回顾：

1.linux内核等待队列编程方法2

工作队列：底半部一种实现方法,延后执行的一种手段
延后处理函数工作在进程上下文,可以休眠操作

等待队列：实现进程在内核中随时随地休眠,随时随地唤醒

编程实现两种形式：

方式1：

1.定义初始化等待队列头
2.定义初始化装载休眠进程的容器
3.添加
4.设置为休眠状态
5.进入真正的休眠，等待被唤醒
6.一旦被唤醒,设置为运行,移除
7.判断唤醒的原因
8.在某个地方进行唤醒,一般在中断处理函数中唤醒

方式2：

1.定义初始化等待队列头
2.直接进入休眠
3.在某个地方进行唤醒,一般在中断处理函数中唤醒
4.注意,编程口诀：唤醒前设置为真;唤醒以后设置为假

2.linux内核内存分配相关内容

2.1.linux内核1G虚拟内存的划分

划分的本质目的：一方面让内核访问到所有的物理地址
另一方面让内核的内存访问效果提高

2.2.linux内核内存分配的方法

kmalloc/kfree
__get_free_pages/free_pages
注意：GFP_KERNEL/GFP_ATOMIC
vmalloc/vfree
启动参数添加vmalloc=?
启动参数添加mem=?

2.3.linux内核地址映射的函数:ioremap

明确：软件一律不允许直接访问硬件外设的物理地址
问：如何将外设的物理地址映射到内核的虚拟地址上？
答：ioremap

3.问：如何将外设的物理地址映射到用户空间的虚拟地址上呢？
如果一旦完成这种映射,将来用户应用程序只需要在用户
空间访问映射的用户虚拟地址就是在访问实际的硬件物理地址？
也就是原先驱动中的这些代码：

```
*gpiocon &= ~((0xff << 12)); //gpiocon此时就是用户虚拟地址
...
```

这些代码都应该在应用程序中完成！

答：利用mmap

3.1.回忆：应用程序mmap系统调用函数的使用：

```
void *addr;
int fd = open("a.txt", O_RDWR);

addr = mmap(0, 0x1000, PROT_READ|PROT_WRITE,
            MAP_SHARED, fd ,0);
```

说明：

功能：将文件a.txt映射到当前进程的3G虚拟内存的MMAP内存映射区中的

某块虚拟内存上,一旦完成映射,将来访问映射的用户虚拟内存就是
在访问实际的文件;
明确:
"文件":太抽象,文件实际代表的是一个硬件设备(硬盘),通常说
访问文件,其他就是在说访问硬盘的物理存储地址空间,由于linux系统
不允许直接访问硬盘的物理地址,需要进行映射,利用mmap就能够
将硬盘的物理地址映射到用户的虚拟地址上,将来访问映射的用户
虚拟地址就是在访问实际的硬盘物理地址

第一个参数0:让内核帮你在MMAP内存映射区中找一块空闲的虚拟内存
用来映射外设的物理地址

第二个参数0x1000:要映射的用户虚拟内存区域的大小
一般为页面大小的整数倍
第三个参数PROT_READ|PROT_WRITE:将来给这块虚拟内存设置一个访问权限
...
返回值addr:保存的就是内存映射区域的首地址

3.2.mmap系统调用的过程做了哪些工作:
1.当应用程序调用mmap系统调用函数
2.首先调用到C库的mmap函数的定义
3.C库的mmap函数将做两件事:
1.保存mmap系统调用号到R7寄存器
2.调用svc指令触发软中断异常
至此进程由用户态陷入内核态
4.进入内核的软中断处理的入口,做两件事:
1.从R7寄存器取出之前保存的系统调用号
2.以系统调用号为下标在内核的系统调用表中
找到对应的内核函数sys_mmap
调用此函数;
5.内核的sys_mmap同样做三件事:
1.内核会在当前进程的3G的MMAP内存映射区中找一块
空闲的虚拟内存区域来映射硬件的物理地址
2.一旦内核找到了空闲内存区域,内核用struct vm_area_struct
结构体帮你创建一个对象来描述这块空闲内存区域的属性(起始地址,大小,访问权限等)

```
struct vm_area_struct {
  unsigned long vm_start;//空闲虚拟内存区域的起始地址
  unsigned long vm_end;//结束地址
  pgprot_t vm_page_prot;//访问权限
  unsigned long vm_pgoff; //偏移量
  ...
};
```

总结:应用程序的mmap的返回值addr就是vm_start
此数据结构对应的对象由内核创建
3.最后内核的sys_mmap调用底层驱动的mmap接口,切记并且
内核将创建的vm_area_struct结构体对象指针传递给底层驱动
的mmap接口,底层驱动的mmap接口通过参数获取空闲虚拟内存
区域的属性!

6.底层驱动的mmap接口函数只做仅作一件事,将已知的空闲虚拟内存
和外设的物理内存建议映射即可,一旦建议映射,将来对硬件的访问
过程都是在应用程序中完成!
底层驱动mmap接口的角色类似“媒婆”,只负责映射(牵线),
具体的硬件访问操作跟mmap没有任何关系!

7.底层驱动mmap接口

```
struct file_operations {
     int (*mmap)(struct file *file,
                      struct vm_area_struct *vma)
};
```

功能：

此接口函数只做仅作一件事：将已知的用户虚拟地址和已知的
物理地址建立映射

参数：

file:文件指针

vma:指向内核描述空闲虚拟内存区域的属性对象
vma指向的对象由内核创建
对象描述内核找到的空闲用户虚拟内存区域的属性

底层驱动mmap接口只需调用一下函数即可完成最终的映射：

```
int remap_pfn_range(struct vm_area_struct *vma,
                      unsigned long from,
                      unsigned long to,
                      unsigned long size,
                      pgprot_t prot);
```

功能：将已知的用户虚拟地址和已知的物理地址进行映射

参数：

vma: 指向内核描述空闲虚拟内存区域的属性对象

from:已知要映射的用户虚拟的起始地址vm_start

to:已知要映射的物理地址,注意要将物理地址>>12

size:映射的用户虚拟内存的大小

prot：用户虚拟内存区域的属性

切记切记：mmap地址映射时,地址(不管是用户虚拟地址还是物理地址)
必须是页面大小的整数倍！

例如：

GPC1CON:0xE0200080

GPC1DATA:0xE0200084

注意：不能拿0xE0200080或者0xE0200084去做地址映射,这两个地址
不是页面大小的整数倍！所以拿0xE0200000做地址映射即可：

物理地址	用户虚拟地址
0xE0200000	vm_start
0xE0200080	vm_start + 0x80
0xE0200084	vm_start + 0x84

案例：利用mmap实现LED驱动

切记：利用mmap进行GPIO操作(输入和输出操作时),把cache关闭

```
vma->vm_page_prot = pgprot_noncached(vma->vm_page_prot);
```

****

3.linux内核分离思想

3.1.案例：将TPAD开发板的LED1,LED2对应的GPIO更换成GPF1_4,GPF1_5
并且实现其驱动程序

分析：

明确：一个完整的硬件设备驱动必然包含两个内容：
纯硬件信息和纯软件信息(软件操作硬件)

如果要实现以上驱动,只需将昨天利用ioremap实现的LED驱动改造即可
分析改动的部分；

原先：

LED1->GPC1_3

LED2->GPC1_4
0xE0200080,0xE0200084
现在
LED1->GPF1_4
LED2->GPF1_5
0xE020.... 0xE020....

总结：
1.如果是硬件发生变化,之前的LED驱动修改的部分还是蛮多
2.可以采用C的#define来优化此驱动,但是#define不能给硬件
  添加额外的属性

3.问：如何让这个LED驱动具有更好的可移植性呢(如果将来仅仅
是硬件发生变化,只需修改硬件信息,软件一旦写好,就无需改动,
甚至都不用去看)
  答：采用linux内核的分离思想

3.2.linux内核分离思想
linux内核分离思想就是将一个完整的硬件驱动的纯硬件信息和
纯软件信息彻底分开,一旦驱动的软件部分写好,将来无需在改动,
硬件变化,只需修改硬件信息即可,将来驱动开发者的重心放在硬件
部分即可.

linux内核分离思想的实现基于platform机制

3.3.linux内核的platform机制
机制实现原理：参见PPT

3.4.linux设备驱动采用platform机制实现,驱动开发者只需关注
两个数据结构：
struct platform_device
struct platform_driver
问：如何使用？

1.struct platform_device的使用操作步骤：
```
  struct platform_device {
      const char      * name;
      int             id;
      u32             num_resources;
      struct resource * resource;
      ...
  };
```
  功能：描述硬件外设的纯硬件信息
  成员说明：
  name:描述硬件节点的名称,将来用于匹配
  id：硬件节点的编号,如果dev链表中仅有一个硬件节点,id=-1
      如果dev链表上有同名的多个硬件节点,id=0,1,2...
  resource:装载纯硬件信息
```
          struct resource {
             unsigned long start;
             unsigned long end;
             unsigned long flags;
             ...
          };
```
          功能：描述纯硬件信息
          start:硬件的起始信息

end:硬件的结束信息
flags:硬件的类型：
IORESOURCE_MEM：地址类信息
IORESOURCE_IRQ: IO类信息
num_resources:纯硬件信息对象的个数

配套函数：
int platform_device_register(&硬件节点对象);
功能：注册硬件节点到dev链表
1.添加节点到dev链表
2.内核帮你遍历drv链表,取出每一个软件节点进行匹配
3.一旦匹配成功,内核调用软件节点的probe函数
4.顺便把匹配成功的硬件节点的首地址给probe函数
完成硬件和软件的再次结合

void platform_device_unregister(&硬件节点对象);
功能：将硬件节点从dev链表上删除
此时内核会调用remove函数

2.struct platform_driver
```
struct platform_driver {
    int (*probe)(struct platform_device *pdev);
    int (*remove)(struct platform_device *pdev);
    struct device_driver driver;
    ...
};
```
功能：描述驱动的纯软件信息
成员：
probe:当硬件和软件匹配成功,内核调用此函数
形参pdev指向匹配成功的硬件节点
注意：它是否被调用至关重要！只有此函数被调用
一个完成的驱动诞生！
remove：当卸载软件或者硬件节点时,内核调用此函数
形参pdev指向匹配成功的硬件节点
注意：remove和probe永远是死对头！

driver：只需关注其中的name字段,将来用于匹配

配套函数：
int platform_driver_register(&软件节点对象);
功能：注册软件节点到drv链表
1.添加节点到drv链表
2.内核帮你遍历dev链表,取出每一个硬件节点进行匹配
3.一旦匹配成功,内核调用软件节点的probe函数
4.顺便把匹配成功的硬件节点的首地址给probe函数
完成硬件和软件的再次结合

void platform_driver_unregister(&软件节点对象);
功能：将软件节点从drv链表上删除
此时内核会调用remove函数

案例：利用platform机制优化LED驱动
实验步骤：
cd /home/drivers/
insmod led_drv.ko
insmod led_dev.ko

```
rmmod led_dev
rmmod led_drv
...
```

项目实施：
project.doc文档
根据文档从制作文件系统开始
内容：
制作根文件系统
tslib移植
QT移植