第12章 内核模块

实验目的

- 了解内核模块的概念和特点
- 学习如何编写一个内核模块
- 掌握内核模块的实现机制，学会模块的加载和卸载

主要内容

- 背景知识
  - 内核模块概述
  - 内核模块编程
  - 内核模块机制的实现
- 实验内容
  - 通过内核模块显示进程控制块信息

内核模块概述

❖内核模块

- 是在核心空间运行的一种目标文件，不能单独执行但其代码可在运行时链接到系统中作为内核的一部分执行或卸载。Linux内核模块是一种特有的机制，它由一组函数和数据结构组成，可作为独立程序来编译。
- 当模块被安装时，它被链接到内核中，可在系统启动时进行模块安装，称静态加载；也可在系统运行时进行模块安装，称动态加载。

内核模块概述

❖ 内核模块的主要作用

- 是动态地增加或减少内核功能，许多情况下用户需要增加内核态程序。
  - ✓ 例如，添加一个文件系统或设备驱动程序，而这类程序运行在内核态，工作在核心空间，由于它们种类繁多、体积庞大，要求内核全部包含进去是十分困难的，于是Linux提供了一种称为可动态加载和卸载的内核模块（Loadable Kernel Modules，LKM）机制，通过模块机制来实现系统运行时对内核功能的动态扩充，就能大大提高单内核操作系统的灵活性与可扩展性。

内核模块概述

❖Linux内核模块

- 是一个编译好的、具有特定格式的独立目标文件，用户可通过系统提供的一组与模块相关的命令将内核模块加载进内核，当内核模块被加载后，它有以下特点：
  - ✓1）与内核一起运行在相同的内核态和内核地址空间；
  - ✓2）运行时具有与内核同样的特权级；
  - ✓3）可方便地访问内核中的各种数据结构。

内核模块概述

❖Linux内核模块

- 被载入内核的内核模块代码与静态编译进内核的代码没有区别，内核模块与内核中的其他模块交互只需采用函数调用，此外，内核模块还可以很容易地被移出内核，当用户不再需要某功能模块时，可以自动地将它从内核卸载以节省系统主存开销，配置十分灵活。

内核模块概述

❖Linux内核模块

- Linux内核需要对载入的内核模块进行管理，管理内核模块主要有两项任务:
  - ✓一是内核符号表管理；
  - ✓二是维护内核模块的引用计数。
- 内核将资源登记在符号表中，当内核模块被加载后，模块可以通过符号表使用内核中的资源，
  - ✓新模块载入内核时，系统把新模块提供的符号加进符号表中，这样新载入模块就可访问已装载模块提供的资源；
  - ✓在卸载一个模块时，系统释放分配给该模块的所有系统资源，如内核主存区等，同时将该模块提供的符号从符号表中删除。

内核模块概述

❖Linux内核模块

- 由于所有内核模块在加载后都在同一地址空间中，内核模块之间可相互引用各自导出的符号，因此内核模块之间会产生依赖性:
  - ✓如果A模块需要用到B模块导出的符号，而B模块没有被载入内核的话，A模块的加载就会出错；
  - ✓同样，一个内核模块如果有其他内核模块引用它导出的符号，内核也不允许该内核模块被卸载，内核模块的引用计数器便用来管理内核模块之间的依赖性。
  - ✓如果一个内核模块被依赖，它的引用计数就会增加；当依赖减少时，相应的引用计数也会减少；一个内核模块只有在引用计数为0时候才允许被卸载。

主要内容

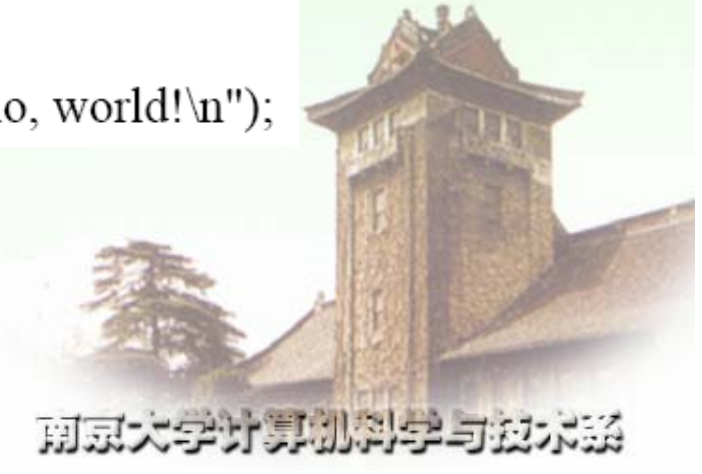
- 背景知识
  - 内核模块概述
  - 内核模块编程
  - 内核模块机制的实现
- 实验内容
  - 通过内核模块显示进程控制块信息

内核模块编程

❖ 内核模块的结构

- 编写 “Hello,world!”内核模块

```
1)  #include <linux/init.h>
2)  #include <linux/module.h>
3)  MODULE_LICENSE("GPL");
4)
5)  static int hello_init(void)
6)  {
7)          printk(KERN_ALERT "Hello, world!\n");
```



内核模块编程

❖内核模块的结构

- 编写“Hello,world!”内核模块

```
8)              return 0;
9)      }
10)
11)     static void hello_exit(void)
12)     {
13)             printk(KERN_ALERT "Goodbye, world!\n");
14)     }
15)
16)     module_init(hello_init);
17)     module_exit(hello_exit);
```

内核模块编程

❖编译与加载

- 在v2.6中，编译、链接后生成的内核模块后缀为.ko，编译过程中首先会到内核源代码目录下，读取顶层makefile文件，然后返回模块源代码所在的目录继续编译。
- 编译内核模块的makefile，只需要下面一行:
  - ✓ obj-m:=hello.o
- 生成的内核模块为hello.ko。
- 如果需要生成一个名为mymodule.ko的内核模块，并且该内核模块的源代码来源于modulesrc1.c和modulesrc2.c两个文件，makefile应该写成如下形式:
  - ✓ obj-m:=mymodule.o
  - ✓ module-objs:=modulesrc1.o modulesrc2.o

内核模块编程

❖编译与加载

- 如果用户采用这种makefile，在调用make命令时，需要将内核源代码所在目录作为一个参数传递给make命令。
- 例如，如果v2.6的内核源代码位于/usr/src/linux-2.6下，用户模块源代码所在目录应该使用的make命令为：
  - ✓make-C/usr/src/linux-2.6 M=`pwd`modules

内核模块编程

❖编译与加载

- makefile还提供另一种形式，用户可指定内核源代码所在的目录。
- 对于“hello world!”例子，在makefile中指定内核源代码的方式为:

```
/*如果定义了KERNELRELEASE宏，则可直接通过配置内核的内核源代码目录编译文件*/
ifneq($(KERNELRELEASE),)
obj-m:= hello.o
else
/*否则，需要从命令行获取配置内核的内核源代码目录信息，并编译文件*/
KERNELDIR ?= /lib/modules/$(shell uname -r)/build
PWD:= $(shell pwd)
default:
$(MAKE)-C$(KERNELDIR) M=$(PWD) modules
endif
```

内核模块编程

❖编译与加载

- 在这个makefile中，KERNELDIR指定内核的源代码目录，该目录通过当前运行内核使用的模块目录中的build符号链接指定。
- 当编译好内核模块后，用户以超级用户身份就可将内核模块加载到内核中。
- 内核提供modutils软件包供用户对内核模块进行管理，该软件包安装后会在/sbin目录下安装insmod、rmmod、ksyms、lsmod、modprobe等实用程序

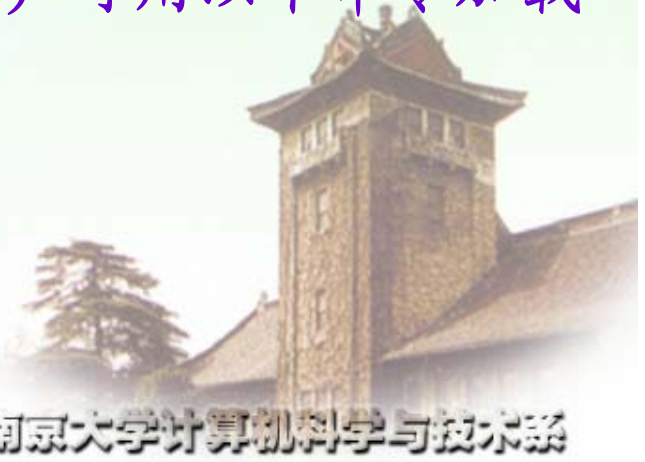
内核模块编程

- 编译与加载
  - insmod命令
    - 把需要载入的模块以目标代码形式加载进内核中，insmod自动调用modules_init( )函数中定义的过程运行，超级用户使用这个命令，其格式为：
      - # insmod [path]modulename
  - rmmod命令
    - 将已经载入内核的模块从内核中卸载，rmmod自动调用modules_exit( )函数中定义的过程运行，命令格式为
      - # rmmod [path]modulename

内核模块编程

❖编译与加载

- ksyms命令
  - ✓用来显示内核符号和模块符号信息。
- lsmod命令
  - ✓显示已经载入内核的所有模块信息，包括被载入模块的模块名、大小和引用计数等，命令格式为：
    - # lsmod
  - ✓对于本节中的hello模块，超级用户可用以下命令加载模块：
    - #insmod hello.ko



内核模块编程

❖内核符号表

- 是一个用来存放所有模块可以访问的符号，以及对应地址的特殊数据结构，模块的链接是将模块插入到内核的过程，模块所导出的符号都将成为内核符号表的一部分。模块根据符号表从核心空间获取主存地址，从而确保在核心空间中正确地运行，对于从模块中导出的符号，在符号表中会包含第3列“所属模块”，在v2.6内核中，用户可从/proc/kallsyms中以文本方式读取内核符号表。

内核模块编程

❖内核符号表

- 在v2.4内核中，缺省情况下，模块中的非静态全局变量及函数在模块加载后会输出到内核符号表，而在v2.6内核中，缺省情况下，这些符号不会被输出到内核符号表中。如果模块需要导出符号供其他模块使用，应该使用下面定义的两个宏：
  - ✓EXPORT_SYMBOL(name)
  - ✓EXPORT_SYMBOL_GPL(name)

内核模块编程

❖初始化与清理函数

- 内核模块必须调用宏module_init与module_exit去注册初始化与清理函数。初始化函数通常定义为:

```
static int __init init_func (void)
{
/* 初始化代码*/
}
module_init(init_func);
```

内核模块编程

❖初始化与清理函数

- 与初始化函数相对应的是清理函数，大部分模块都需要设置清理函数，该函数在模块卸载时被调用；如果一个模块没有定义清理函数，内核将不会允许它被卸载，模块在清理函数中需要将已申请的资源归还给系统。清理函数的定义为：

```
static void__exit exit_func(void)
{
/* 清理代码 */
}
module_exit(exit_func);
```

内核模块编程

❖模块参数

- 用户在加载内核模块之前，可能需要传递参数给内核模块，与内核模块进行简单的交互，用户和内核模块能有多种交互方式，例如通过/proc文件系统或内核模块参数。如果用户只需要在模块加载时与模块发生交互，则采用内核模块参数方法较好。
- 用户在insmod、modprobe命令中直接制定参数，命令形式为:
  - ✓#modprobe modname var=value

  modname是要加载的模块名，var是要传递的变量名，value是传递的参数值。

内核模块编程

❖模块参数

- 如果用户每次加载模块时传递的参数都相同，每次在命令行中输入参数比较繁琐，可以在/etc/modprobe.conf配置文件中预先写入参数，每次当模块被加载时，该配置文件中的参数就会被自动传递给模块。
- 模块要使用用户传递的参数，应该采用以下定义的宏：
  - ✓module_param(name, type, perm)
  - ✓module_param_array(name, type, nump, perm)

内核模块编程

❖模块参数

- type指明参数类型，模块直接支持的参数类型有:
  - ✓bool、invbool。bool为布尔型，invbool类型是颠倒的布尔类型，真值为false，假值为true。
  - ✓charp。字符串指针，指针指向用户传入的字符串。
  - ✓int、long、short、uint、ulong、ushort。基本变长整型值，以u开头的是无符号值。

内核模块编程

❖模块参数

- perm为参数在sysfs文件系统中所对应的文件节点的属性。
  - ✓在v2.6内核中，系统使用sysfs文件系统，该文件系统可以动态、实时，有组织有层次地反映当前系统中的硬件、驱动程序等情况。
  - ✓模块被加载后，在/sys/module/目录下将出现以此模块名命名的目录，如果此模块存在perm不为0的命令行参数，则在此模块的目录下将出现parameters目录，包含一系列以参数名命名的文件节点，这些文件的权限值等于perm，文件的内容为参数的值。

内核模块编程

❖模块参数

- nump为一个指针。
  - ✓该指针所指的变量保存输入的数组元素个数，当不需保存实际输入的数组元素个数时，该指针可设置为NULL。用户传递数组参数给模块时，使用逗号分隔输入的数组元素。设定好nump后，如果用户传递的数组大小大于指定的nump，模块加载者会拒绝加载更多的数组元素。
  - ✓需要注意，所有模块参数应当给定一个缺省值，如果用户没有传递参数给模块，参数会使用预先设定好的缺省值。

内核模块编程

❖模块参数

- 以本章的“hello world!”模块（称为hellop）为例，说明模块参数的使用方法，模块使用2个参数：一个是名为howmany的整型值；一个称为whom的字符串。当模块加载时，将对whom说hello不止一次，而是howmany次。用户可采用以下命令行加载：
  - ✓#insmod hellop howmany=10 whom="Mom"

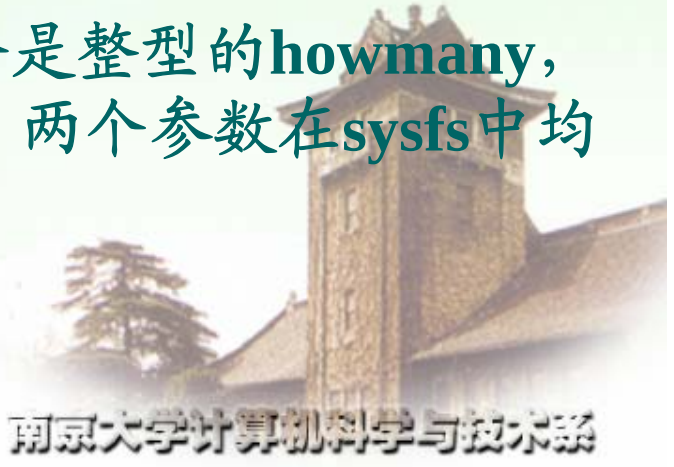
内核模块编程

❖模块参数

- 当模块以这样的方式加载后，模块会显示“hello,Mom”10次。为了使用howmany和whom两个参数，在模块源代码中，应该插入如下代码:

```
static char *whom="world";
static int howmany=1;
module_param(howmany, int,S_IRUGO);
module_param(whom, charp,S_IRUGO);
```

- module_param定义两个参数，一个是整型的howmany，另一个参数是字符串类型的whom，两个参数在sysfs中均是只读的。



主要内容

- 背景知识
  - 内核模块概述
  - 内核模块编程
  - 内核模块机制的实现
- 实验内容
  - 通过内核模块显示进程控制块信息

内核模块机制的实现

❖模块在内核中的表示

- 内核在管理模块时使用的管理数据结构为struct module，每一个内核模块被载入时，都要为其分配一个module对象，用一个双向链表把所有module对象组织起来，该链表的第1个元素为modules，开发者能够通过该元素依次访问内核中所有的module对象。
- 内核通过module对象主要是为了记录模块的依赖，并进行模块导出符号的管理。

内核模块机制的实现

❖模块在内核中的表示

表 12-1module 结构重要成员

类型	成员名	作用
enum module_stat	stat	模块当前状态
char [60]	nam	模块名
const struct kernel_symbol *	syms	导出符号数组
unsigned int	num_syms	导出符号数组大小
const struct kernel_symbol *	gpl_syms	GPL 导出符号数组
unsigned int	num_gpl_syms	GPL 导出符号数组大小
struct module_ref	ref[NR_CPUS]	每个 CPU 上对该模块的引用计数

- **stat成员表明当前模块的状态：MODULE_STATE_LIVE（处于激活状态）, MODULE_STATE_COMING（正在被初始化状态）,MODULE_STATE_GOING（正在被卸载状态）。**

内核模块机制的实现

- 模块在内核中的表示
  - 每个module对象都包含有多个引用计数，每个CPU都有一个引用计数。
    - ✓ 每次当模块被使用时，模块的引用计数都会加1；
    - ✓ 相反，当模块不被使用时，模块的引用计数就会相应减少，仅当引用计数为0时，该模块才能被内核卸载。
    - ✓ 例如，假设一个MS-DOS的文件系统被编译成为模块，当模块被加载后，模块的引用计数为0；当用户用mount命令挂上一个MS-DOS的分区后，模块引用计数就变成1；只有在用户umount后，引用计数变成0，该模块才能被卸载。

内核模块机制的实现

- 模块在内核中的表示
  - 在内核代码段中，有3个段保存导出符号的相关信息:
    - __kstrtab保存导出符号的名字；
    - __ksymtab保存供所有模块使用的符号的地址；__ksymtab_gpl保存仅供GPL协议模块使用的符号的地址。
    - 只有被EXPORT_SYMBOL和EXPORT_SYMBOL_GPL宏导出的符号才会被C编译器写入内核代码的相应段中，在加载内核时，会根据代码段中的符号信息创建符号表。当前内核符号表的内容可通过/proc/kallsyms查看:

内核模块机制的实现

- 模块在内核中的表示
  - 在内核代码段中，有3个段保存导出符号的相关信息:

```
c0400294 T _stext
c0400294 t run_init_process
c0400294 T stext
c04002d0 t init
c04005c7 t rest_init
c04005e8 t try_name
c0400765 T name_to_dev_t
c04009ac T calibrate_delay
c0400c40 T hard_smp_processor_id
c0400c50 t target_cpus
```

内核模块机制的实现

- 模块的加载与卸载
  - 模块的加载
    - 用户通过insmod命令将模块载入内核，该命令的主要操作如下：
      - 从命令行读入要被载入的模块名。
      - 获得模块代码，它通常放在/lib/modules目录下。
      - 调用init_module( )函数，将包含模块代码缓存的指针、模块代码长度和用户参数传递给函数，该函数将完成模块的加载工作。

内核模块机制的实现

- 模块的加载与卸载
  - 模块的加载
    - ✓ init_module( )函数的主要工作有:
      1. 检查用户是否有权限加载模块（具有CAP_SYS_MODULE权限）。
      2. 在核心空间中为模块申请主存，并将模块目标代码拷贝进入核心空间
      3. 检查模块代码是否是有效的ELF(Executable and Linking Format，可执行连接格式)，如果不是，报错。
      4. 在核心空间为用户传递的模块参数申请主存，并将参数拷贝到核心空间。
      5. 内核通过模块名检查模块是否已经被载入内核。

内核模块机制的实现

- 模块的加载与卸载
  - 模块的加载
    - ✓ init_module( )函数的主要工作有:
      6. 为模块的可执行代码分配空间，并将模块目标代码中的相应段拷贝到该空间。
      7. 为模块的初始化代码分配空间，并将模块目标代码中的相应段拷贝到该空间。
      8. 获得模块的module对象的位置，在模块目标代码的正文段中存储着该对象。
      9. 根据6）、7）行，初始化module对象中的module_code和module_init成员。
      10. 初始化modules_which_use_me，并把模块的引用计数置成0。
      11. 根据模块的授权协议，设置license_gplok，如果该模块不符合GPL协议，该标志置为0。

内核模块机制的实现

- 模块的加载与卸载
  - 模块的加载
    - ✓ init_module( )函数的主要工作有:
      12. 根据模块和内核符号表，对模块代码进行重定位，将代码中的符号解析成主存地址。
      13. 设置module对象中的syms和gpl_syms成员，这两个值被设置成模块导出符号表的地址。
      14. 解析用户传递过来的参数，并将值赋给模块中相应的符号。
      15. 注册module对象中的mkobj成员。注册后，在sysfs文件系统的module目录中会增加一个该模块的目录，该目录下包含有该模块的信息。

内核模块机制的实现

- 模块的加载与卸载
  - 模块的加载
    - ✓ init_module( )函数的主要工作有:
      16. 将第2）步中申请的主存释放。
      17. 将module对象加入全局的模块对象双向链表。
      18. 将模块状态设置为MODULE_STATE_COMING。
      19. 如果模块自定义初始化函数，调用模块的初始化函数。
      20. 设置模块状态为MODULE_STATE_LIVE。
      21. 结束

内核模块机制的实现

- 模块的加载与卸载
  - 模块的卸载
    - ✓ 用户可以通过rmmod命令将内核模块卸载，该命令所做的操作是:
      - 1）读取要被卸载的模块名。
      - 2）打开/proc/modules文件，查看该模块是否已经被卸载。
      - 3）调用delete_module( )，把模块名传递给该函数，该函数将完成模块的卸载工作。

内核模块机制的实现

- 模块的加载与卸载
  - 模块的卸载
    - ✓ **sys_delete_module( )函数的主要工作是：**
      1. 检查用户权限，只有具有CAP_SYS_MODULE权限的用户才可以卸载模块。
      2. 将模块名拷贝进核心空间。
      3. 在全局模块对象的双向链表中查找到该模块的module对象。
      4. 通过module对象的modules_which_use_me成员检查是否有其他模块依赖该模块。只有在没有依赖的情况下，函数才能继续完成卸载。
      5. 检查模块状态，只有处于MODULE_STATE_LIVE状态的模块才能被卸载。

内核模块机制的实现

- 模块的加载与卸载
  - 模块的卸载
    - ✓ **sys_delete_module( )函数的主要工作是：**

6. 如果该模块有自定义初始化函数，该模块只有在也定义了清理函数的情况下才允许被卸载。
7. 为了防止竞争，将系统中的其他CPU停止。
8. 将模块的状态设置成MODULE_STATE_GOING。
9. 如果模块的引用计数大于0，模块不能被卸载。
10. 如果模块定义了清理函数，调用清理函数。

内核模块机制的实现

- 模块的加载与卸载
  - 模块的卸载
    - ✓ **sys_delete_module( )函数的主要工作是：**
      11. 将模块加载时注册的mkobj，取消掉。
      12. 如果有其他模块被该模块使用，修改其他模块的引用关系。
      13. 释放该模块的module对象
      14. 释放模块占用的主存（代码、符号表、异常表）
      15. 结束

主要内容

- 背景知识
  - 内核模块概述
  - 内核模块编程
  - 内核模块机制的实现
- 实验内容
  - 通过内核模块显示进程控制块信息

通过内核模块显示进程控制块信息

❖实验说明

- 在内核中，所有进程控制块都被一个双向链表连接起来，该链表中的第1个进程控制块为init_task。
- 编写一个内核模块，模块接收用户传递的一个参数num，num指定要打印的进程控制块的数量；若用户不指定num或者num<0，模块则打印所有进程控制块的信息。需要打印的进程控制块信息有：进程PID和进程的可执行文件名。

通过内核模块显示进程控制块信息

- 解决方案
  - 定义模块参数
    - ✓ 该模块需要接受用户传递的参数，在使用该参数之前，需要在代码中预先定义好该参数，回顾前节内容可知，需要将该参数的类型设置为整型，并且在sysfs中的权限是只读的。定义的方法为：
      - static int num=-1;
      - module_param(num, int, S_IRUGO);
    - ✓ 该参数的初始值被设置为-1。-1将作为打印所有进程控制块的标记，默认值为-1，意味着当用户不传入任何参数时，模块将打印所有的进程的信息。

通过内核模块显示进程控制块信息

- 解决方案
  - 访问进程控制块链表
    - ✓ 在内核中，进程控制块被组织成多个双向链表，其中有一个双向链表包含所有的进程控制块，只需要访问该双向链表，就可以访问到所有进程控制块。Linux内核中几乎所有双向链表都采用相同的数据结构来实现，内核中定义list_head通用数据结构，其定义如下：

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

通过内核模块显示进程控制块信息

- 解决方案
  - 访问进程控制块链表
    - ✓ list_head中，开发者可以通过内核提供的一组宏创建并操作一个双向链表，而该链表中元素的类型为该数据结构，如图 12-1所示。

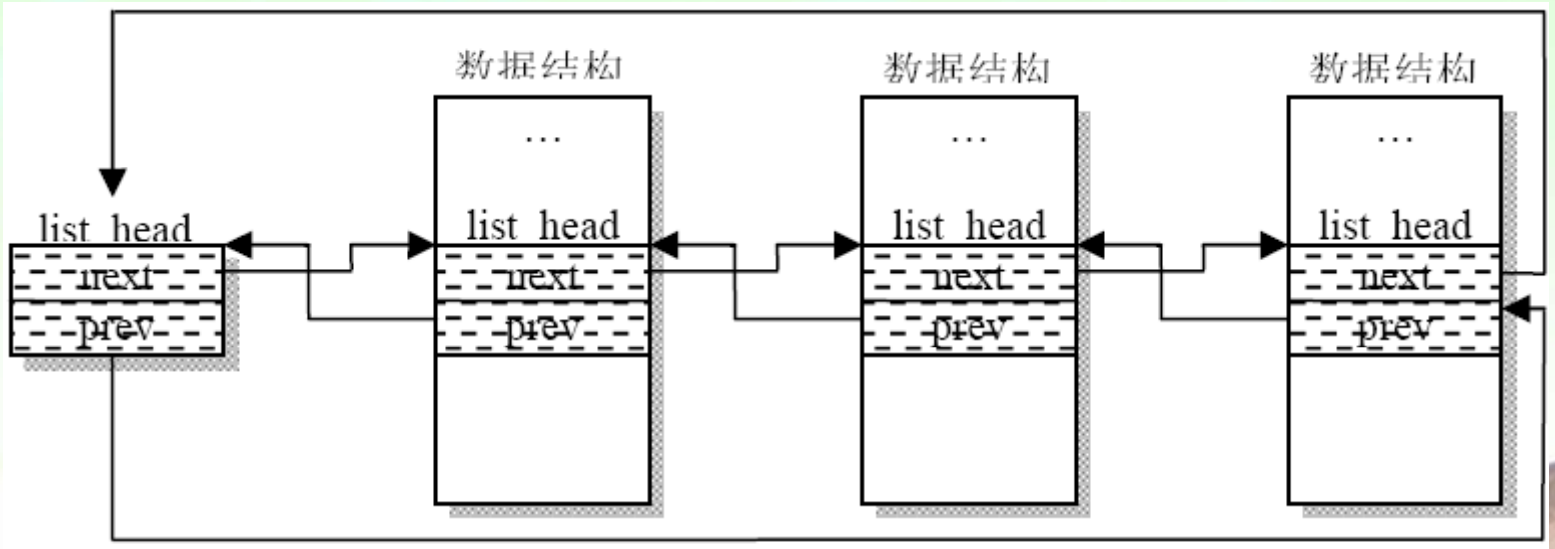


图 12-1 Linux 通用双向链表

通过内核模块显示进程控制块信息

- 解决方案
  - 访问进程控制块链表
    - ✓ list_head中，开发者可以通过内核提供的一组宏创建并操作一个双向链表，而该链表中元素的类型为该数据结构，如图 12-1所示。

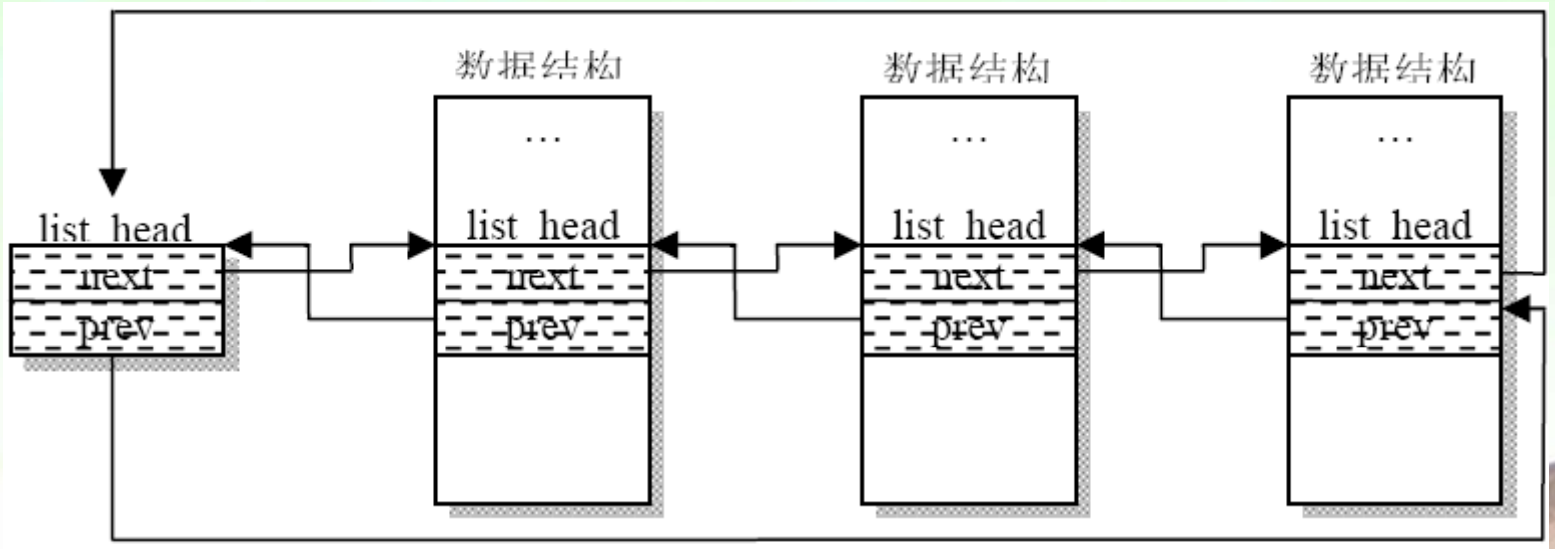


图 12-1 Linux 通用双向链表

通过内核模块显示进程控制块信息

- 解决方案
  - 访问进程控制块链表
    - ✓ 在进程控制块task_struct中，包含一个名为tasks的成员，该成员的类型为list_head，这意味着进程控制块能够通过该成员将进程控制块串成一个双向链表。Linux内核通过该成员将所有的进程都放入同一个双向链表，因为list_head结构中的next和prev指针并不是指向包含list_head的数据结构，而是指向另一个list_head数据结构。为了访问包含list_head的数据结构，内核提供一个宏：
      - list_entry(ptr,type,member);

通过内核模块显示进程控制块信息

- 解决方案
  - 访问进程控制块链表
    - ✓ 在该宏中，ptr是一个指向list_head的指针，type是包含list_head的数据结构类型，而member是list_head在该数据结构中的成员名。例如，若一个进程控制块中的tasks的地址为p，为了访问该进程控制块，可以采用：
      - list_entry(p,struct task_struct,tasks);
    - ✓ 该宏便会返回该进程控制块的地址。

通过内核模块显示进程控制块信息

- 解决方案
  - 访问进程控制块链表
    - ✓ 知道如何使用双向链表后，就可以方便地访问内核中所有的进程控制块，因为它通过tasks成员串成一个双向链表，如果得到一个进程控制块的地址p，开发者可以通过：
      - list_entry(p->tasks.next, struct task_struct, tasks);
    - ✓ 访问该双向链表中的下一个进程控制块。在该双向链表中，第1个进程控制块为init_task，如果开发者发现下一个进程控制块为init_task时，说明已经完整地遍历过所有进程控制块。

通过内核模块显示进程控制块信息

- 解决方案
  - 访问进程控制块链表
    - ✓ 内核定义宏for_each_process用于遍历所有的进程控制块，开发者通过该宏就能将所有的进程控制块访问一遍，该宏展开的形式为：
      - for (p = &init_task ; (p = list_entry((p)->tasks.next, struct task_struct, tasks) != &init_task ;)

通过内核模块显示进程控制块信息

- 解决方案
  - 输出进程控制块信息
    - ✓ 进程控制块中包含进程大部分信息，根据实验要求，模块需要打印进程的pid和可执行文件名，在进程控制块的数据结构中，成员pid为进程的PID，而成员comm包含进程的可执行文件名。在内核中，模块可以通过printk( )内核函数将这些信息打印到系统日志中。

第12章 内核模块