
  - I/O映射方式
  - 内存映射方式

I/O内存资源访问

- 运行时，不能直接使用物理地址
- 通过页表映射到核心虚拟地址空间

```
Void *ioremap( ... )
Void iounmap(void * addr)
```

- I/O资源读写
  - 特定的I/O读写函数

内存映射和管理--内核空间映射到用户空间

- 对于驱动来说，内存映射可用来提供用户程序对设备内存的直接存取
- 采用mmap方法在用户空间访问内核地址
- 用户空间的应用程序通过映射可以直接访问设备的I/O存储区或DMA缓冲
- 映射一个设备是指关联一些用户空间地址到设备内存

mmap

- 串口或其他面向流的设备无需映射
- 映射粒度
  - PAGE_SIZE
- file_operations结构的一部分

```
mmap( caddr_t addr, size_t len, int prot, int flags, int fd, off_t offset)
```

- 文件操作

作用?

```
int    (*mmap) (struct file * filp, struct vm_area_struct *vma)
```

建立页表

- 驱动程序为vma建立页表

- 两种方法
  - remap_pfn_range函数
    - 一次建立所有页表
  - nopage VMA方法
    - 每次只建立一个页表

工作队列（work queue）

- linux内核将工作推后执行的机制
- 工作队列把推后的工作交给内核线程去执行
- 优势：允许重新调度甚至睡眠
- 2.6内核开始引入工作队列

工作队列—新版本work_struct

- 2.6.20之后的版本

- 解决问题
  - 将用户的数据作为参数传给func

  - 实现延迟工作—delayed_work

工作队列

- 数据结构：

```
struct work_struct {
        atomic_long_t data;
        struct list_head entry;
        work_func_t func;
};
```

工作队列

- 初始化

  INIT_WORK(_work, _func, _data)

- 使用

  int schedule_work(struct work_struct *work);

  int schedule_delayed_work(struct work_struct *work, unsigned long delay);

  void flush_scheduled_work(void);

  int cancel_delayed_work(struct work_struct *work);

- 创建和调度工作队列

  struct workqueue_struct *create_workqueue(const char *name);

  int queue_work(struct workqueue_struct *wq, struct work_struct *work);

  …

异步I/O

- 异步事件非阻塞I/O，也叫做异步I/O(AIO)
  - 用户程序可以通过向内核发出I/O请求命令，不用等待I/O事件真正发生，可以继续做另外的事情，等I/O操作完成，内核会通过函数回调或者信号机制通知用户进程
  - 很大程度提高了系统吞吐量
- 要使用AIO功能，需要包含头文件aio.h

异步I/O（AIO）几个阶段

- (1)同步阻塞I/O
  - 用户进程进行I/O操作，一直阻塞到I/O操作完成为止
- (2)同步非阻塞I/O
  - 用户程序可以通过设置文件描述符的属性O_NONBLOCK，I/O操作可以立即返回，但是并不保证I/O操作成功

异步I/O（AIO）几个阶段

- (3)异步事件阻塞I/O
  - 用户进程可以对I/O事件进行阻塞，但是I/O操作并不阻塞
  - 通过select/poll/epoll等函数调用来达到此目的
- (4)异步事件非阻塞I/O
  - 用户程序不用等带I/O事件真正发生
  - I/O操作完成，内核会通过函数回调或者信号机制通知用户进程

设备操作—同步 or 异步

- 块设备、网络设备驱动的操作
  - 全是异步
- 字符型
  - 一般是同步的
  - 异步实现：在驱动中实现对应的异步函数
  - AIO结构：kiocb

DMA（Direct Memory Access）

- 直接内存存取
- 解决快速数据访问
- DMA控制器可以不需要处理器的干预，在设备和系统内存高速传输数据

DMA控制器组成

- 主存地址寄存器
- 数据数量寄存器
- DMA的控制/状态逻辑
- DMA请求触发器
- 数据缓冲寄存器
- 中断结构

DMA数据传输--三个阶段

- 传送前的预处理
  - 向DMA控制器发送设备识别信号，启动设备，测试设备运行状态，送入内存地址初值，传送数据个数、DMA的功能控制信号
- 数据传送
  - 在DMA卡控制下自动完成
- 传送结束处理

DMA数据传送触发

- 软件对数据的请求
- 硬件异步地将数据传给系统

DMA定义

- 不同的处理器对于DMA有不同的定义
- ARM平台定义

```
struct dma_struct {
    void                *addr;          //单个DMA地址
    unsigned long       count;          //单个DMA大小
    struct scatterlist buf;             //单个DMA
    int                 sgcount; //DMA SG的数量
    struct scatterlist *sg;         //分散搜集列表，解决多页的DMA传输问题
    unsigned int        active:1; //传输激活状态
    unsigned int        invalid:1;
    unsigned int        dma_mode;       //DMA模式
    int                 speed;          //DMA速度
    unsigned int        lock;           //设备已分配
    const char          *device_id;     //设备名称
    const struct dma_ops *d_ops;
};
```

DMA映射函数

- 内核提供
- 用来分配DMA缓冲区的，同时为这个缓冲区生成能被设备访问的地址的组合
- 分为连续映射（coherent dma mappings）和流式映射（steaming dma mappings）

THANKS!