《嵌入式系统》

6-1 嵌入式系统驱动程序设计基础

清華大學 Tsinghua University

提纲

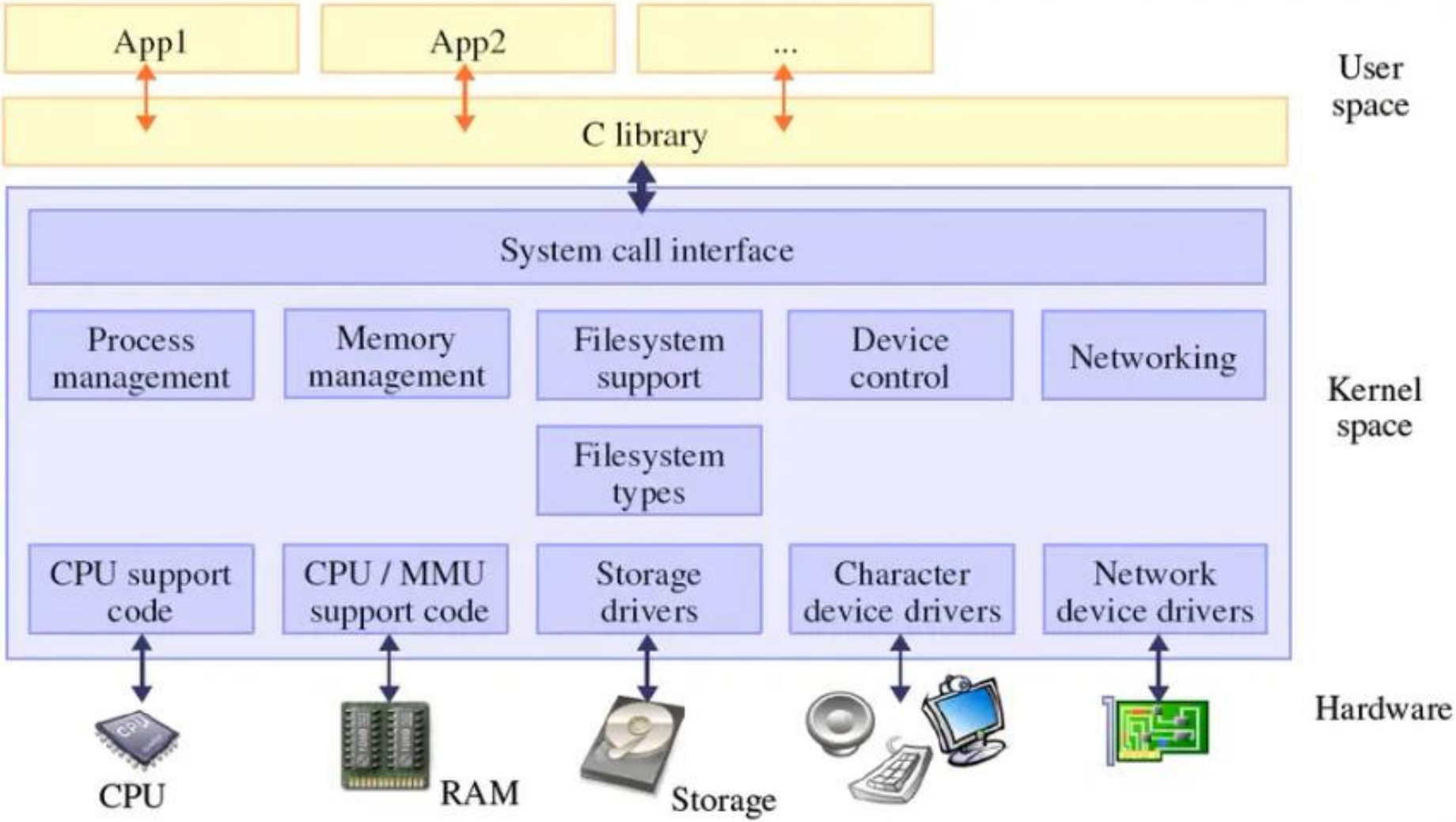
Linux设备驱动程序简介

- 设备驱动的功能
- 设备分类
- 设备文件和设备号
- 代码分布
- 驱动程序结构

Linux驱动相关内核机制

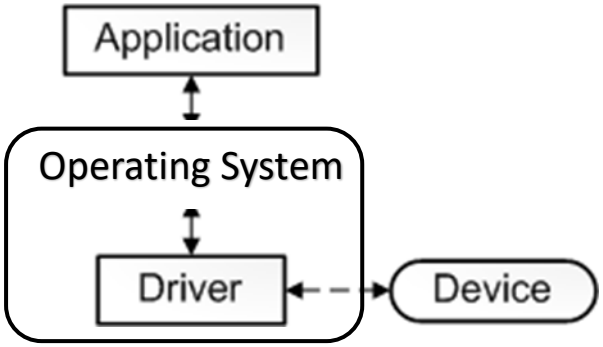
- 设备驱动模型
- 关键数据结构
- 同步机制
- 异步IO
- DMA

嵌入式系统中的驱动程序



驱动程序简介

- 一种描述性的定义
  - *设备驱动程序可以理解为操作系统的一部分，它的作用就是让操作系统能正确识别和控制设备。*



- 在Linux OS中，用户空间的应用开发者只需要通过C函数库来和内核空间打交道；而内核空间通过系统调用和VFS（virtual file system），来调用各类硬件设备的驱动。
- 硬件驱动通过操作对应硬件的寄存器来直接的控制硬件设备。

设备驱动的功能

- 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

设备驱动程序的基本特征

- **内核代码**：设备驱动程序是内核的一部分，如果设备驱动程序出错，则有可能导致系统崩溃。
- **内核接口**：设备驱动程序必须为内核或者其子系统提供一个标准接口。
- **内核机制与服务**：设备驱动可以使用标准的内核服务，如内存分配、中断和等待队列等。
- **动态装载、可配置**：作为Linux内核的模块，大多数设备驱动程序都可以在需要时动态地装载进内核，在不需要时从内核中卸载。内核编译的时候，可以选择把哪些驱动程序集成到内核里。

设备分类

- Linux支持三类硬件设备
  - **字符设备**：提供连续的数据流，应用程序可以顺序读取的设备，通常没有缓冲、不支持随机存取，支持按 字节/字 来读写数据；如鼠标、键盘、串口设备等。
  - **块设备**: 通过缓冲进行读写固定大小数据块(512/1024B)、支持随机（不需按顺序）访问的设备；如硬盘、软盘、CD-ROM驱动器、闪存等。
  - **网络设备**：通过BSD套接口(Berkeley Socket)访问

设备文件

- 设备文件（回顾 “嵌入式Linux文件系统”）
  - Linux中所有硬件设备都使用一个特殊的设备文件名（有时也称作“设备进入点”）来表示，如第一个IDE硬盘表示为/dev/hda，第一个打印机表示为 /dev/lp0
  - 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
- 驱动程序的主要任务之一，就是实现上述系统调用函数。

设备号

- 嵌入式Linux系统通过设备号区分不同设备。设备号分为主设备号和次设备号。内核通过主设备号将设备与相应的驱动程序对应起来。当一个驱动程序要控制若干个设备时，就要用次设备号来区分它们。
  - 主设备号，标识设备的种类、使之对应使用的驱动程序
  - 次设备号，标识使用同一设备驱动程序的不同硬件设备

创建设备文件

- 创建设备文件的命令(要求root权限)格式为：

  mknod　/dev/xxx type major minor

  其中：

  xxx为设备名；

  type为设备类型，若为字符设备，则为c，若为块设备，则为b；

  major和minor分别为主设备号、次设备号。

- 查看设备文件是否创建成功，命令的一般格式为：

  ls -l /dev |grep devicename

  //在dev目录中查找匹配“devicename”的设备文件

设备驱动代码的分布

- Linux源码大多数都是设备驱动！
- 所有驱动源码都放在Drivers目录下，分为几类:
  - block
  - char
  - cdrom：特殊接口（而非IDE或SCSI）的CDROM设备
  - pci
  - scsi
  - net
  - sound

清華大學 Tsinghua University

提纲

Linux设备驱动程序简介

- 设备驱动的功能
- 设备分类
- 设备文件和设备号
- 代码分布
- 驱动程序结构

Linux驱动相关内核机制

- 设备驱动模型
- 关键数据结构
- 同步机制
- 异步IO
- DMA

嵌入式设备驱动程序结构

- Linux的设备驱动程序与外界的接口分为三部分：
  - 驱动程序与操作系统内核的接口
    - 通过include/linux/fs.h中的file_operations数据结构来完成。
  - 驱动程序与系统引导的接口
    - 利用驱动程序对设备进行初始化。
  - 驱动程序与设备的接口
    - 描述驱动程序如何与设备进行交互，这部分的实现与具体设备密切相关。

嵌入式设备驱动程序结构

- 根据功能划分，Linux的设备驱动程序的代码结构可分为：
  - 驱动程序的注册与注销
  - 设备的打开与释放
  - 设备的读写操作
  - 设备的控制操作
  - 设备的轮询和中断处理

设备驱动程序的注册与注销

- 设备驱动注册是将驱动程序与其主设备号相关联的过程。
  - 字符设备的注册函数：fs/devices.c中的register_chrdev();
  - 块设备的注册函数：fs/block_dev.c中的register_blkdev();

- 将不需要的资源及时释放是一个良好的设计习惯。注销设备驱动只需要调用对应的注销函数：
  - unregister_chrdev ();
  - unregister_blkdev();

  占用的主设备号也会同时被释放。

设备的打开与关闭

- open( )函数

在设备驱动程序中，设备的打开操作由功能接口函数open( )完成。它主要提供初始化设备、识别次设备号等能力，为以后对设备进行I/O操作做准备。

- release( )函数

release( )函数是释放设备的接口。

设备的读写操作

- 在字符设备驱动程序中，由接口函数read( )和write( )完成字符设备的读写操作。
- 函数read( )和write( )的主要任务就是把内核空间的数据复制到用户空间，或者从用户空间把数据复制到内核空间。
- 块设备使用block_read()和block_write()，缓存不命中或要将数据写入设备时，才执行真正的与设备之间的数据传输

设备的控制操作

- 在设备驱动程序中，接口函数ioctl( )主要用于对设备进行读写之外的其他控制操作。函数ioctl( )的操作与设备密切相关。比如，串口的传输波特率、马达的转速等等，这些操作一般无法通过read( )和write( )操作来完成。
- 在用户空间ioctl函数的定义为：

```
int ioctl(int fd, int cmd, ...);
```

其中fd是设备文件名，cmd是用户程序对设备的控制命令，后面的省略号代表若干补充参数，一般最多一个，这个参数有无和cmd的意义相关。

设备的轮询和中断处理

- 主流方式——中断：在设备驱动程序的初始化模块中定义了设备中断。设备驱动程序通过调用request_irq( )函数来申请中断，并通过中断信息将中断号和中断服务联系起来。中断使用结束，可以通过调用free_irq( )函数来释放中断。

- 对不支持中断的设备，使用轮询：CPU按照一定周期查看设备状态，以决定是否做进一步处理。

- 思考：如何理解中断是比轮询效率更高的机制？

提纲

Linux设备驱动程序简介

- 设备驱动的功能
- 设备分类
- 设备文件和设备号
- 代码分布
- 驱动程序结构

Linux驱动相关内核机制

- 设备驱动模型
- 关键数据结构
- 同步机制
- 异步IO
- DMA

设备模型

- 设备模型提供独立的机制表示设备，并表示设备在系统中的拓扑结构
- 主要功能
  - 建立系统中**总线、驱动、设备**三者之间的联系桥梁
  - 向用户空间展示内核中各种设备的拓扑结构
- 上述信息是作为驱动程序开发者在开发过程中必须掌握的，设备模型将内核管理的上述信息展现给了用户空间，这也就是设备模型的意义。

sysfs

- sysfs是设备拓扑结构的文件系统表现,其主要的几个顶级目录：
  - 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：系统中的设备类型
  - 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
  - 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

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

上述4种基本结构描述了Linux系统设备模型中可以感受的对象，即总线、设备、设备驱动、设备类型，称为设备模型的基本元素。

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/*/

提纲

Linux设备驱动程序简介

- 设备驱动的功能
- 设备分类
- 设备文件和设备号
- 代码分布
- 驱动程序结构

Linux驱动相关内核机制

- 设备驱动模型
- 关键数据结构
- 同步机制
- 异步IO
- DMA

关键数据结构——kobject

- 设备模型的核心对象：kobject
  - 提供基本的对象管理
  - 每个在内核中注册的kobject对象都对应于sysfs文件系统中的一个目录，同时对象模型层次结构在内存中形成树状结构，导致最终sysfs的形成
  - 提供统一的引用计数系统

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

- kobject初始化其引用计数为1。如果引用计数不为0，则该对象会继续留在内存中，如果引用计数减少到0，对象将被摧毁。
- 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

- ktype描述一族具有共同特性的kobject

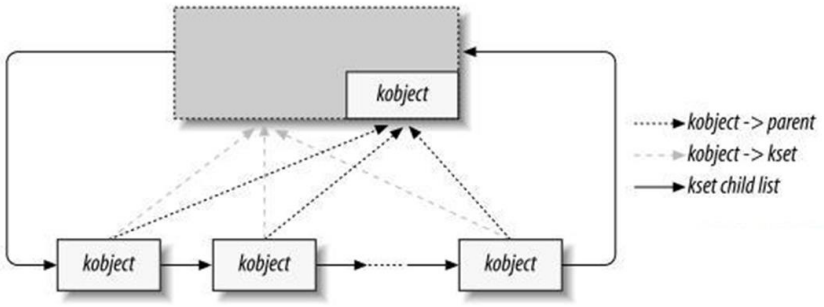
```
struct kobj_type{
        void (*release)(struct kobject *kobj);          /*析构函数*/
        struct sysfs_ops *sysfs_ops;/*包含对属性进行读和写操作的两个函数的结构体*/
        struct attribute **default_attrs;/*定义了kobject相关的默认属性*/
}
```

关键数据结构——kset

- kset是kobject对象的集合体
- 与ktype的区别：具有相同ktype的kobject可以被分到不同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*/}
```

包含在kset中的所有kobject组织成一个双向循环链表，list域是该链表的头。kset内嵌了一个kobject对象（kobj），所有属 于这个kset的kobject对象的parent域均指向这个kobj。此外，kset还依赖于kobj维护引用计数：kset的引用计数实际上就是内嵌的kobj的引用计数。



提纲

Linux设备驱动程序简介

- 设备驱动的功能
- 设备分类
- 设备文件和设备号
- 代码分布
- 驱动程序结构

Linux驱动相关内核机制

- 设备驱动模型
- 关键数据结构
- 同步机制
- 异步IO
- DMA

同步机制

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

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 — 读写锁（1）

- rwlock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。
  - 这可能导致写饥饿现象（**想想为什么？**）
- 读写锁类型是rwlock_t，在<linux/spinlock.h>

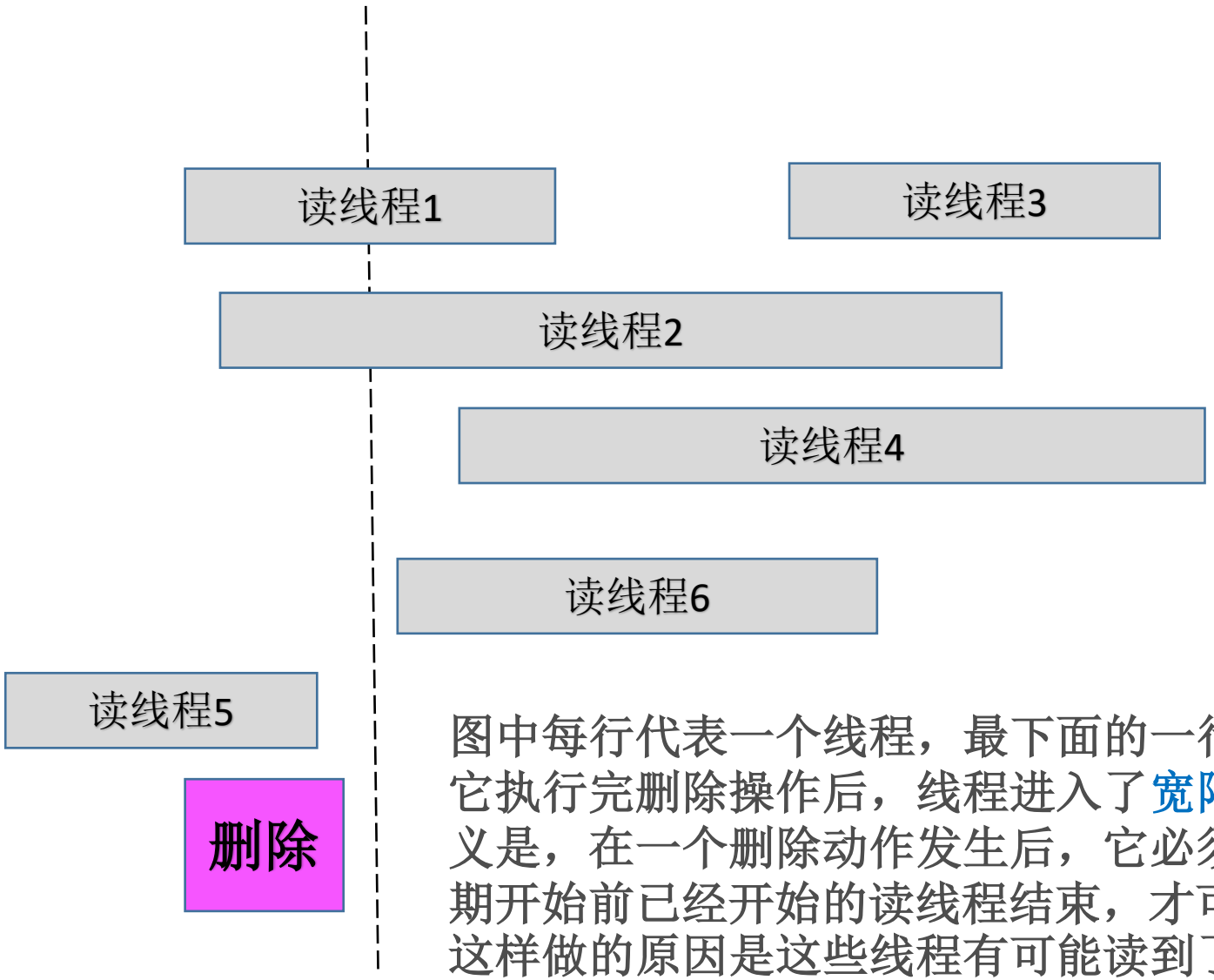
同步锁 — RCU锁（1）

- RCU（Read-Copy Update）锁机制是Linux2.6内核中新的锁机制
- 高性能的RCU锁机制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

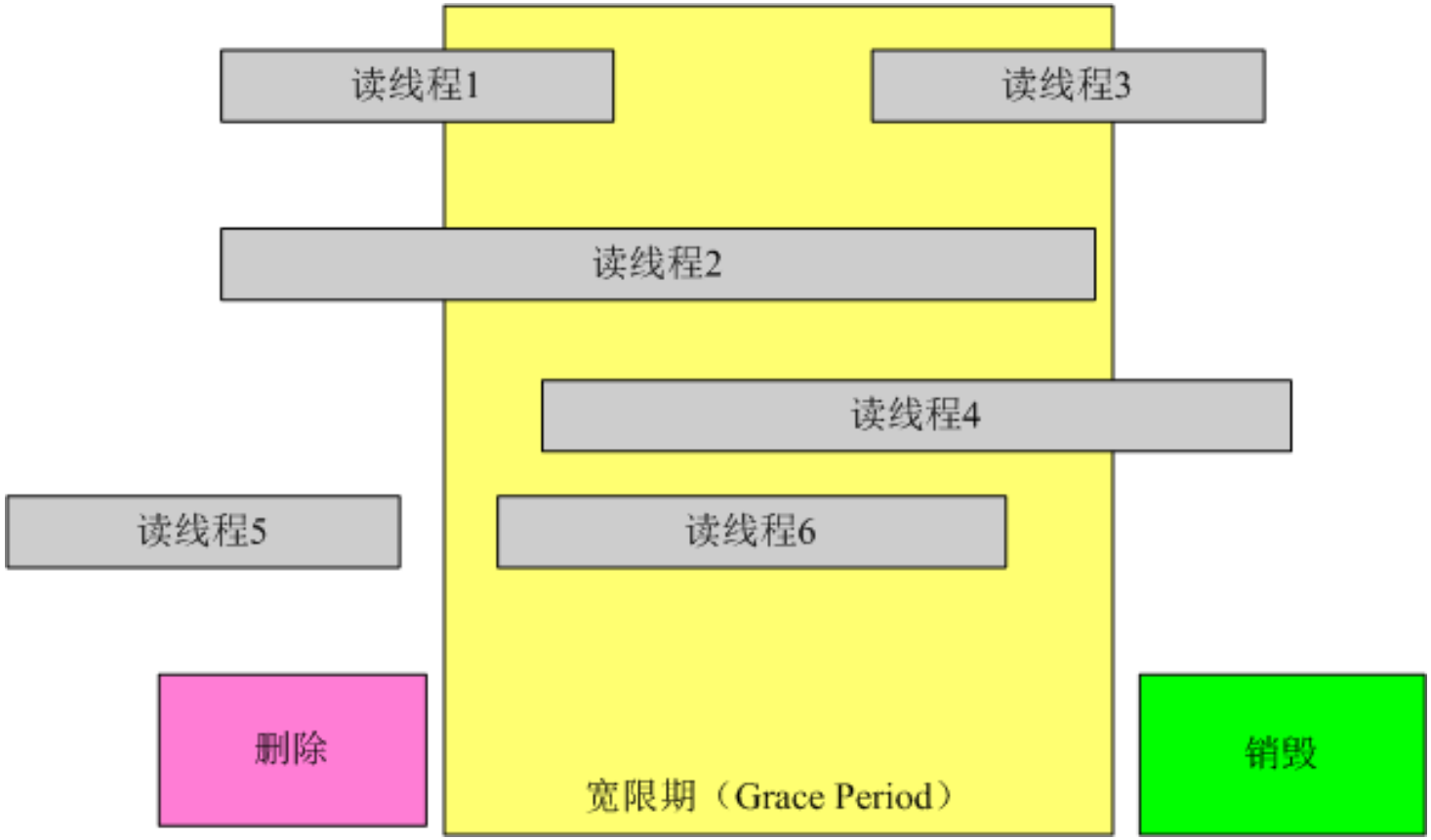
同步锁 — RCU锁（2）

- 读(Read)：读者不需要获得任何锁就可访问RCU保护的临界区；
- 拷贝(Copy)：写者在访问临界区时，写者“自己”将先拷贝一个临界区副本，然后对副本进行修改；
- 更新(Update)：RCU机制将在在适当时机使用一个回调函数(Callback)把指向原来临界区的指针重新指向新的被修改的临界区，锁机制中的垃圾收集器负责回调函数的调用。
  - 时机：所有引用该共享临界区的CPU都退出对临界区的操作。即没有CPU再去操作这段被RCU保护的临界区后，这段临界区即可回收了，此时回调函数即被调用。

宽限期的一个例子



图中每行代表一个线程，最下面的一行是删除线程，当它执行完删除操作后，线程进入了宽限期。宽限期的意义是，在一个删除动作发生后，它必须等待所有在宽限期开始前已经开始的读线程结束，才可以进行销毁操作。这样做的原因是这些线程有可能读到了要删除的元素。


读线程1
读线程3
读线程2
读线程4
读线程5
读线程6
删除
宽限期（Grace Period）
销毁

同步锁 — seqlock

- 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。
  - 适合如下场景：要保护的资源小、简单、常常被读存取、很少写存取、写存取动作完成很快

信号量

- 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，位于等待队列的第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

读写信号量

- 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。
- 写者具有排他性和独占性

原子操作

- 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
- 原子操作主要用在资源计数，很多应用计数（refcnt）就是通过原子操作实现的

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

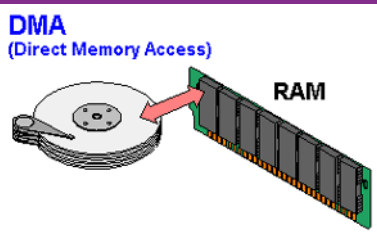
  void complete(struct completion *comp);

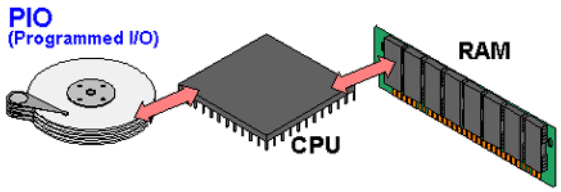
  void complete_all(struct completion *comp);

异步I/O

- 全称：异步事件非阻塞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。
- 块设备和网络设备驱动的操作全是异步的，但是对于字符型设备，需要在驱动程序中实现对应的异步函数，才能实现异步操作
- 要使用AIO功能，需要包含头文件aio.h

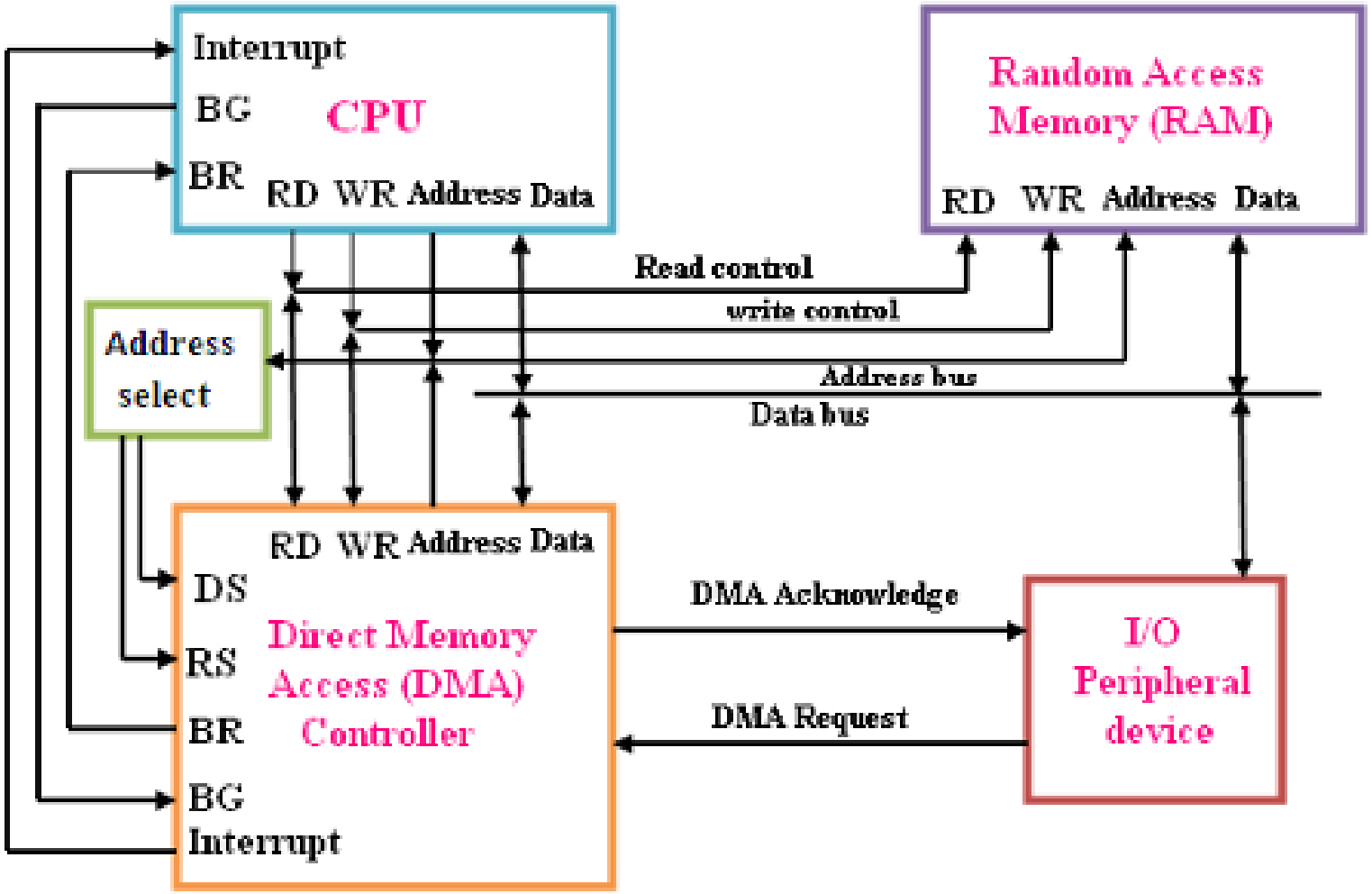
DMA





- CPU写入外部设备有两种不同的方式，一种是直接操作硬件设备寄存器，这一般是不需要中断的，CPU在写入一个寄存器的时候会处于阻塞状态，直到写入完成之前不能继续，因此通常只有处理非常少的数据的时候才会这样做。
- 另一种方式是使用DMA，DMA控制器是一个专用的外部设备，CPU将需要发送的数据提前在内存中准备好，然后设置DMA设备的寄存器，让DMA设备从内存的指定位置开始，将内存中数据依次写到对应地址的外部硬件寄存器里，这样在DMA写入的同时CPU就可以做其他工作。DMA写入完成后会产生一个中断通知CPU。

DMA


Interrupt
BG
BR
CPU
RD WR Address Data
Random Access Memory (RAM)
RD WR Address Data
Read control
write control
Address bus
Data bus
Address select
RD WR Address Data
DS
RS
BR
BG
Interrupt
Direct Memory Access (DMA) Controller
DMA Acknowledge
DMA Request
I/O Peripheral device

DMA

- DMA数据传送过程
  - 传送前预处理：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，传输数据个数，DMA功能控制信号。（由CPU完成）
  - 数据传送：在DMA控制器控制下自动完成
  - 传送结束处理
- 有两种情况会引发数据传输
  - 软件对数据的请求
  - 硬件异步地将数据传给系统

DMA

- 软件对数据的请求：以read函数为例
  - 在进程调用read时，驱动程序分配一个DMA缓冲区，并让硬件传输数据到缓冲区，此时进程处于睡眠状态。
  - 当硬件传输数据到缓冲区完毕时，产生一个中断
  - 中断处理程序获得输入的数据，应答中断，并唤醒中断，该进程可读取数据。

DMA

- 异步使用DMA：当一个数据区块，即使没有进程读取它的数据，也不断有数据写入。
  - 有新数据到来，硬件发生中断
  - 中断处理程序分配一个缓冲区
  - 外围设备将数据写入缓冲区，完成时产生另一个中断
  - 处理程序分发新数据，唤醒相关进程，最后执行清理工作


Thank you!