回顾：
1.linux内核并发和竞态之原子操作
  特点：
  能够解决所有的竞态问题
  位原子操作：对共享资源进行位操作,考虑竞态问题
  整形原子操作：对共享资源进行整型操作,考虑竞态问题
  数据类型：atomic_t(类比成int)
  总结：操作务必要使用内核提供的相关函数

2.linux内核等待队列机制
  产生根本原因：外设的处理速度远远慢速CPU
  能够让进程随时随地休眠,随时随地被唤醒
  编程步骤：
  1.定义初始化等待队列头(全局变量)
  2.定义初始化装载休眠进程的容器(局部变量)
    current：内核全局变量,指向"当前进程"
  3.添加休眠进程到等待队列中
  4.设置当前进程的休眠状态
  5.进入真正的休眠状态,释放CPU资源,等待被唤醒
    注意：唤醒的方法
  6.一旦进程被唤醒,设置进程的状态为运行,并且将进程从
    等待队列中移除
  7.一般要判断进程唤醒的原因
  8.驱动主动唤醒的方法
    wake_up/wake_up_interruptible
  9.务必掌握按键驱动包括去抖动

3.linux内核等待队列编程方法2：
  编程步骤：
  1.定义初始化等待队列头(构造鸡妈妈)
    wait_queue_head_t wq;
    init_waitqueue_head(&wq);
  2.调用以下两个方法即可实现进程的休眠
    wait_event(wq, condition);//切记：此乃宏
    说明：
    wq：等待队列头
    condition:如果condition为真,进程不会休眠,立即返回,即硬件设备可用(可读或者可写)
              如果condition为假,进程将进入不可中断的休眠状态,等待被唤醒,即硬件设备不可用

    或者
    wait_event_interruptible(wq, condition);
    说明：
    wq：等待队列头
    condition:如果condition为真,进程不会休眠,立即返回,即硬件设备可用(可读或者可写)
              如果condition为假,进程将进入可中断的休眠状态,等待被唤醒,即硬件设备不可用
    总结：
    以上两个宏等价于编程方法1的第2步~第7步

  3.唤醒的方法
    wake_up/wake_up_interruptible

  方法2的编程框架：

```
//休眠的地方
xxx(...) {
	//起初condition为假
	wait_event_interruptible(wq, condition);
	//一旦被唤醒
	condition设置为假
}

//唤醒的地方
yyy(...) {
	condition设置为真
	//唤醒休眠的进程
	wake_up_interruptible(&wq);
}
```

案例：利用等待队列编程方法2实现按键驱动

****

3.linux内核内存分配相关内容

3.1.linux内核内存的划分(了解)

明确：不管是在用户空间还是在内核空间,软件一律不允许访问
硬件外设的物理地址,要想软件访问硬件外设的物理地址
必须将硬件外设的物理地址映射到用户虚拟地址或者
内核虚拟地址,将来软件只要访问用户虚拟地址或者
内核虚拟地址就是在访问对应的物理地址

特例：uclinux操作系统软件访问的地址都是物理地址

回顾用户空间3G虚拟内存的划分

总结：用户空间3G虚拟内存地址和物理地址之间的映射
属于动态映射(用到时进行映射,不用时将映射关系解除)

4G虚拟地址划分为用户虚拟地址和内核虚拟地址

用户虚拟地址空间范围：0~0xbfffffff

内核虚拟地址空间范围：0xc0000000~0xffffffff

linux内核1G虚拟内存地址和物理内存地址的映射属于一一映射,
即内核在启动的时候就已经将物理内存地址和内核1G虚拟内存地址
建立好映射关系,将来软件无需再次建立映射,直接访问即可,内存
的访问效率最高：

物理内存地址	内核虚拟地址
0x0	0xC0000000
0x1	0xC0000001
0x2	0xC0000002
...	...
1G	1G

一一映射存在线性关系！

问题：如果采用一一映射,linux内核最多只能访问1G的物理内存
如何让内核访问到所有的物理内存地址呢？又要兼顾内存的访问
效率

答：linux内核将1G虚拟内存划分若干个区域

X86划分：

直接内存映射区：
大小为896M
内核在启动的时候,将直接内存映射区的内核虚拟内存地址
和物理内存地址进行一一映射,这块内存区域的效率最高
又称低端内存

动态内存映射区：

默认大小为120M
内核代码需要访问某块物理内存时,内核动态建立动态内存映射区
的虚拟内存和物理内存的映射,使用完毕,一定要记得解除
地址映射,否则造成内存泄漏

永久内存映射区：
固定内存映射区：
固定=永久
大小都为4M
如果频繁的访问某块物理内存,考虑到效率,可以将
物理内存映射到永久或者固定内存的虚拟内存上
前者映射时会导致休眠,不能用于中断上下文
后者不会导致休眠

高端内存=动态内存映射区+永久+固定

TPAD开发板,linux内核1G虚拟内存的划分：
启动开发板,观察内核打印信息,找到1G内核虚拟内存的划分：
Virtual kernel memory layout:
区域名　　内核起始地址　内核结束地址　区域大小

异常向量表
vector　: 0xffff0000 - 0xffff1000　(　 4 kB)

固定内存映射区
fixmap　: 0xfff00000 - 0xfffe0000　(896 kB)

DMA内存映射区
DMA　　: 0xff000000 - 0xffe00000　(　14 MB)

动态内存映射区
vmalloc : 0xec800000 - 0xfc000000　(248 MB)

直接内存映射区
lowmem　: 0xc0000000 - 0xec600000　(710 MB)

//模块加载区域
modules : 0xbf000000 - 0xc0000000　(　16 MB)

//初始化段
　.init : 0xc0008000 - 0xc0037000　(188 kB)
//代码段
　.text : 0xc0037000 - 0xc0832000　(8172 kB)
//数据段
　.data : 0xc0832000 - 0xc0886960　(339 kB)

切记：一个物理地址可以有多个虚拟地址(用户的和内核的)
　　但是一个虚拟地址不能对应多个物理地址

3.2.linux内核内存分配的函数
回忆应用内存分配：
int a; //局部,全局的(初始化和没初始化)
malloc/free //堆

内核分配函数一：
kmalloc/kfree

函数原型：
void *kmalloc(int size, gfp_t flags)
功能：
1.从直接内存映射区分配内存
  访问效率高
2.分配内存大小最小32字节,最大4MB
3.分配的内核虚拟内存和物理内存都是连续的

参数：
size:指定分配内存的大小,单位为字节
flags:指定分配内存时的行为标志：
      GFP_KERNEL:告诉内核,请努力将这次内存分配搞定
                 如果内存不足,会导致休眠,所以不能用在
                 中断上下文
      GFP_ATOMIC:如果内存不足,不会进行休眠,而是立即返回
                 可以用在中断上下文
返回值：返回分配内核虚拟内存的首地址
例如：
```
void *addr;
addr = kmalloc(100, GFP_KERNEL);
if (addr == NULL)
     return -ENOMEM;

memcpy(addr, "hello,world", 12);
```

不再使用时,记得要释放内存：
```
void kfree(void *addr)
```

内核内存分配函数二：
__get_free_pages/free_pages
函数原型：
```
unsigned long __get_free_pages(gfp_t flags, int order)
```
功能：
1.从直接内存映射区分区
2.物理和虚拟内存上都是连续的
3.最大4MB

参数：
flags：表示分配内存时的行为
       GFP_KERNEL:告诉内核,请努力将这次内存分配搞定
                  如果内存不足,会导致休眠,所以不能用在
                  中断上下文
       GFP_ATOMIC:如果内存不足,不会进行休眠,而是立即返回
                  可以用在中断上下文
order:order=0,分配1页
      order=1,分配2页
      order=2,分配4页
      order=3,分配8页
      ...
返回值：返回分配内存的首地址,注意数据类型的转换

```
unsigned long addr;
addr = __get_free_pages(GPF_KERNEL, 2);
memcpy((void *)addr, "hello,world", 12);
```

内存不再使用时,记得要释放内存：

```
void free_pages(unsigned long addr, int order);
```

内核内存分配函数三：
vmalloc/vfree
```
void *vmalloc(int size);
```
函数功能：
1.从动态内存映射区分配内存
2.内核虚拟地址是连续的,但是对应的物理地址不一定连续
  动态内存映射区的内存访问效率低
3.理论默认最大分配120M
4.同样会导致休眠,所以不能用在中断上下文

释放内存：
```
void vfree(void *addr)
```

内核内存分配另类方法：
在内核启动参数中,添加vmalloc=? (例如vmalloc=250M)表示
内核启动时,将动态内存映射区的大小由原先的120M扩展到
250M

案例：跳转动态内存映射区的大小
1.启动开发板的linux系统,观察内核打印信息,首先确认
  直接内存映射区和动态内存映射区的大小(默认832M和120M)

2.重启开发板,进入uboot,执行：
```
setenv bootargs root=/dev/nfs nfsroot=192.168.1.8:/opt/rootfs
ip=192.168.1.110:192.168.1.8:192.168.1.1:255.255.255.0::eth0:on init=/linuxrc
console=ttySAC0,115200 vmalloc=250M
saveenv
boot
cat /proc/cmdline
```
继续观察内核的启动信息,再次查看直接内存映射区和动态内存映射区的大小是否有变化

内核内存分配另类方法：
在内核启动参数中,添加mem=? (例如mem=8M)表示
内核启动时,将物理内存的最后8M预留出来,将来给驱动单独使用
但是驱动使用时,必须利用ioremap函数进行地址映射,将最后的
8M物理内存和内核的动态内存映射区的虚拟内存做映射,一旦完成
映射,将来驱动访问映射的内核虚拟内存就是在访问最后的8M物理
内存

****
4.大名鼎鼎的ioremap函数
4.1.明确
嵌入式系统,CPU访问某个外设,必须要先获取到这个外设的基地址
只要有了这个基地址,将来就可以以地址指针的形式访问：
```
*(unsigned long *)0x20000000 = 0x55;
```

4.2.明确
linux系统不管是用户空间还是内核空间,一律不允许访问外设的物理基地址
,要想访问,必须将设备的物理地址映射到用户虚拟地址上或者
内核虚拟地址上,将来访问映射的用户或者内核虚拟地址就是在
访问对应的物理地址

4.3.问：如何将外设的物理地址映射到内核的虚拟地址上呢?
  答：利用大名鼎鼎的ioremap函数

4.4.ioremap使用
函数原型：
void *ioremap(unsigned long phys_addr, int size)
函数功能：
1.将外设的物理地址映射到内核的虚拟地址上
2.映射到内核1G的动态内存映射区

phys_addr:硬件外设的起始物理地址
size:映射的"物理内存"的大小

返回值：返回的映射的内核起始虚拟地址

参考代码：
LED1,LED2硬件寄存器信息：
GPC0CON:起始物理地址0xE0200060,大小4字节
GPC0DATA:起始物理地址0xE0200064,大小4字节

映射如下：
```
unsigned long *gpiocon *gpiodata;
gpiocon = ioremap(0xE0200060, 4);
gpiodata = ioremap(0xE0200064, 4);
```

或者：
由于发现两个硬件寄存器的内存空间都是连续的,物理上连续：
```
gpiocon = ioremap(0xE0200060, 8);
gpiodata = gpiocon + 1;

*gpiocon &= ~(0xf << 12);
*gpiocon |= (1 << 12);
...
```

内存不再使用时,记得要释放内存,解除地址映射
```
iounmap(void *addr);
```

例如：
```
iounmap(gpiocon);
```

案例：不再使用GPIO库函数,实现LED驱动,接口采用ioctl

小项目：寄存器编辑器
用户需求：随意能够查看和修改CPU的任何一个寄存器
思路：
应用测试思路：
```
./regeditor w regaddr regdata
./regeditor r regaddr //打印寄存器的值
```
例如：
```
./regeditor w 0xE0200080 0x11000 //将0x11000写入寄存器0xE0200080
./regeditor r 0xE0200080 //打印寄存器0xE0200080的值
```

应用程序编程思路：
```
struct reg_info {
      unsigned long regaddr;
      unsigned long regadata;
};
```

```
#define REG_WRITE     0x100001 //写寄存器
#define REG_READ      0x100002 //读寄存器

struct reg_info reg;

if (!strcmp) { //写寄存器
      reg.regaddr = strtoul(...);
      reg.regdata = strtoul(...);
      ioctl(fd, REG_WRITE, &reg);
} else { //读
      reg.regaddr = strtoul(...);
      //reg.regdata = ?
      ioctl(fd, REG_READ, &reg); //读寄存器
      printf("寄存器值=%#x\n", reg.regdata);
}
```

底层驱动ioctl接口：

```
 struct reg_info {
      unsigned long regaddr;
      unsigned long regadata;
};

#define REG_WRITE     0x100001 //写寄存器
#define REG_READ      0x100002 //读寄存器

reg_ioctl(...) {
      unsigned long *regbase;
      struct reg_info reg;
      copy_from_user(&reg, (struct reg_info*)arg, 8);
      regbase = ioremap(reg.regaddr, 4);
      switch(cmd) {
              case REG_WRITE:
              *regbase = reg.regdata;
              break;
              case REG_READ:
              reg.regdata = *regbase;
              copy_to_user((struct reg_info *)arg, &reg, 8);
              break;
      }
      iounmap(regbase);
}
```

检验：利用寄存器编辑器软件开关蜂鸣器