

南 京 理 工 大 学

软 件 课 程 设 计 三

姓 名：蒋旭钊 学 号：918106840727

学院(系): 计算机科学与工程学院

专 业：计算机科学与技术

课 程：软件课程设计III

2021年 12月

1. 内核模块

在内核模块上，我在JXZInnerModule目录下编写了Process.c文件，实现了打印当前系统的所有进程的程序，并最终编译成Process.ko模块注入了系统中，最后将固件打包配置到带有Padavan的YK-L1路由器上验证成功。

这里有一个技巧，我在/root/test/padavan-ng/trunk/linux-3.4.x/drivers目录下建立自己的JXZInnerModule目录。打开trunk目录下的build_firmware.sh的源码我们可以看到，这里主要根据.config里面已有的各种CONFIG配置进行和YK-L1硬件挂钩的二进制流烧制，最后可以注意到有make命令，这是根据trunk文件夹里面的Makefile文件相关的，主要进行二进制流烧制前各类modules的编译工作，其中比较重要的代码有：

```
ARCH_CONFIG = $(ROOTDIR)/vendors/config/mips/config.arch
```

它在config.arch里面指定了最终编译的文件是和MIPS系统有关的。

```
.PHONY: modules
```

它定义了Makefile要编译的各类模块，其中会根据各个模块中的.config文件进行配置，.config文件又和各个文件夹中的Kconfig文件有关，上面的drivers驱动模块目录下也包含着这一系列文件。而Kconfig文件可以让我们在linux-3.4.x目录下使用menuconfig指令调出模块选择的操作，最终可以实现模块的移植。

所以最终我们只需要在JXZInnerModule下编写Process.c的Makefile和Kconfig文件，然后在drivers目录下修改Makefile和Kconfig文件加上我们刚刚的JXZInnerModule目录和其下的Kconfig文件，就能最终打包编译成集成了我们的模块的固件。

```

1 #include <linux/init.h>
2 #include <linux/module.h>
3 #include <linux/kernel.h>
4 #include <linux/sched.h>
5
6 // 初始化函数
7 static int hello_init(void)
8 {
9     struct task_struct *p; //Linux内核的进程控制块是task_struct结构体，所有运行在系统中的进程都以task_struct链表的形式存在内核中
10    printk(KERN_ALERT "名称\t进程号\t状态\t父进程号\t");
11    for_each_process(p) //for_each_process是一个宏，在sched.h里面定义：是从init_task开始遍历系统所有进程，init_task是进程结构链表头。
12    {
13        if(p->mm == NULL){ //对于内核线程，mm为NULL
14            printk(KERN_ALERT "%16s\t%-6d\t%-6d\t%-6d\t%-6d\n", p->comm, p->pid, p->state, p->normal_prio, p->parent->pid);
15        }
16    }
17    return 0;
18 }
19 // 清理函数
20 static void hello_exit(void)
21 {
22    printk(KERN_ALERT "GoodBye JXZ!\n");
23 }
24
25 // 函数注册
26 module_init(hello_init);
27 module_exit(hello_exit);
28
29 // 模块许可申明
30 MODULE_LICENSE("GPL");
31

```

图1. Process.c文件

```

1  menu "JXZINNERMODULE Driver "
2
3  config JXZINNERMODULE
4      tristate "JXZInnerModule test"
5      default m
6
7  endmenu
8

```

图2. JXZInnerModule下的Kconfig文件

```

1 obj-$(CONFIG_JXZINNERMODULE) += Process.o
~
~
~
~
~

```

图3. JXZInnerModule下的Makefile文件

```

129 source "drivers/clk/Kconfig"
130
131 source "drivers/hwspinlock/Kconfig"
132
133 source "drivers/clocksource/Kconfig"
134
135 source "drivers/iommu/Kconfig"
136
137 source "drivers/remoteproc/Kconfig"
138
139 source "drivers/rpmsg/Kconfig"
140
141 source "drivers/virt/Kconfig"
142
143 source "drivers/devfreq/Kconfig"
144
145 source "drivers/JXZInnerModule/Kconfig"
146
147 endmenu
"Kconfig" 147L, 2413C

```

图4. drivers下的Kconfig文件

```

121 obj-$(CONFIG_VLYNQ) += vlynq/
122 obj-$(CONFIG_STAGING) += staging/
123 obj-y += platform/
124 obj-y += ieee802154/
125 #common clk code
126 obj-y += clk/
127
128 obj-$(CONFIG_HWSPINLOCK) += hwspinlock/
129 obj-$(CONFIG_NFC) += nfc/
130 obj-$(CONFIG_IOMMU_SUPPORT) += iommu/
131 obj-$(CONFIG_REMOTEPROC) += remoteproc/
132 obj-$(CONFIG_RPMSG) += rpmsg/
133
134 # Virtualization drivers
135 obj-$(CONFIG_VIRT_DRIVERS) += virt/
136 obj-$(CONFIG_HYPERV) += hv/
137
138 obj-$(CONFIG_PM_DEVFREQ) += devfreq/
139
140 #put the InnerModule into Makefile -- jxz
141 obj-$(CONFIG_JXZINNERMODULE) += JXZInnerModule/
"Makefile" 141L, 4017C

```

图5. drivers下的Makefile文件

通过在 linux-3.4.x 下执行 make menuconfig 命令，选取我们的 JXZInnerModule test 模块：

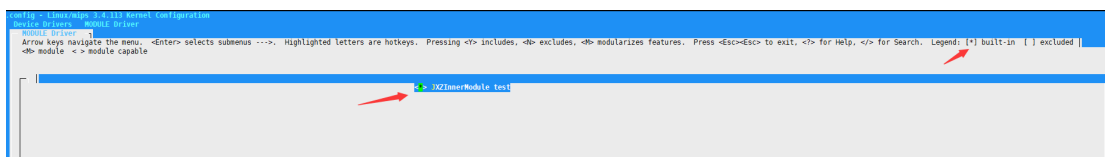


图6. Linux-3.4.x下的menuconfig界面

最终我们用路由器进行烧制，得到的结果如下，这里有一个区别，如果我们在编写Process.ko的Kconfig文件时，如果我们代码写“default m”代表最终模块是需要我们自己插入的。在Administration的Console下面我们通过find命令找到Process.ko模块，然后insmod进系统，通过lsmod发现已经插入以后，就可以在System Log中看到所有进程信息。最后用rmmod命令可以卸载模块，也可以看到打印信息。

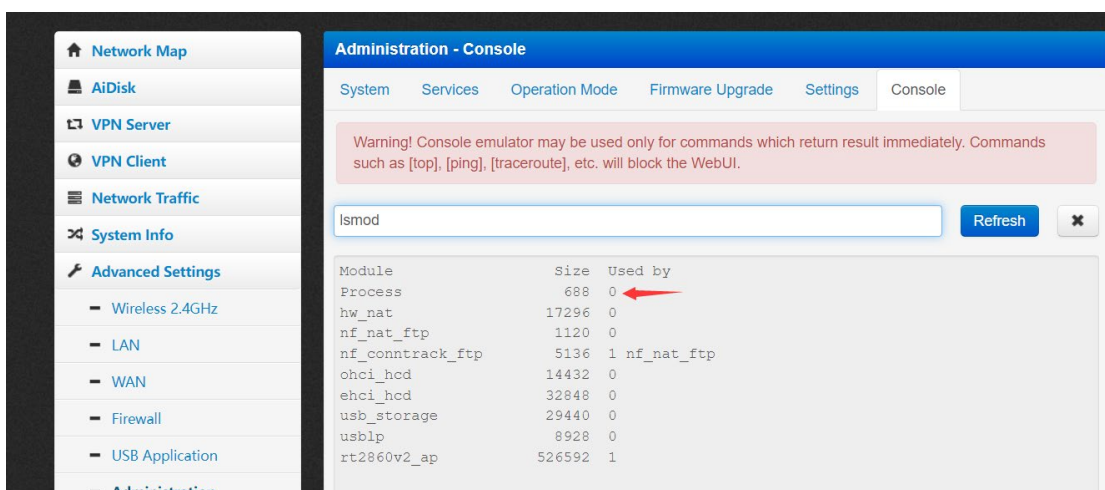


图7. Padavan下lsmod(default m)

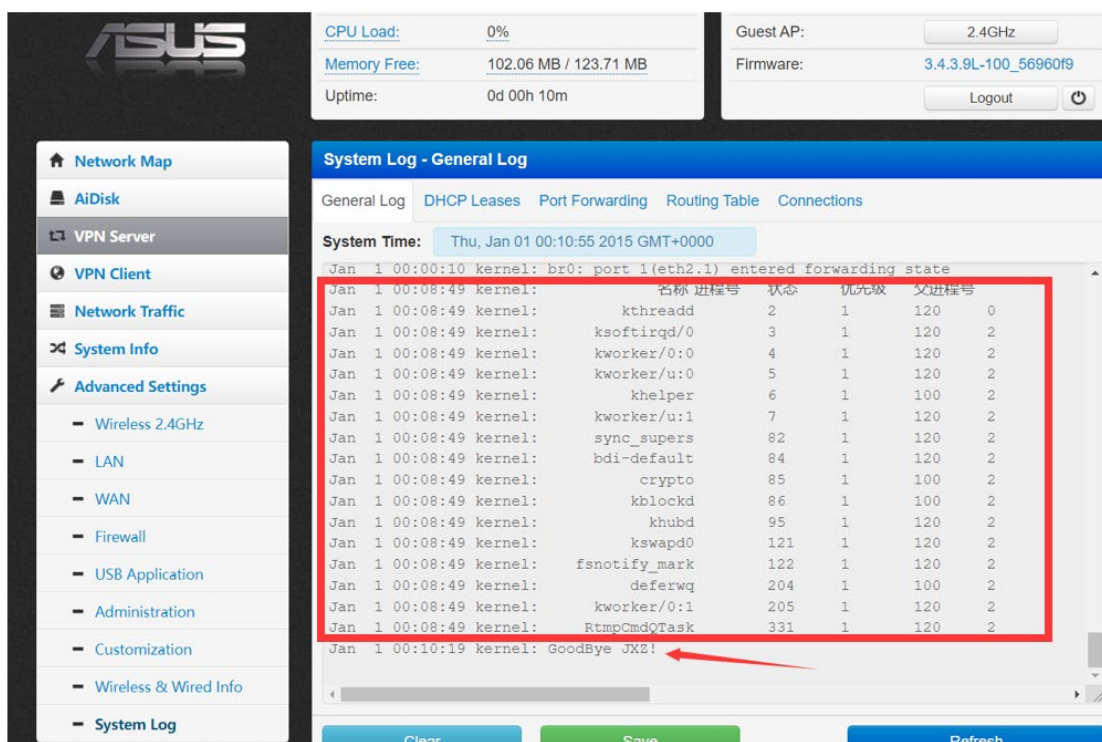


图8. Padavan下的System Log(default m)

如果我们在Kconfig中“default y”则表示模块已经built-in了，可以随系统一起启动。

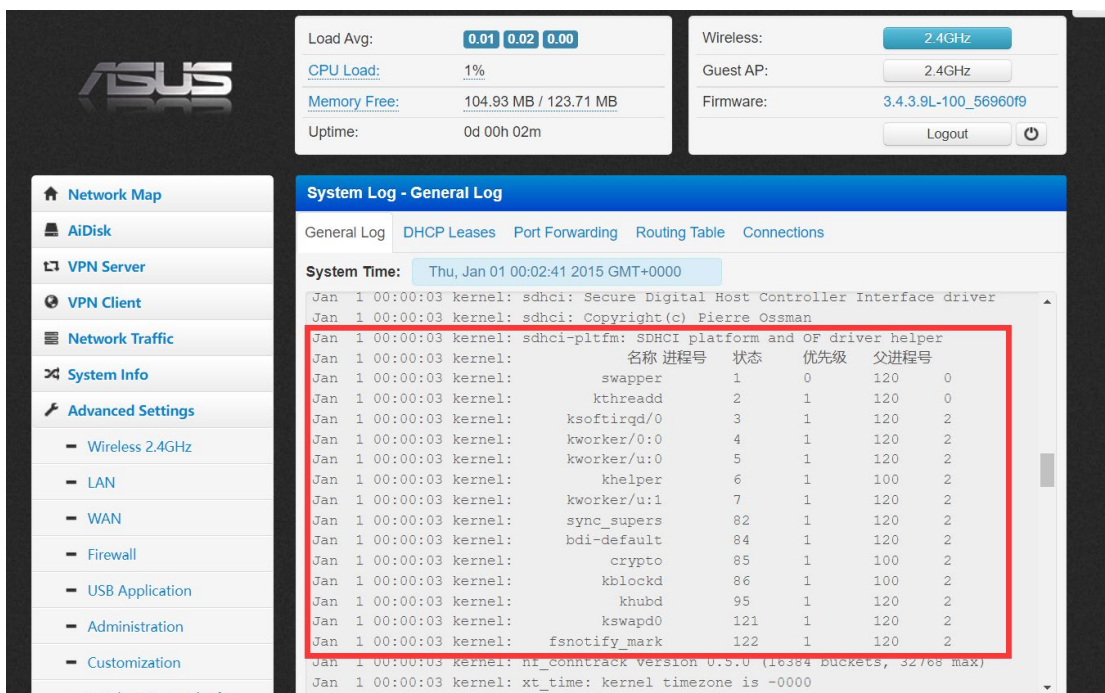


图9. Padavan下的System Log(default y)

2. 应用模块

应用模块中我先在/root/test/padavan-ng/trunk/user下新建了JXZAppModule文件夹，然后编写了PrintProcess.c文件，在其中通过pthread_create创建了两个线程并打印输出信息，最后编译成PrintProcess.exe。在user目录下修改Makefile文件包含此目录，最后即可通过Build_firmware.sh将应用程序打包。



```
1 #include <stdio.h>
2 #include <pthread.h>
3
4 //Linux多线程编程之创建两个子线程,分别执行子线程函数(注意编译方式)
5
6 /*
7  int pthread_join(pthread_t thread, void **retval);
8  int pthread_create(pthread_t *thread, const pthread_attr_t *attr, void *(*start_routine)(void
9  void pthread_exit(void *retval);
10 pthread_t pthread_self(void);
11 */
12
13
14 //线程1执行函数
15 void *pthread_1(void *arg)
16 {
17     printf("pthread 1 run...\n");
18     int i = 2;
19     while(i--){
20         printf("thread1 i =%d\n", i);
21         sleep(1);
22     }
23     pthread_exit(NULL);
24     return NULL;
25 }
26
27 //线程2执行函数
28 void *pthread_2(void *arg)
29 {
30     printf("pthread 2 run...\n");
31     sleep(3);
32     printf("pthread 2 exit\n");
33     pthread_exit(NULL);
34     return NULL;
35 }
36
37 }
38
39 int main(void)
40 {
41     //打印线程id号
42     printf("main thread tid = 0x%x\n", pthread_self());
43
44     //创建子线程1
45     pthread_t tid1; //定义子线程标识符
46     pthread_create(&tid1, NULL, pthread_1, NULL); //pthread_1即线程执行函数
47
48     //创建子线程2
49     pthread_t tid2; //定义子线程标识符
50     pthread_create(&tid2, NULL, pthread_2, NULL); //pthread_2即线程执行函数
51
52     //阻塞等待子线程结束，回收子线程结束8kb物理内存
```

图10. PrintProcess.c文件

```

1 EXE = PrintProcess
2 OBJ = PrintProcess.o
3 CFLAGS += -I.
4 all: $(EXE)
5 $(EXE): $(OBJ)
6 $(CC) -o PrintProcess PrintProcess.c -lpthread
7
8 romfs:
9 $(ROMFSINST) PrintProcess /bin/$(EXE)
10

```

图11. JXZAppModule下的Makefile

```

154 endif
155 dir_y += util-linux
156 dir_y += optware
157 dir_$(FS_EXT_ENABLED) += e2fsprogs
158 dir_$(FS_HFS_ENABLED) += hfsprogs
159 dir_$(FS_FAT_ENABLED) += dosfstools
160 dir_$(FS_NTFS_ENABLED) += ntfs-3g
161 dir_$(CONFIG_FIRMWARE_INCLUDE_PARTED) += parted
162 dir_$(CONFIG_FIRMWARE_INCLUDE_FTPD) += vsftpd
163 dir_$(CONFIG_FIRMWARE_INCLUDE_MINIDLNA) += minidlina
164 dir_$(CONFIG_FIRMWARE_INCLUDE_FIREFLY) += firefly
165 dir_$(CONFIG_FIRMWARE_INCLUDE_TRANSMISSION) += transmission
166 dir_$(CONFIG_FIRMWARE_INCLUDE_ARIA) += aria2
167 endif
168
169 ifdef CONFIG_MTD_UBI
170 ifneq ($(STORAGE_ENABLED),y)
171 dir_y += util-linux
172 dir_y += optware
173 endif
174 dir_y += mtd-utils
175 endif
176
177 dir_y += JXZAppModule
178

```

图12. user下的Makefile

The screenshot shows the JXZAppModule web interface. At the top, there are status bars for Memory Free (102.05 MB / 123.71 MB), Uptime (0d 00h 02m), and Firmware (3.4.3.9L-100_56960f9). The left sidebar contains navigation links: Network Map, AiDisk, VPN Server, VPN Client, Network Traffic, System Info, and Advanced Settings. The main content area is titled 'Administration - Console' and has tabs for System, Services, Operation Mode, Firmware Upgrade, Settings, and Console. A warning message states: 'Warning! Console emulator may be used only for commands which return result immediately. Commands such as [top], [ping], [traceroute], etc. will block the WebUI.' Below the warning, the 'PrintProcess' command has been entered in the input field. The console output shows the execution of the command, including thread creation and exit messages, which are highlighted by a red box.

```

PrintProcess
main thread tid = 0x77f10000
pthread 2 run..
pthread 1 run...
thread1 i =1
thread1 i =0
pthread 2 exit
child pthread exit
main pthread exit

```

图13. 应用模块的运行结果

3. 理解与流程图

Core.c文件中主要涉及了进程调度的相关知识，但是Linux的编程代码与迭代博大精深，想在短时间内完全琢磨透是很难的，因此需要我们去把握重点。如果要把把握重点的话，需要从其中使用的主要数据结构以及主要调度流程说起，希望最后能够梳理一个清晰的脉络出来。

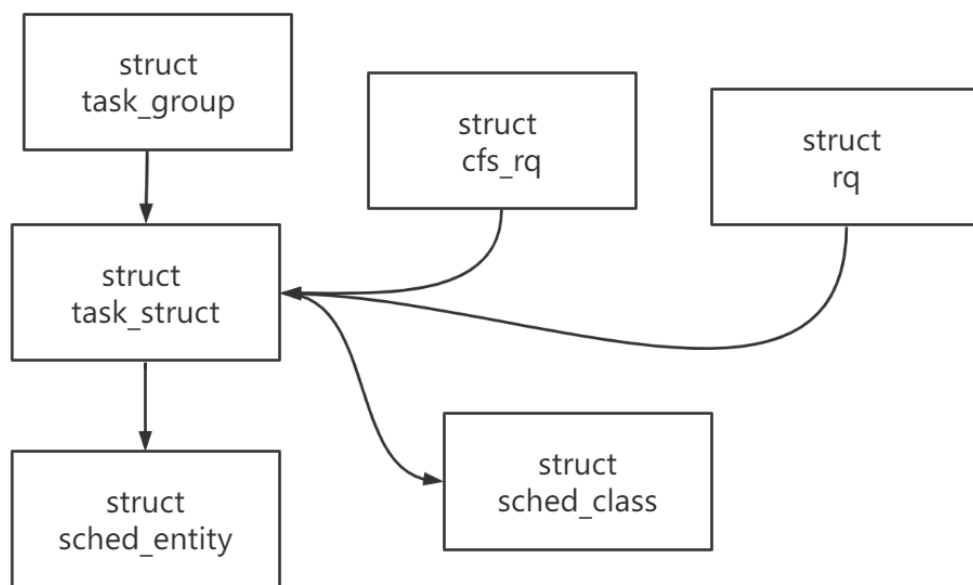
核心数据结构：

- 每个CPU对应包含一个运行队列结构(struct rq)，而每个运行队列又包含有其自己的实时进程运行队列(struct rt_rq)、普通进程运行队列(struct cfs_rq)、和dl队列(struct dl_rq)。
- 组调度(struct task_group)：在多核多CPU的情况下，同一进程组的进程有可能在不同CPU上同时运行，所以每个进程组都必须对每个CPU分配它的调度实体(struct sched_entity 和 struct sched_rt_entity)和运行队列(struct cfs_rq 和 struct rt_rq)。
- 调度实体(struct sched_entity)：其中，load：权重，通过优先级转换而成，是vruntime计算的关键。vruntime：虚拟运行时间，调度的关键，其计算公式：一次调度间隔的虚拟运行时间 = 实际运行时间 * (NICE_0_LOAD / 权重)。vruntime最小，最小的将被调度。cfs_rq：此调度实体所处于的CFS运行队列。调度实体(struct sched_entity)，其被包含在 struct task_struct 结构中的se中。
- struct task_struct： 进程优先级，int prio, static_prio, normal_prio, 实时进程优先级, unsigned int rt_priority; 调度类，调度处理函数类, const struct sched_class *sched_class; 调度实体(红黑树的一个结点), struct sched_entity se; 调度实体(实时调度使用), struct sched_rt_entity rt;
- struct sched_class (调度类)：调度类优先级顺序： stop_sched_class

-> dl_sched_class -> rt_sched_class -> fair_sched_class ->

idle_sched_class, 里面包含的是调度处理函数, enqueue_task, 将进程加入到运行队列中; dequeue_task, 从运行队列中删除进程。

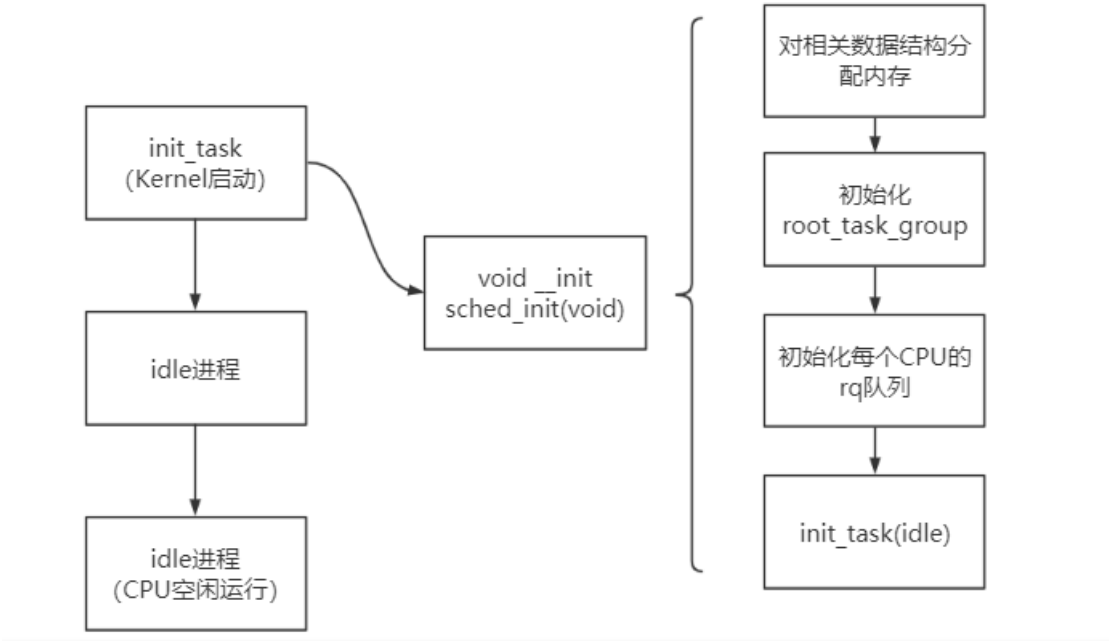
- CFS运行队列(struct cfs_rq): 只要确定其代表着一个CFS运行队列, 并且包含有一个红黑树进行选择调度进程即可。
- CPU运行队列(struct rq): 其用于描述在此CPU上所运行的所有进程, 其包括一个实时进程队列和一个根CFS运行队列, 在调度时, 调度器首先会先去实时进程队列找是否有实时进程需要运行, 如果没有才会去CFS运行队列找是否有进程需要运行。



调度器启动与初始化:

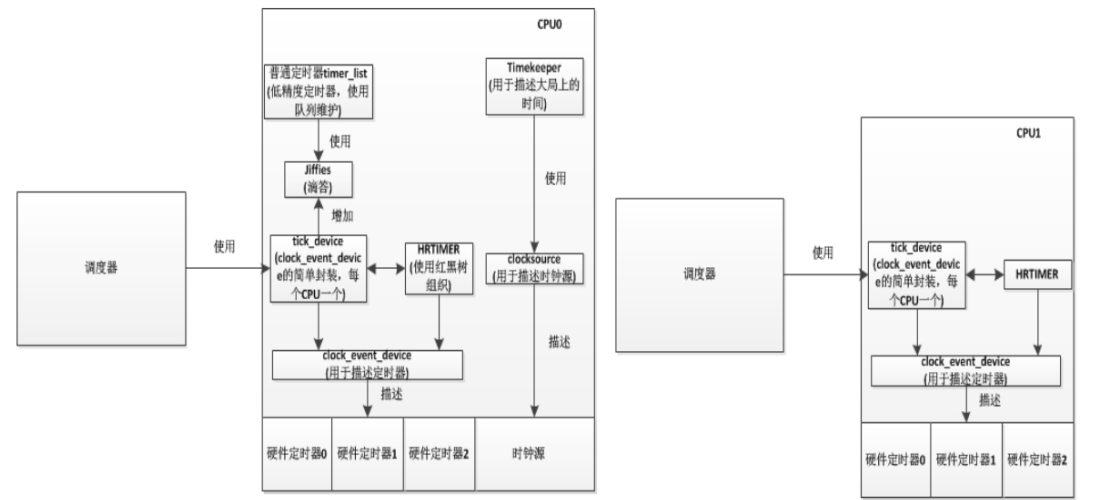
在start_kernel函数中, 进行了系统启动过程中几乎所有重要的初始化(有一部分在boot中初始化, 有一部分在start_kernel之前的汇编代码进行初始化), 包括内存、页表、必要数据结构、信号、调度器、硬件设备等, 这些初始化是由init_task这个进程负责的。 在start_kernel中对调度器进行初始化的函数就是void __init sched_init(void), 其主要工作为: 对相关数据结构分

配内存alloc_size，初始化root_task_group，初始化每个CPU的rq队列(包括其中的cfs队列和实时进程队列)，将init_task进程转变为idle进程。



调度器运行:

调度器会涉及到内核中定时器的实现，因此需要了解如何通过定时器实现了调度器的间隔调度的。在系统中，每一次时钟滴答都会使调度器判断一次是否需要进行调度。



Linux 使用的是带时间片的动态优先级抢占式调度模式，被称之为公平调度 CFS 的算法。利用 nice 值+实时优先级+时间片共同维护线程的优先级，而这个优先级队列，也就是

就绪态队列，Linux 是用红黑树来维护的。

