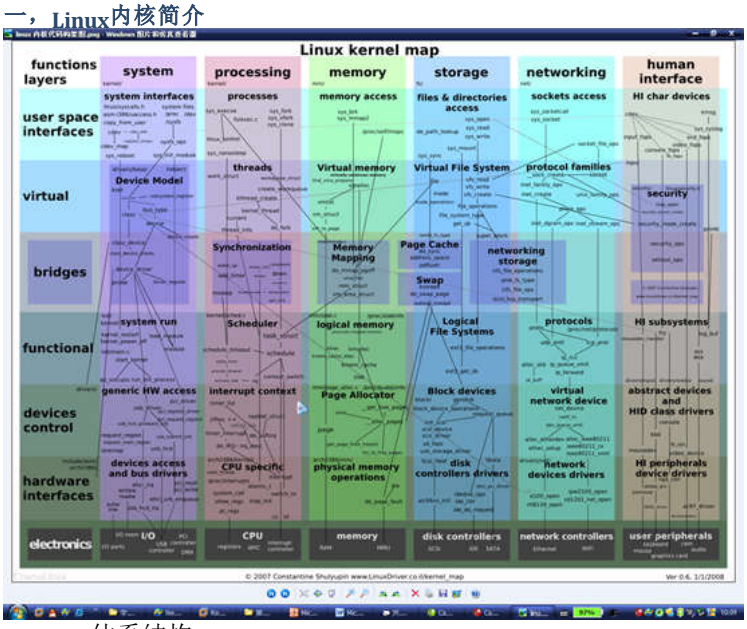
Linux内核

笔记本： Linux
创建时间： 2015/12/15 21:36
标签： Linux

Linux内核

2015年12月15日
21:36

一，Linux内核简介



1，linux体系结构

Linux由**用户空间和内核空间**组成。X86的CPU，用户代码运行在Ring3，内核代码运行在Ring0。通过**系统调用**和**硬件中断**可以在两种方式下转换。

Linux内核由以下部分组成：系统调用接口(SCI)，进程管理(PM),内存管理(MM)，虚拟文件系统(VFS)，体系结构(ARCH)，设备驱动(DD)，网络协议栈。

2,linux内核源代码

源代码采用**树形结构**组织，把功能相关的文件放在了同一目录下，可读性强。

- **Arch目录**：内核支持的cpu体系相关的代码，每个子目录对应一种cpu，包括系统引导，内存管理，系统调用等。
- **Block目录**：部分块设备驱动程序
- **Crypto目录**：加密、压缩、CRC检验算法
- **Documentation目录**：内核的相关文档
- **Drivers目录**：设备驱动程序
- **Fs目录**：存放各种文件系统的实现代码
- **Include目录**：内核需要的头文件，与平台无关的放在linux子目录下。
- **Lib目录**：库文件代码
- **Mm目录**：实现内存管理中与CPU体系结构无关的部分。
- **Net目录**：网络协议的实现代码。
- **Samples目录**：内核编程的范例
- **Scripts目录**：配置内核的脚本
- **Security目录**：SELinux的模块。

- **Sound目录**：音频设备驱动
- **Usr目录**：cpio命令的实现
- **Virt目录**：内核虚拟机

二，linux内核配置与编译

Linux具有可定制的优点，具体步骤为

1，清除临时文件、中间文件和配置文件：执行位置是在内核源码的顶层位置。

- **Make clean**：删除大部分文件，配置文件不删除
- **Make mrproper**：删除所有文件，包括配置文件
- **Make distclean**：相当于mrproper，同时 删除编辑器留下的backup和patch文件。

2，确定目标系统的软硬件配置情况。

3，配置内核，以下中的选择一个命令

- **Make config**：基于文本模式的交互式配置
- **Make menuconfig**：基于文本模式的菜单型配置
- **Make oldconfig**：使用已有的配置文件，但会询问是否新增配置选项。
- **Make xconfig**：图形化的配置。

可以参考已经存在的config配置文件，再在make menuconfig命令中增减功能。例如现有系统的config在/boot/下，使用时要拷贝到内核目录 重命名为**.config**。

4,编译内核

- **Make zImage**和**make bzImage**：在x86平台上，zImage只能用于小于512k的内核。加上**V=1**选项可以显示编译的详细信息。
- 编译好的内核位于**arch/<cpu>/boot/**目录下。

5，编译内核模块：**make modules**

6，安装内核模块：**make modules_install**。

7，制作init ramdisk：**mkinitrd initrd-$version $version**：initrd-$version为内核名，自己写的，$version是编译的内核版本，不能乱写。$version可以查询/lib/modules下的目录得到。

8，内核安装（x86为例）

- **Cp arch/X86/boot/bzImage /boot/vmlinux-$version**
- **Cp $initrd-$version /boot/**
- **修改/etc/grub.conf或者/etc/lilo.conf**

三，内核模块开发

内核文件本身不包括某组件，需要的时候动态的添加到正在运行的内核中，即内核模块。

1，程序结构

- 模块加载函数（必需）：**module_init**(函数名）
- 模块卸载函数（必需）：**module_exit**(函数名）

内核模块的编译是通过makefile实现的，不是使用gcc命令。在内核模块的源文件里没有main函数，而且标准输出是printk。

2，用于单内核模块编译的Makefile的基本格式:

```
Ifneq ($(KERNELRELEASE),)
Obj-m := hello.o需要变，内核模块的名称
```

```
Else
KDIR :=/lib/modules/2.6.18.53.e15/build需要变
All:
   Make -C $(KDIR) M=$(PWD) modules
Clean:
  Rm -f *.ko *.o *.mod.o *.mod.c *.symvers
endif
```

3，用于多内核模块编译的Makefile的基本格式:

```
Ifneq ($(KERNELRELEASE),)
Obj-m := hello.o需要变，内核模块的名称
Hello-objs :=main.o add.o 这里的hello要与模块名相同，多文件都加在这里
Else
KDIR :=/lib/modules/2.6.18.53.e15/build需要变
All:
   Make -C $(KDIR) M=$(PWD) modules
Clean:
  Rm -f *.ko *.o *.mod.o *.mod.c *.symvers
endif
```

4，模块的安装和卸载

模块编译生成之后，需要的就是安装到内核和从内核卸载：

- 安装模块：**insmod 模块名.ko**；**modprobe**也可以加载模块，但它会首先查看依赖性，先将有依赖的模块加载到内核。
- 卸载模块：**rmmod 模块名**
- 查看当前加载的模块：**lsmod**

5，模块可选信息

- **MODULE_LICENSE("GPL")**：模块的许可证，没有这样的说明加载模块时内核会抱怨，常见许可证有：GPL、GPLv2、GPL and additional rights、Dual BSD/GPL、Dual MPL/GPL、Proprietary。
- **MODULE_AUTHOR("")**：作者申明
- **MODULE_DESRIPTION("")**：模块描述
- **MODULE_VERSION("")**：模块的版本
- **MODULE_ALIAS("")**：模块的别名
- **Module_param(name,type,perm)**：定义模块参数，perm为权限,常见值**S_IRUGO**(读权限)、**S_IWUSR**(root可以修改参数)，type常见值为**bool**、**int**、**charp**，可以在加载模块时指定参数。加载模块时：**insmod 模块名.ko 参数名=值**来传递参数。

6，内核符号导出

一个模块里的变量和函数要被其他模块使用，首先要加载该模块(有依赖关系)，除此之外还要将变量名和函数名作为符号导出:

EXPORT_SYMBOL(符号名)；**EXPORT_SYMBOL_GPL(符号名)**.后者只能用于包含GPL许可的模块

7，常见问题

- 版本不匹配：内核模块版本是由KDIR指定的。解决办法：找到与内核版本相同的系统重新编译模块，即重新指定KDIR。

2.4与2.6内核模块对比：2.4模块是.o的，2.6的模块是先编译出.o再编译出.ko模块。

Printk打印可以附加不同优先级,级别不同打印会变化，但是都会在**/var/log/messages**中保存打印信息。

- KERN_EMERG：<0>，用于紧急消息，崩溃前的消息
- KERN_ALERT：<1>，需要立刻行动的消息
- KERN_CRIT：<2>，严重情况
- KERN_ERR：<3>，错误情况
- KERN_WARNING：<4>，有问题警告，默认级别
- KERN_NOTICE：<5>，正常情况，但仍然需要注意
- KERN_INFO：<6>，信息型消息
- KERN_DEBUG：<7>，用作调试消息

控制台优先级配置:**/proc/sys/kernel/printk**中四个数字，依次表示console_loglevel、default_message_loglevel、minimum_console_level、default_console_loglevel。先级比控制台优先级高时，打印信息可以在控制台输出。

四，uboot移植

1，**bootloader**：OS运行前初始化硬件设备，调用操作系统。Cpu上电以后会从一个固定的地址执行，而bootloader在起始地址处，这样就可以先执行bootloader。Bootloader依赖于cpu的结构和具体地嵌入式板及设备的配置。

Bootloader分为启动模式和下载模式：启动模式是正常的工作模式；下载模式通过串口等从主机下载文件，控制启动历程。

2，交叉工具链：

解压交叉工具链压缩包--->修改/etc/profile文件--->使修改生效source /etc/profile。常用工具：

- Arm-linux-gcc：交叉编译器，同gcc使用一样
- Arm-linux-objdump：反汇编工具，从可执行程序得到汇编代码。使用方法：**Arm-linux-objdump -D -S 可执行程序**。
- Arm-linux-readelf：elf文件查看工具。**-a 可执行程序**选项为查看所有信息；**-d**选项查看使用到的动态链接库。

3，uboot：是bootloader的一种，支持多种嵌入式cpu。Uboot重要的目录：

- **Board**：每一个开发板对应一个目录
- **Common**：实现boot支持的命令代码
- **Cpu**：与特定处理器加工相关的代码
- **Disk**：对磁盘的支持
- **Doc**：uboot文档目录
- **Drivers**：uboot支持的设备驱动程序
- **Fs**：文件系统的支持
- Include：uboot使用的头文件，configs目录存放与开发板相关的配置头文件；asm目录存放与cpu相关的头文件。
- Net：与网络协议相关的代码
- Tools：uboot的工具，如mkimage、crc等。

4，uboot的编译过程

1. 源码包的清理：和内核的清理命令一样。

2. 配置uboot要使用的开发板：**make +配置文件。**具体有哪些配置文件可以在Makefile中查看
3. 开始编译uboot：make。完成会在当前目录生成uboot.bin。

5，uboot命令

- Help：查看当前板上所有的uboot命令。

环境变量相关命令

- **Printenv**：查看环境变量，后跟环境变量名则只查看某个环境变量
- **Setenv 变量 值**：添加/改变环境变量值，**setenv 变量** 删除变量
- **Saveenv**：将当前所有定义变量存入flash

文件下载命令

Tftp：使用tftp要先配置好网络：ipaddr、serveraddr，分别为开发板ip和linux的ip。连通之后使用**tftp 开发板地址 文件名**进行下载。

内存操作

- Md：显示内存区的内容，**md[.b .w .l] address**，bwl是长度标识符，任选。
- Mm：修改内存，**mm[.b .w .l] address**，最后输入空格回车结束修改。

Flash相关命令

- **Nand info**：flash的大小等信息。
- **Nand erase start length**：擦除nand flash，将相应块全部位变为1
- **Nand write [内存地址][flash地址] length**：写入数据
- **Nand read [内存地址][flash地址] length**：读出数据

执行程序

- **Go addr [arg..]**：执行内存中的二进制代码，addr为开始地址。
- Bootm [addr [arg...]]：也是执行代码，可以不指定addr，因为有一个默认的地址。要求代码有固定格式的文件头。
- **Bdinfo**：显示开发板信息，如内存大小，时钟频率，mac地址等。

自动启动：

Setenv bootcmd tftp x0008000 uImage \; bootm c0008000

saveenv

五，嵌入式linux系统构建

1，嵌入式内核编译

1. 清理工作，make clean等命令
2. 配置内核：**make menuconfig ARCH=arm**。这里要将默认的x86变成arm。
3. 编译内核：**make uImage ARCH=arm CROSS_COMPILE=arm-linux-**

2，根文件系统制作

为创建的目录和文件建立数据

1. 创建目录：**mkdir bin dev etc lib proc sbin sys usr mnt tmp var;mkdir usr/bin usr/lib usr/sbin lib/modules**
2. Dev目录下：mknod -m 666 **console** c 5 1;mknod -m 666 **null** c 1 3
3. Etc目录下：构建基本的配置文件，可以从已有的复制，必须有的：inittab，profile，fstab，init.d目录

4. 编译内核模块：**make modules ARCH=arm CROSS_COMPILE=arm-linux-**
5. 安装内核模块到文件系统：make modules_install ARCH=arm INSTALL_MOD_PATH=/xxx/rootfs
6. 配置busybox：busybox会生成linux常用的命令和工具，配置和编译的过程和内核编译差不多。编译完成后执行安装make install

3，嵌入式文件系统

- 基于flash的文件系统

Flash(闪存)是嵌入式的主要存储媒介，有nor和nand两种。**Jffs2**文件系统主要用于nor flash，**Yaffs/yaffs2**主要用于nand flash。**Cramfs**是一种只读的文件系统，可靠性高。

- 基于ram的文件系统

Ramdisk是将一块固定大小的内存当做块设备加上一种文件系统使用，并非一个实际的文件系统。**Initramfs**出现在2.6内核中，基于内存，可以自动配置更多空间。

- 基于网络的文件系统

NFS主要用在嵌入式的开发调试阶段，内核移植到开发板后文件系统是通过网络从主机拷贝的，改变主机的文件系统，板上的也回改变，用于动态加入删除程序。

4，inittamfs文件系统制作

1. 配置linux内核，使之支持initramfs
2. 进入根文件系统，执行：ln -s ./bin/busybox init
3. 重新编译内核

Initramfs会将内核和根文件系统合并到一起形成镜像，这种方式不需要再单独编译文件系统去移植，只要将uimage移植即可。

六，内存管理子系统

1，地址类型：

- 物理地址：cpu地址总线寻址物理内存的地址信号
- 线性地址：虚拟地址，采用16进制表示的
- 逻辑地址：出现在汇编程序中的地址

段式管理：16位cpu：逻辑地址=基地址+偏移量；物理地址PA=段寄存器值*16+逻辑地址。

页式管理：线性地址被分为固定长度的组，是虚拟的。

物理页：将内存划分为固定长度的管理单元，是实际存在的。

Linux内核有限度的用了段式管理，完全利用了分页机制。因为linux中逻辑地址和线性地址是相同的，段式管理没有了意义。

2，进程地址空间

linux采用虚拟内存管理，每个进程有独立的进程地址空间，程序可使用比实际内存更大的地址空间。相同程序的虚拟地址相同，但是对应的物理地址不同。

3，内核内存分配：<linux/slab.h>

void *kmalloc(size_t size,int flags)：size为分配大小,flags为控制kmalloc的标识，取值为：

- **GFP_ATOMIC**：成功或失败，不会睡眠
- **GFP_KERNEL**：最常用，分配实际内存，分配不到会睡眠
- **_GFP_DMA**：DMA传输的内存区(16M以下的页帧)

- **_GFP_HIGHMEM**：分配高端内存(896M以上)

按页分配：分配大块的内存

- Get_zeroed_page(unsigned int flags):返回新页指针并将页面清零
- __get_free_page(unsigned int flags):同上，但不清零
- __get_free_pages(unint flags,unint order):分配若干连续页面，返回内存区指针，不清零。

释放内存：释放和分配的页面数目不同会导致系统错误

- Kfree(const void*ptr)
- Free_page(unlong addr)，free_pages(unlong addr,unlong order)

4，内核地址空间：不会改变，固定的：

- 直接映射区：线性地址=3G+物理地址
- 动态映射区：由vmalloc分配，线性空间连续，但物理空间不一定连续。
- 永久映射区：访问高端内存的，**alloc_page(__GFP_HIGHMEN)**分配高端内存页，**kmap**函数将分配的内存映射到该区域。
- 固定映射区：每个地址项服务于特定用途，映射关系不变。

5，内核链表：<linux/list.h>

内核链表据有双向循环链表功能，指针指向的是struct list_head结构而不是节点的首位置。这样指针类型是list_head，而不是节点类型，实现了**通用指针**。

内核链表操作：

- INIT_LIST_HEAD(list_head *head)：初始化链表头
- **List_add(list_head*new,list_head *head)**：head之后插入节点，new为整个节点中list_head对应的成员名
- list_add_tail(list_head*new,list_head*head)：head之前插入节点
- List_del(list_head *entry):删除节点。不能再list_for_each中遍历删除节点，可以使用**list_for_each_safe(pos,n,head)**
- **List_entry(ptr,type,member)**：提取数据结构，ptr为指向list_head的指针，type为数据域的结构，member为整个节点中list_head对应的成员名，返回值为指向数据域的指针。
- **List_for_each(list_head*pos,list_head*head)**：遍历链表，pos为返回当前位置。**此宏是for循环，使用要加大括号**。数据的比较都在大括号内，宏本身只是pos位置的移动。

6，内核定时器

Jiffies：每次时钟中断，jiffies加1，记录了自linux启动中断的次数，驱动程序常用jiffies计算事件的时间间隔。

内核定时器：控制某个函数在未来某个时间执行，只执行一次。定时器使用**struct timer_list**结构描述。操作函数有：

- **Init_timer(timer_list *timer)**:初始化定时器队列，但只会初始化两个内核使用的变量。其余参数要自己设置。
- Void add_timer(timer_list*timer):启动定时器
- Int del_timer(timer_list*timer):定时器超时前删除，超时后会自动删除。

七，进程管理子系统

1，进程四要素：可执行的程序、内核空间堆栈、内核中**task_struct**数据结构(pcb进程控制块)、独立的用户空间。Task_struct包含大量进程线程信息，重

要的有：

- **Pid_t pid** 进程号，最大10亿
- **volatile long state** 进程状态。TASK_RUNNING执行或就绪态；TASK_INTERRUPTIBLE可中断阻塞；TASK_UNINTERRUPTIBLE 不可中断阻塞；TASK_STOPPED中止状态；TASK_KILLABLE 睡眠状态，类似不可中断阻塞，但是可被SIGKILL唤醒；TASK_TRACED调试态；TASK_DEAD进程退出
- **Int exit_state**：进程退出时的状态。EXIT_ZOMBLE僵死进程；EXIT_DEADj僵死撤销状态。
- **Int prio**：优先级，数值越大，优先级越小。
- **Unsigned int policy**：进程调度策略。

Linux内核中**current**全局变量指向当前正在运行进程的task_struct

2，进程调度：调度策略、调度时机、调度步骤

调度策略：

- SCHED_NORMAL：普通的分时进程(CFS调度类)
- SCHED_FIFO：先入先出实时进程(实时调度类)
- SCHED_RR：时间片轮转的实时进程(实时调度类)
- SCHED_BATCH：批处理进程(CFS调度类)
- SCHED_IDLE：只在系统空闲时才被调度执行的进程。(CFS调度类)

调度时机：

- 主动式：内核中直接调用schedule()，为了等待资源暂停
- 被动式(抢占)：用户抢占(2.4、2.6)和内核抢占(2.6)。从内核空间返回用户空间时发生用户抢占；返回内核空间或解锁和软中断时发生内核抢占。**preempt_count**称为内核抢占计数，该值>0不能内核抢占。

调度标志：TIF_NEED_RESCHED，表示需要重新执行一次调度。一般，某进程耗尽时间片时、高优先级进程进入可执行态时设置。

调度步骤：schedule函数流程

清理当前运行的进程--->选择下一个运行进程(pick_next_task)-->设置新进程运行环境-->进程切换

3，系统调用

系统调用的函数个数可以在**arch/cpu平台/include/asm/unistd.h**找到。

工作原理：程序先用适当的值填充寄存器，再调用特殊指令跳转到内核某一固定位置，内核根据应用程序填充的值找到函数执行。

- 适当的值：每个系统调用的唯一编号，系统调用号。
- 特殊指令：intel中为0x80中断指令；arm为SVC指令
- 固定位置：arm中为ENTRY(vector_swi)，在entry-common.S中。
- 函数：sys_call_table系统调用表找到内核函数，arm在**calls.S**中，x86在**syscall_table.S**中

添加新的系统调用

- Kernel/sys.c中添加函数的实现：asmlinkage int sys_add()
- 在unistd.h添加系统调用号，格式参照已有的
- 向kernel/calls.S 或kernel/syscall_table.S添加代码，指向新实现的系统调用函数。

- 在include/linux/syscalls.h中添加函数的声明

4，proc文件系统：在用户态检查内核状态的机制

文件内容都是动态创建的，并不存在于磁盘上。可以自己编写程序添加一个/proc目录下的文件。**Struct proc_dir_entry**来描述proc下的文件和目录。要实现一个proc文件，首先创建一个proc_dir_entry，然后对其中的成员赋值即可。**<linux/proc_fs.h>**

- **创建proc文件**：create_proc_entry(char*name,mode_t mode,proc_dir_entry *parent)，parent为该文件的父目录。
- **创建proc目录**：proc_mkdir(char*name,proc_dir_entry*parent)
- **删除目录和文件**：remove_proc_entry(char*name,proc_dir_entry*parent)
- 读取：Read_func(char*buffer,char**stat,off_t off,int count,int*peof,void*data)：buffer为返回信息，stat和data不使用，off偏移量，count读取的字节数，peof为1表示读到文件尾
- 写入：Write_func(struct file*file,char*buffer,ulong count,void*data)：file一般忽略，buffer为要写入的数据，count为写入大小，data不用。

读取和写入函数都是要自己实现的，只有参数不用变，然后设置proc_dir_entry的read_proc和write_proc成员为实现的函数。

5，linux内核异常分析

Oops信息：内核异常的信息安装固定的格式显示在屏幕或系统日志中。分析步骤：错误原因提示--->调用栈(反汇编程序)--->寄存器。

八，字符设备驱动

1，驱动分类：

- 字符设备驱动(重点)：按字节访问，不能随机访问，drivers/char
- 网络接口驱动(重点)：发送和接收数据
- 块设备驱动：整块数据，linux可以传送任意数目，与字符设备的区别仅仅是与内核的接口不同。可以随机访问。

驱动程序安装：模块方式；直接编译进内核(将源程序拷贝到相应驱动目录下，修改Kconfig和Makefile)。**Kconfig**为menuconfig菜单中的配置选项。源代码放在哪儿，修改哪个目录下的Kconfig和Makefile

Linux是通过**设备文件**来使用驱动程序操作字符设备和块设备，设备文件在**/dev**下。

2，**设备号**：使用ls -l以c开头的为设备文件，逗号隔开的两个数字即为主次设备号。主设备号反映设备类型；次设备号区分同类型设备。**Dev_t**为设备号类型，高12位为主设备号，低20位为此设备号。**MAJOR(dev_t dev)**：取得主设备号；**MINOR(dev_t dev)**：取得次设备号。**MKDEV(major,minor)**，生成一个dev_t类型设备号。

静态申请设备号：根据Documentation/devices.txt，确定一个没有使用的主设备号；然后**register_chrdev_region**注册。

- **register_chrdev_region(dev-t from,unsigned count,char*name)**：from为设备号，count为申请使用设备号数目，name为设备名

动态分配设备号：alloc_chrdev_region函数。

- **alloc_chrdev_region(dev_t *dev,unsigned baseminor,unsigned count,char*name)**:dev是获取值的，baseminor为起始次设备号，次设备数目。
- **Unregister_chrdev_region(dev_t from,unsigned count)**：注销设备号，从from开始的count个设备号。

3，设备文件：

手工创建(shell)：**mknod filename type major minor**。Type 为设备类型。

自动创建：2.4内核中devfs_register函数；2.6内核udev(mdev)实现

- Busybox配置mdev，因为默认没有mdev
- 之后在驱动程序中，**class_create**创建一个class类
- **Device_create**创建对应的设备。这样加载驱动时会自动创建文件

4，字符设备驱动的数据类结构

- **Struct file**：一个打开的文件，重要成员：loff_t f_pos文件读写位置；struct file_operations*f_op
- **struct inode**：文件物理信息，一个文件可以对应多个file结构，但是只有一个inode，重要成员：dev_t dev
- **struct file_operations**：函数指针集合，定义可以进行的操作。这是驱动程序的主体。主要来自用户空间的buffer内核不能直接使用，要用**copy_from_user**和**copy_to_user**函数引用。

5，字符设备驱动注册

字符设备用**struct cdev**描述，注册步骤：分配cdev-->初始化cdev-->注册cdev-->设备注销。

- **Struct cdev* cdev_alloc(void)**：分配cdev，返回cdev指针
- **Cdev_init(struct cdev*cdev,struct file_operations*fops)**：初始化cdev
- **Cdev_add(struct cdev*p,dev_t dev,unsigned count)**：注册
- **Cdev_del(struct cdev*p)**：注销设备驱动。

6，**并发**：多个执行单元同时执行；**竞态**：并发执行单元对共享资源的访问导致的竞争状态。Linux通过semaphore机制和spin_lock机制进行调度。

信号量机制：用于内核的睡眠锁。<asm/semaphore.h>

- 定义信号量：**struct semaphore sem**；
- 初始化信号量：**sema_init(struct semaphore*sem,int val)**,val为初值。**init_MUTEX(struct semaphore*sem)**初始化为1；**init_MUTEX_LOCKED(struct semaphore*sem)**初始化为0。
- 定义和初始化可以一起完成，使用如下宏：DECLARE_NUTEX(name)、DECLARE_MUTEX_LOCKED(name)
- 获取信号量：**down(struct semaphore*sem)**将sem的值减1，获取不成功进入TASK_UNINTERRUPTIBLE状态。**Down_interruptible(struct semaphore*sem)**获取不成功进入TASK_INTERRUPTIBLE状态。**Down_killable(struct semaphore*sem)**获取不成功进入TASK_KILLABLE状态。
- 释放信号量：**up(struct semaphore*sem)**加1

自旋锁机制：不会睡眠，会一直占用cpu

- 初始化自旋锁：**Spin_lock_init(x)**
- 获取自旋锁：**spin_lock(lock)**、**spin_trylock(lock)**。后者失败时会立即返回。
- 释放自旋锁：**spin_unlock(lock)**

7，IOctl设备控制

Ioctl系统调用原型：**int ioctl(int fd,ulong cmd，...)**，fd为要控制的文件的描述符；cmd为控制命令，...为cmd的参数。

Ioctl驱动原型：**int(*ioctl)**在2.6.36以后的版本中不用了，使用**long(*unlocked_ioctl)**。

Unlocked_ioctl实现步骤：

1.
2. 不用检测的：
3. Copy_from_user,copy_to_user,get_user,put_user
4. 需要检测的：
5. __get_user,__put_user
6.
    - **_IO(type,nr)**：没有参数的命令
    - **_IOR(type,nr,datatype)**：从驱动读数据
    - **_IOW(type,nr,datatype)**：写数据到驱动
    - **_IOWR(type,nr,datatype)**：双向传送，type和number作为参数

1. 实现unlocked_ioctl：返回值：**-EINVAL**为没有匹配命令时返回值；参数：如果参数是指针，要检查地址是否有效，检测函数为：**int access_ok(int type,void*addr,ulong size)**：type为VERIFY_READ或VERIFY_WRITE，addr为指针，size为操作长度，返回1成功0失败.命令操作：使用switch来判断。几个宏：**_IOC_TYPE(cmd)**检测命令类型；**_IOC_NR(cmd)**检测命令序号;**_IOC_DIR(cmd)**检测命令方向；**_IOC_SIZE(cmd)**检测参数大小。

8，等待队列：类型wait_queue_head_t

- 定义并初始化：**DECLARE_WAIT_QUEUE_HEAD**(my_queue)
- 有条件睡眠：**wait_event(queue,condition)** condition为假进入TASK_UNINTERRUPTIBLE模式睡眠；**wait_event_interruptible(queue,condition)** 睡眠时进入TASK_INTERRUPTIBLE模式。**Wait_event_killable(queue,condition)**睡眠进入TASK_KILLABLE模式。
- 从等待队列唤醒进程：**wake_up(wait_queue_t*q)**唤醒所有进程；**wake_up_interruptible(wait_queue_t*q)**只唤醒interruptible的进程

9，阻塞型字符设备驱动

阻塞方式：阻塞进程进入睡眠，知道请求得到满足

非阻塞方式：使用**O_NONBLOCK**标识，系统返回**-EAGAIN**，不阻塞，该标志在file-->flags&O_NONBLOCK中判断。

阻塞判断这块使用while是为了应对终端信号等唤醒进程的情况。

10，Poll设备方法：对应select系统调用

Select用于多路监控，没有一个文件满足要求时，将阻塞进程。

Select(intmaxfsd,fd_set*readfds,fd_set*writefds,fe_set*execptfds,struct timeval*tineout):maxfd为描述符的范围，比最大文件描述符大1；readfds被读监控的描述符集；writefds被写监控的描述符集；execptfds被异常监控的描述符集；timeout定时器。Timeout为0时，不会阻塞立即返回；为NULL，会阻塞；为正整数，select在timeout时间内阻塞进程。

Select的返回值：正常返回满足要求的文件个数；没有满足的返回0；select被某个信号打断，返回-1并errno为EINTR；调用出粗返回-1并设置errno。

操作描述符集的宏：

- **FD_SET(int fd,fd_set*fdset)**：将fd添加到fdset中
- **FD_CLR(int fd,fd_set*fdset)**：将fd从fdset中清除
- **FD_ZERO(fd_set*fdset)**：清空fdset中所有的fd

- **FD_ISSET(int fd,fd_set*fdset)**：判断fd在fdset中是否变化

Poll设备方法：使用poll_wait将等待队列添加到poll_table；返回设备掩码：**POLLIN**(设备可读)、**POLLRDNORM**(数据可读)、**POLLOUT**(设备可写)、**POLLWRNORM**(数据可写)。

Poll方法只是做了一个登记，真正的阻塞发生在do_select函数

11，mmap设备方法：对应mmap系统调用

Mmap把文件内容映射到进程的虚拟内存空间，以后就可以通过指针对文件读取和修改，不需要readwrite操作：

Void*mmap(void*addr,size_t len,int prot,int flags,int fd,off_t offset)：addr一般NULL，port为映射区保护方式：PORT_EXEC(可执行)、PORT_READ、PORT_WRITE；flags为映射区特性：MAP_SHARED、MAP_PRIVATE。

解除映射：**int munmap(void*start,size_t length)**成功返回0，失败-1.

Linux内核使用**vm_area_struct**结构描述虚拟内存区域。Mmap驱动要做的是建立虚拟地址到物理地址的页表。

- 构建页表：**int remap_pfn_range(vm_area_struct*vma,ulong addr,ulong pfn,ulong size,pgprot prot)**，addr为虚拟地址起始值，pfn为物理地址页帧号(virt_to_phys(指针)>>PAGE_SHIFT得到)，size为映射区大小，prot为保护属性。

九，硬件访问

1，寄存器操作有副作用：读取某个地址可能改变该地址内容发生变化。只有X86有IO地址空间。

- IO端口：寄存器或内存位于IO空间
- IO内存：寄存器或内存位于内存空间

IO端口操作

- **Struct resource*request_region(ulong first,ulong n,char*name)**:申请IO端口，从first开始的n个，name为设备名。系统端口分配情况在/proc/ioports中。
- 访问端口：8位端口inb、outb；16位inw、outw；32位inl、outl
- 释放端口：**void release_region(ulong first,ulong n)**

IO内存操作

- 申请内存：struct resource*request_mem_region(ulong start,ulong len,char*name)len为IO内存的长度，返回物理地址
- 映射内存：void*ioremap(ulong phys_addr,ulong size)将物理地址映射为虚拟地址
- 访问内存：ioread8，ioread16，ioread32；iowrite8，iowrite16...
- 释放内存：void*iounmao(*addr),void release_mem_region(ulong start,ulong len)解除映射并释放IO内存

2，混杂设备驱动：

混杂设备：都是字符设备，主设备都是10。混杂设备使用**strut miscdevice**描述

注册混杂设备驱动：int misc_register(miscdevice*misc)

注销混杂设备驱动：misc_deregister(miscdevice*misc)