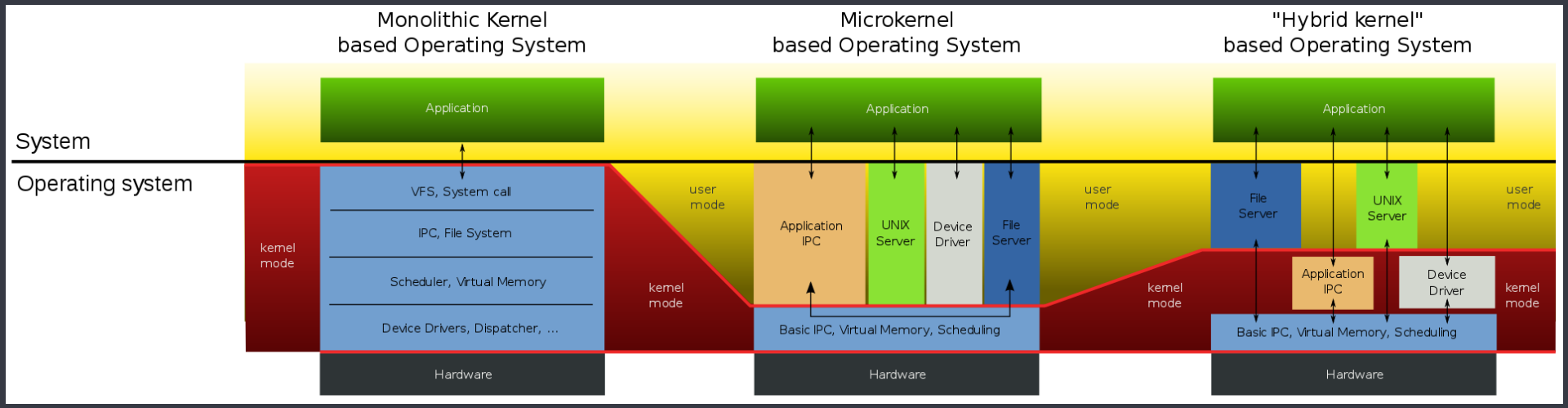
Linux内核编程：模块机制

主讲：王利涛

《嵌入式工程师自我修养》系列视频教程

1 可加载模块的概念

- 宏内核与微内核
  - 程序的运行状态：用户态和内核态
  - 宏内核：内核是个大箩筐，什么都可以往里装
  - 微内核：只保留核心模块，易维护、低性能
  - 混合内核：结合了两者优势，折中平衡



LKM：loadable kernel module
LKM：loadable kernel module
LKM：loadable kernel module

- Linux内核的模块机制
  - LKM： Loadable Kernel Module
  - 内核模块化、高度可定制和裁剪
  - 适配不同的架构、硬件平台
  - 支持运行时动态加载或卸载一个模块
  - 不需要重新编译、重启内核
- 实验
  - 一个内核模块的编译和运行
  - 动态加载、动态卸载

- 本期课程主要内容
  - 掌握Linux内核模块编程方法
  - 内核模块的 Makefile 编写方法
  - 模块引用、模块间的依赖
  - 内核中的头文件如何包含
  - 可加载模块的运行过程分析
  - 理解内核中模块实现机制
  - 学会使用模块机制分析Linux内核
- 实验环境
  - Vmware + Ubuntu + QEMU
  - U-boot + Linux-5.10.4 + NFS

2 内核模块的构成

- 内核模块的构成
  - module_init(hello_init)：　模块加载函数
  - module_exit(hello_exit)：　模块卸载函数
  - MODULE_LICENSE("GPL")：模块许可声明
  - module_param：　模块参数
  - MODULE_PARAM_DESC：　模块参数描述
  - MODULE_AUTHOR：　模块作者
  - MODULE_DESCRIPTION：　模块描述信息
  - EXPORT_SYMBOL：　导出全局符号
  - EXPORT_SYMBOL_GPL：
- 头文件
  - linux/module.h：提供模块相关的接口
  - linux/init.h：初始化、清理相关

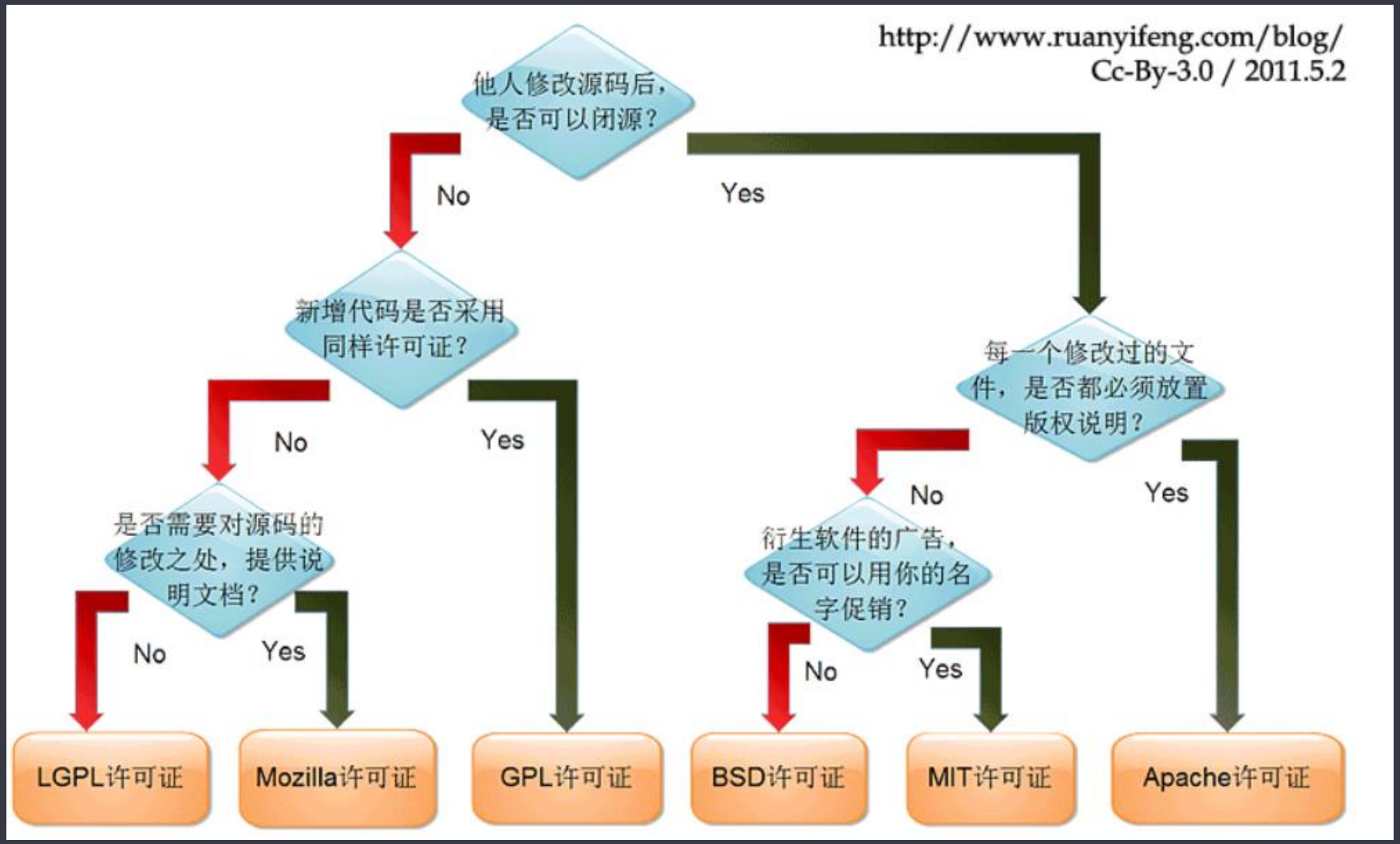
- hello_init
  - 模块加载入口函数，主要完成模块初始化工作
  - 使用__init声明，使用module_init指定
  - 模块被加载到内核时，入口函数自动被内核执行
  - 返回值：errno
  - 应用层可根据返回值，使用 perror 进行解析
- hello_exit
  - 模块卸载函数，模块卸载时该函数自动被内核执行
  - 使用__exit声明，使用module_exit指定
  - 主要完成结束模块运行的相关工作、清理各种资源
  - 返回类型：void

3 内核许可声明

- 模块许可声明
  - 用来描述内核的许可权限：内核以GPL发布
  - 模块不声明LICENSE，内核会有(kernel tainted)警告
  - 内核状态此时是受污染的（dirty）
  - 内核受污染后，一些调试、打印功能可能会失效

```
[root@vexpress ] # insmod hello.ko
[   32.793845] hello: loading out-of-tree module taints kernel.
[   32.794406] hello: module license 'GPL V2' taints kernel.
[   32.794770] Disabling lock debugging due to kernel taint
[   32.800411] Hello world!
```

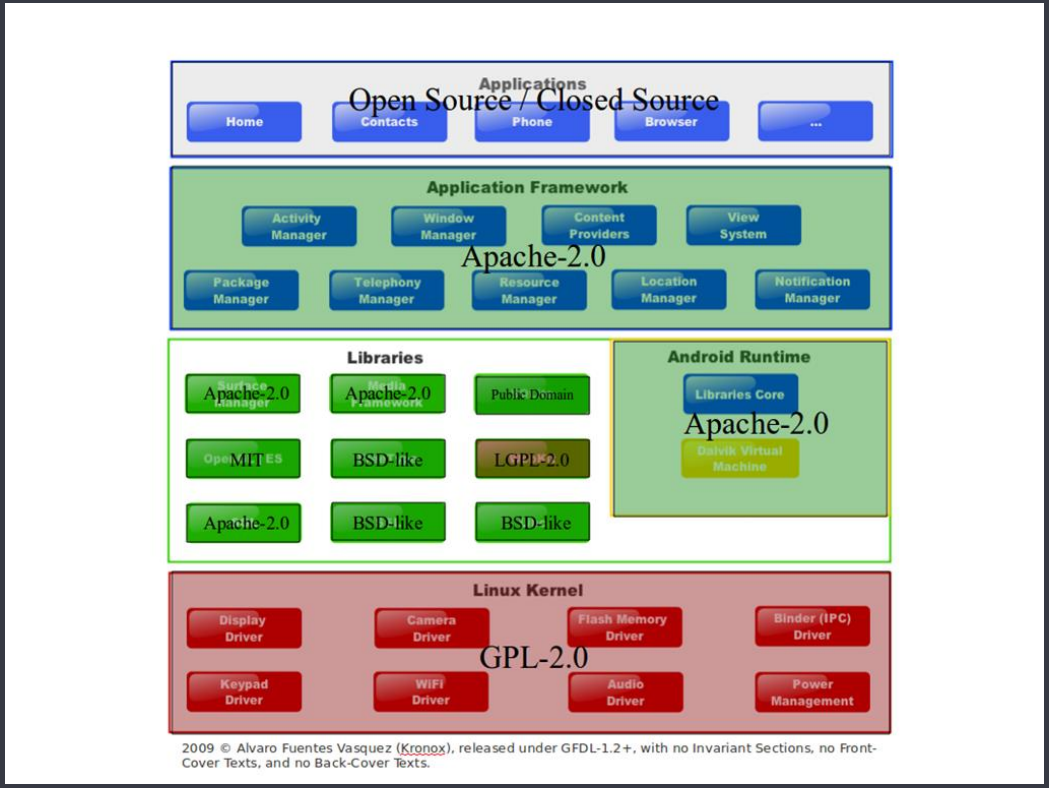
- 协议分类


http://www.ruanyifeng.com/blog/
Cc-By-3.0 / 2011.5.2
他人修改源码后，是否可以闭源？
No
Yes
新增代码是否采用同样许可证？
No
Yes
是否需要对源码的修改之处，提供说明文档？
No
Yes
每一个修改过的文件，是否都必须放置版权说明？
No
Yes
衍生软件的广告，是否可以用你的名字促销？
No
Yes
LGPL许可证
Mozilla许可证
GPL许可证
BSD许可证
MIT许可证
Apache许可证

- 内核可以接受的协议
  - GPL：免费使用，可修改代码，但要开源
  - GPL v2：
  - GPL add additional rights：
  - Dual BSD/GPL：代码修改后不用开源，可销售、但不能借用作者名头宣传
  - Dual MIT/GPL：比BSD更宽泛，只要注明作者版权，其他无限制
  - Dual MPL/GPL：与GPL类似，但允许核心代码以库的形式发布，接口API以MPL协议发布

- 个人的代码如何选择协议？
  - GPL协议
  - Apache协议
  - BSD/MIT协议
- 公司代码如何选择协议？
  - 驱动源码、内核模块
  - 项目代码
  - GPL恐惧症
  - GPL默契：止步于内核空间

- Android与Linux内核之间的关系
  - AOSP：Android内核使用了kernel，采用GPL协议
  - 驱动：放在HAL层或以KO形式动态加载(如GPU)
  - HAL层、 Google应用：采用Apache协议，不开源
  - Bionic libc：采用BSD协议



4 内核污染 (kernel tainted)

- 内核被污染的原因
  - 加载一些不开源、跟GPL不兼容的驱动
  - staging内核模块
  - 一些out-of-tree模块的加载运行
  - 内核编译时依赖的内核版本和运行版本不一致
  - 支持SMP的内核在早期不支持多核的单核CPU上运行
  - BIOS或UEFI中的一些bug
  - 内核启动时的一些机器检查或OOPS
- 内核被污染后
  - 一些调试功能、输出失效，一些API系统调用失效
  - 社区一般不会处理tainted kernel下的bug

- 查看内核被污染的原因

```
# cat /proc/sys/kernel/tainted
0
# cat /proc/sys/kernel/tainted
4096
```

- 内核说明文档

```
===  ===  ======  ========================================================
Bit   Log  Number  Reason that got the kernel tainted
===  ===  ======  ========================================================
 0   G/P       1   proprietary module was loaded
 1   _/F       2   module was force loaded
 2   _/S       4   SMP kernel oops on an officially SMP incapable processor
 3   _/R       8   module was force unloaded
 4   _/M      16   processor reported a Machine Check Exception (MCE)
 5   _/B      32   bad page referenced or some unexpected page flags
 6   _/U      64   taint requested by userspace application
 7   _/D     128   kernel died recently, i.e. there was an OOPS or BUG
 8   _/A     256   ACPI table overridden by user
 9   _/W     512   kernel issued warning
10   _/C    1024   staging driver was loaded
11   _/I    2048   workaround for bug in platform firmware applied
12   _/O    4096   externally-built ("out-of-tree") module was loaded
13   _/E    8192   unsigned module was loaded
14   _/L   16384   soft lockup occurred
15   _/K   32768   kernel has been live patched
16   _/X   65536   auxiliary taint, defined for and used by distros
17   _/T  131072   kernel was built with the struct randomization plugin
===  ===  ======  ========================================================
```

5 模块签名机制

- 内核模块签名机制
  - CONFIG_MODULE_SIG
  - CONFIG_MODULE_SIG_FORCE
  - CONFIG_MODULE_SIG_ALL
  - 启动选项：module.sig_enforce=1
- 模块签名内核编译配置

```
--- Enable loadable module support
[*]   Module signature verification
[*]     Require modules to be validly signed
[*]     Automatically sign all modules
      Which hash algorithm should modules be signed with? (Sign modules with
SHA-1) --->

# make
# make modules_install
# certs/signing_key.x509、 certs/ signing_key.pem
```

- 手工给模块签名

```
# strip  --strip-debug  hello.ko
# hexdump  –C  hello.ko | tail
00015080  00 00 00 00 00 00 00 00  01 00 00 00 00 00 00 00  |................|
00015090  01 00 00 00 02 00 00 00  00 00 00 00 00 00 00 00  |................|
000150a0  a4 00 01 00 50 03 00 00  26 00 00 00 30 00 00 00  |....P...&...0...|
000150b0  04 00 00 00 10 00 00 00  09 00 00 00 03 00 00 00  |................|
000150c0  00 00 00 00 00 00 00 00  f4 03 01 00 e4 00 00 00  |................|
000150d0  00 00 00 00 00 00 00 00  01 00 00 00 00 00 00 00  |................|
000150e0  11 00 00 00 03 00 00 00  00 00 00 00 00 00 00 00  |................|
000150f0  30 49 01 00 97 01 00 00  00 00 00 00 00 00 00 00  |0I..............|
00015100  01 00 00 00 00 00 00 00                           |........|
00015108
# scripts/sign-file  sha1  signing_key.pem  signing_key.x509  hello.ko
# hexdump  –C  hello.ko | tail
00015350  bb ce 65 71 93 d1 33 af  51 0e 91 cc 14 8a 6f e1  |..eq..3.Q.....o.|
00015360  0f 5d fd d6 5b 9f 62 9f  cb ce 72 02 45 36 a2 cf  |.]..[.b...r.E6..|
00015370  f8 c7 27 08 c7 c8 58 f6  7f ff fe ff 14 26 de 25  |..'...X......&.%|
00015380  6c 0b d8 e2 56 8e b5 d5  33 55 9f 84 9d 66 38 44  |l...V...3U...f8D|
00015390  9c c6 5c 83 e5 a7 76 65  97 c4 65 6c a7 50 ac bd  |..\...ve..el.P..|
000153a0  11 97 41 6b 05 4e 46 30  4a 00 00 02 00 00 00 00  |..Ak.NF0J.......|
000153b0  00 00 00 02 a1 7e 4d 6f  64 75 6c 65 20 73 69 67  |.....~Module sig|
000153c0  6e 61 74 75 72 65 20 61  70 70 65 6e 64 65 64 7e  |nature appended~|
000153d0  0a                                               |.|
000153d1
```

6 将模块编译进内核

- 修改Kconfig和Makefile

```
obj-$(CONFIG_HELLO)    += hello.o

menu "Character devices"
config HELLO
                bool  "A simplist kernel module: hello"
                help
                a kernel module demo: hello world
```

- 编译内核并重启

```
# make  ARCH=arm  CROSS_COMPILE=arm-linux-gnueabi-  uImage  LOADADDR=0x6000300
# dmesg | grep world
  Hello world!
```

- 模块化编译

```
menu "Character devices"
config HELLO
        tristate "A simplist kernel module: hello"
        help
        a kernel module demo: hello world


# make  ARCH=arm  CROSS_COMPILE=arm-linux-gnueabi-  uImage
LOADADDR=0x60003000

# make ARCH=arm CROSS_COMPILE=arm-linux-gnueabi- modules
   CALL    scripts/checksyscalls.sh
   CALL    scripts/atomic/check-atomics.sh
   LD [M]  drivers/char/hello.ko
# cp drivers/char/hello.ko  /home/nfs
# insmod hello.ko
```

7 模块的out-of-tree编译

- Makefile解析

```
.PHONY: all clean

obj-m := hello.o

EXTRA_CFLAGS += -DDEBUG
KDIR := /home/linux-5.10.4
ARCH_ARGS := CROSS_COMPILE=arm-linux-gnueabi-   ARCH=arm

all:
        make $(ARCH_ARGS) -C $(KDIR) M=$(PWD) modules
clean:
        make $(ARCH_ARGS) -C $(KDIR) M=$(PWD) modules clean
```

- 源码内模块编译过程

```
# make CROSS_COMPILE=arm-linux-gnueabi- ARCH=arm -C /home/linux-5.10.4
M=/home/workplace/07 modules
make[1]: Entering directory '/home/linux-5.10.4'
  CC [M]  /home/workplace/07/hello.o
  MODPOST /home/workplace/07/Module.symvers
  CC [M]  /home/workplace/07/hello.mod.o
  LD [M]  /home/workplace/07/hello.ko
make[1]: Leaving directory '/home/linux-5.10.4'
```

- 源码外模块编译过程

```
# make –trace
make CROSS_COMPILE=arm-linux-gnueabi- ARCH=arm -C /home/linux-5.10.4
M=/home/workplace/07 modules
make[1]: Entering directory '/home/linux-5.10.4'
CC [M]  /home/workplace/07/hello.o
scripts/Makefile.build:292: update target '/home/workplace/07/hello.mod'
scripts/Makefile.build:418: update target
'/home/workplace/07/modules.order'
MODPOST /home/workplace/07/Module.symvers
CC [M]  /home/workplace/07/hello.mod.o
LD [M]  /home/workplace/07/hello.ko
make[1]: Leaving directory '/home/linux-5.10.4'
```

8 模块的Makefile分析

- Makefile与Kbuild的关系

```
obj-y := hello.o
obj-m := hello.o
obj-{CONFIG_HELLO} := hello.o
hello-objs := hello.c sub.c


obj-m := hello.o
KDIR := /home/linux-5.10.4
ARCH_ARGS := CROSS_COMPILE=arm-linux-gnueabi- ARCH=arm

all:
	make $(ARCH_ARGS) -C $(KDIR) M=$(PWD) modules
clean:
	make $(ARCH_ARGS) -C $(KDIR) M=$(PWD) modules clean
```

- 将Makefile和Kbuild分开

```
//Kbuild：
obj-m := hello.o

//Makefile：
KDIR := /home/linux-5.10.4
ARCH_ARGS := CROSS_COMPILE=arm-linux-gnueabi- ARCH=arm

all:
	make $(ARCH_ARGS) -C $(KDIR) M=$(PWD) modules
clean:
	make $(ARCH_ARGS) -C $(KDIR) M=$(PWD) modules clean
```

- out-of-tree模块的Makefile，为什么这么写？

```
#ifneq ($(KERNELRELEASE),)
obj-m := hello.o
#else

KDIR := /home/linux-5.10.4
ARCH_ARGS := CROSS_COMPILE=arm-linux-gnueabi- ARCH=arm
all:
	make $(ARCH_ARGS) -C $(KDIR) M=$(PWD) modules
clean:
	make $(ARCH_ARGS) -C $(KDIR) M=$(PWD) modules clean
#endif
```

- Makefile的另一种写法：

```
#ifneq ($(KERNELRELEASE),)
include kbuild
#else

KDIR := /home/linux-5.10-rc3
ARCH_ARGS := CROSS_COMPILE=arm-linux-gnueabi- ARCH=arm
all:
	make $(ARCH_ARGS) -C $(KDIR) M=$(PWD) modules
clean:
	make $(ARCH_ARGS) -C $(KDIR) M=$(PWD) modules clean
#endif
```

对应的Kbuild文件

```
obj-m := hello.o
```

9 模块参数

- 如何给模块传参数？

```
#define module_param(name, type, perm) \
module_param_named(name, name, type, perm)
```

  - name：要传递的参数，对应模块中的全局变量
  - type：要传递的参数类型，要和全局变量类型一致
  - perm：读写权限
    - /sys/module/hello/parameters/xx 参数节点
    - 0666：读写权限
    - 0444：只读权限，无法对这个文件节点执行写的操作
    - 4-读，2-写，1-执行

10 通过U-boot给模块传参

- 通过U-boot给模块传参

```
tftp 0x60003000 uImage;tftp 0x60500000 vexpress-v2p-ca9.dtb;
setenv bootargs 'root=/dev/nfs rw
nfsroot=192.168.33.145:/home/nfs,proto=tcp,
nfsvers=4,nolock init=/linuxrc ip=192.168.33.144 console=ttyAMA0
hello.num=100';bootm 0x60003000 - 0x60500000;
```

重新编译U-boot并重启

```
# dmesg | grep num
# Hello world, param = 100
```

11 EXPORT_SYMBOL

- 用户空间的模块化编程
  - 函数的实现：math.c/int add(int a, int b)
  - 函数的声明：math.h/int add(int a, int b);

```
#include <stdio.h>
#include "math.h"
int main(void)
{
    int sum = 0;
    sum = add(3, 4);
    printf("sum = %d\n", sum);
    return 0;
}
```

- 内核空间的模块化编程
  - 模块的封装：static、EXPORT_SYMBOL
  - 函数的声明：头文件

```
#include <linux/init.h>
#include <linux/module.h>
static int num;
int add(int a, int b)
{
   return a + b;
}
EXPORT_SYMBOL(add);

static int __init math_init(void)
{
   printk("hello math_moudle\n");
   return 0;
}

module_init(math_init);
```

- 按不同的协议导出符号
  - EXPORT_SYMBOL
  - EXPORT_SYMBOL_GPL

12 模块的版本控制

- 模块的版本控制
  - 解决内核模块和内核之间的接口一致性问题
  - 根据函数参数、返回值类型等生成CRC校验码
  - 当内核和模块双方的校验码相等，则为相同接口
  - 内核启动版本控制功能：CONFIG_MODVERSIONS

```
-*- Enable loadable module support
[ ]   Forced module loading
[*]   Module unloading
[ ]     Forced module unloading
[*]   Module versioning support
[ ]   Source checksum for all modules
[ ]   Module signature verification
[ ]   Compress modules on installation
[ ]   Allow loading of modules with missing namespace imports
[ ]   Enable unused/obsolete exported symbols
[ ]     Trim unused exported kernel symbols
```

- 相关的几个文件
  - hello.mod.c
  - hello.ko： __versions section
  - 内核：Modules.symvers
  - 模块：Modules.symvers

13 模块的头文件

- 模块的头文件

```
#include <linux/xx.h>
#include <asm/xx.h>
#include <plat/xx.h>
#include <mach/xx.h>
#include "usb.h"
```

- 头文件分类
  - 内核专用头文件：include/linux
  - 和CPU架构相关：arch/$(ARCH)/include
  - 板级硬件相关：
    - arch/$(ARCH)/plat-xx/include
    - arch/$(ARCH)/mach-xx/include

- 通过GCC –I指定头文件路径

```
#include <module1/module1.h>
#include <module2/module2.h>
#include <module3/module3.h>

├── a.out
├── inc
│   ├── module1
│   │   └── module1.h
│   ├── module2
│   │   └── module2.h
│   └── module3
│       └── module3.h
├── main.c
├── module1
│   └── module1.c
├── module2
│   └── module2.c
└── module3
    └── module3.c
```

- 内核中的头文件路径

```
LINUXINCLUDE    := \
                   -I$(srctree)/arch/$(SRCARCH)/include \
                   -I$(objtree)/arch/$(SRCARCH)/include/generated \
                   $(if $(building_out_of_srctree),-I$(srctree)/include) \
                   -I$(objtree)/include \
                   $(USERINCLUDE)
373行：
SRCARCH             := $(ARCH)

root@ubuntu:/home/linux-5.10.4/arch/arm/include# tree -L 1
.
├── asm
├── debug
├── generated
└── uapi
```

14 多文件构成的模块

- 编程实验
  - 一个复杂模块往往由多个C文件构成
  - 模块内部接口的封装和引用
  - 模块如何封装
  - 模块间如何引用
  - 头文件
  - Makefile 的写法

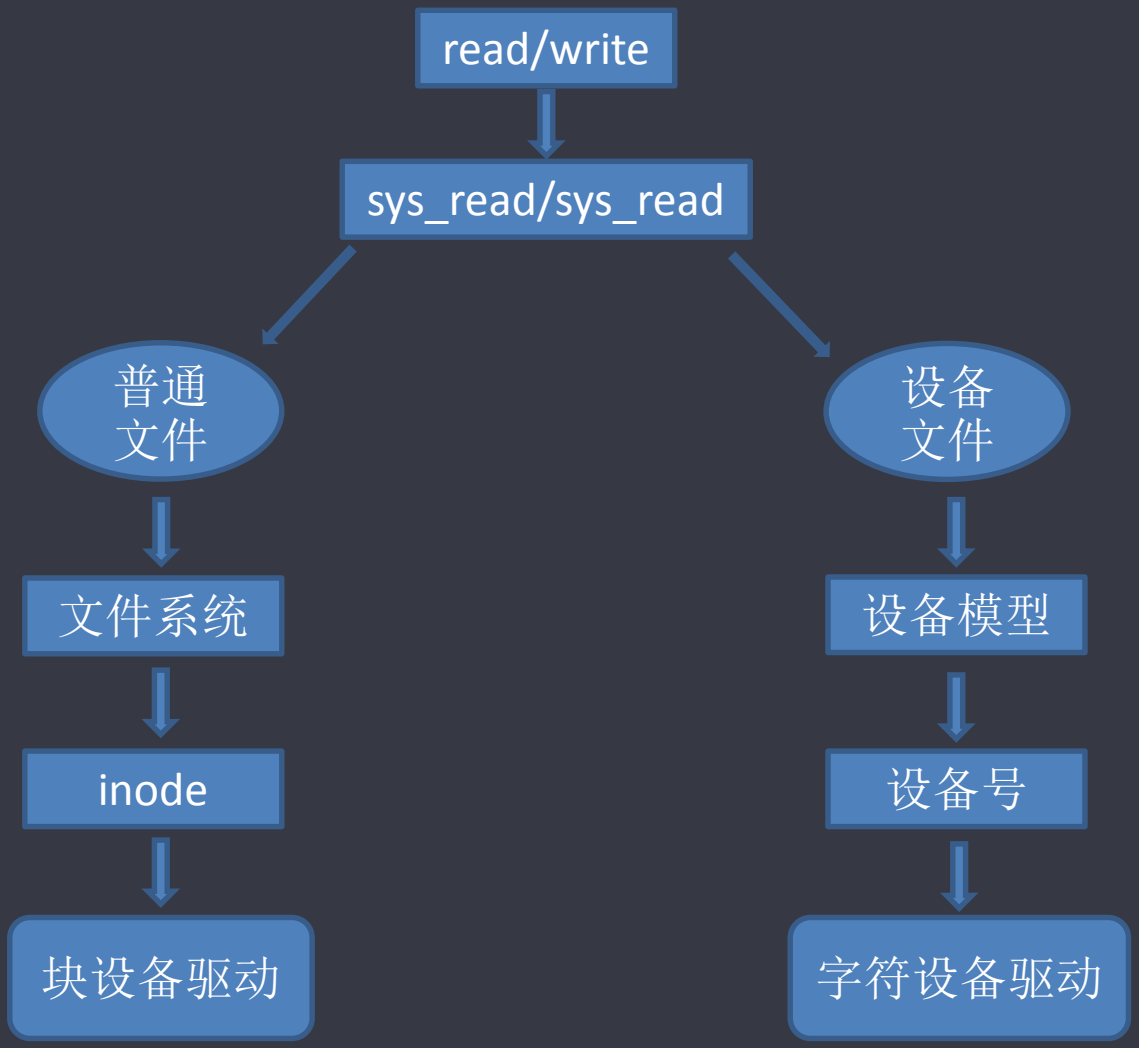
15 模块间的依赖

- 生成模块间的依赖关系
  - # depmod –a
  - 解析/lib/modules/$(kernel_version)下的所有内核模块，通过各个模块EXPORT_SYMBOL和引用的符号，生成一个模块依赖关系表
  - /lib/modules/$(kernel_version)/modules.dep
  - # modprobe hello
  - # modprobe –r hello

16 编写一个字符驱动

- 内核模块的作用
  - 操作系统基础服务
  - 实现一些功能和结构，供内核其他模块使用
  - 各种各样的硬件驱动
  - 实现大量的系统调用接口，供应用程序使用
  - 调用接口：read、write、open、close
- 文件读写过程分析
  - 应用程序、C库函数接口：fread、fwrite
  - 系统调用：sys_read、sys_write函数
  - 内核中的read、write函数

• 文件的读写流程



- 内核编程实验
  - 实现一个最简单的字符驱动
  - 驱动源码：实现基本的read、write接口
  - Makefile编写
  - 创建设备节点
  - 编写应用程序读写设备，看驱动工作是否正常

17 模块的运行过程

- 模块的分类
  - 可加载模块：源码外编译，动态加载、动态卸载
  - 内置模块：直接编译进内核，随内核启动初始化
- 使用dump_stack打印函数调用栈

```
#include <linux/init.h>
#include <linux/module.h>
static int __init hello_init(void)
{
   printk(KERN_INFO"Hello world\n");
   dump_stack();
   return 0;
}
static void __exit hello_exit(void)
{
   printk("Goodbye world\n");
}
module_init(hello_init);
module_exit(hello_exit);
```

- 可加载模块的运行过程

```
Hello world
CPU: 0 PID: 90 Comm: insmod Tainted: G          O      5.10.0-rc3+ #10
Hardware name: ARM-Versatile Express
 [<8010f4bc>] (unwind_backtrace) from [<8010b3a8>] (show_stack+0x10/0x14)
 [<8010b3a8>] (show_stack) from [<808594c4>] (dump_stack+0x98/0xac)
 [<808594c4>] (dump_stack) from [<7f005014>] (hello_init+0x14/0x1000 [hello])
 [<7f005014>] (hello_init [hello]) from [<80101f80>] (do_one_initcall)
 [<80101f80>] (do_one_initcall) from [<801aab5c>] (do_init_module+0x60/0x228)
 [<801aab5c>] (do_init_module) from [<801ace94>] (load_module+0x2070/0x2484)
 [<801ace94>] (load_module) from [<801ad3ec>] (sys_init_module+0x144/0x184)
 [<801ad3ec>] (sys_init_module) from [<80100060>] (ret_fast_syscall+0x0/0x54)
Exception stack(0x81b53fa8 to 0x81b53ff0)
3fa0:                   00000000 000151b4 002154d0 000151b4 001fdfd0 00000000
3fc0: 00000000 000151b4 00000000 00000080 7e9cce48 7e9cce4c 001fdfd0
001e967c
3fe0: 7e9ccb18 7e9ccb08 000367d0 00011350
```

18 模块机制实现分析(上)

- 分析之前的准备
  - C语言与链接脚本、Makefile、Kbuild的交互
  - C语言如何引用链接脚本中定义的符号
  - C语言如何使用Makefile中定义的符号
  - C语言如何使用kbuild配置变量

- C语言使用链接脚本中定义的符号

init/main.c：

```
extern initcall_entry_t __initcall_start[];
extern initcall_entry_t __initcall0_start[];
extern initcall_entry_t __initcall1_start[];
extern initcall_entry_t __initcall2_start[];
extern initcall_entry_t __initcall3_start[];
extern initcall_entry_t __initcall4_start[];
extern initcall_entry_t __initcall5_start[];
extern initcall_entry_t __initcall6_start[];
extern initcall_entry_t __initcall7_start[];
extern initcall_entry_t __initcall_end[];

static initcall_entry_t *initcall_levels[] __initdata = {
                __initcall0_start,
                __initcall1_start,
                __initcall2_start,
                __initcall3_start,
                __initcall4_start,
                __initcall5_start,
                __initcall6_start,
                __initcall7_start,
                __initcall_end,
};
```

内置模块的初始化分析

```
#define module_init(x)          __initcall(x);          // linux/module.h
#define module_exit(x)          __exitcall(x);
#define __initcall(fn)          device_initcall(fn)     // linux/init.h

#define pure_initcall(fn)               __define_initcall(fn, 0)
#define core_initcall(fn)               __define_initcall(fn, 1)
#define arch_initcall(fn)               __define_initcall(fn, 3)
#define arch_initcall_sync(fn)          __define_initcall(fn, 3s)
#define rootfs_initcall(fn)             __define_initcall(fn, rootfs)
#define device_initcall(fn)             __define_initcall(fn, 6)
#define device_initcall_sync(fn)        __define_initcall(fn, 6s)
#define late_initcall(fn)               __define_initcall(fn, 7)
#define late_initcall_sync(fn)          __define_initcall(fn, 7s)

#define __define_initcall(fn, id) ___define_initcall(fn, id, .initcall##id)
#define ___define_initcall(fn, id, __sec)                   \
__ADDRESSABLE(fn)                                           \
asm(".section       \"" #__sec ".init\", \"a\"      \n"     \
"__initcall_" #fn #id ":                        \n"         \
".long      " #fn " - .                         \n"         \
".previous                                      \n");
```

- module_init()宏展开示例

```
module_init(hello_init)
__initcall(hello_init)
device_initcall(hello_init)
__define_initcall(hello_init, 6)
___define_initcall(hello_init, 6, .initcall6)

        .section  ".initcall6.init","a"
        __initcall_hello_init6:
        .long hello_init - .
        .previous
```

19 模块机制实现分析(下)

- # 内核模块的初始化流程

init/main.c
start_kernel - arch_call_rest_init - rest_init
- kernel_init
- kernel_init_freeable
- do_basic_setup
- do_initcalls
- do_initcall_level
- do_one_initcall

```
extern initcall_entry_t __initcall_start[];
extern initcall_entry_t __initcall0_start[];
extern initcall_entry_t __initcall1_start[];
extern initcall_entry_t __initcall2_start[];
extern initcall_entry_t __initcall3_start[];
extern initcall_entry_t __initcall4_start[];
extern initcall_entry_t __initcall5_start[];
extern initcall_entry_t __initcall6_start[];
extern initcall_entry_t __initcall7_start[];
extern initcall_entry_t __initcall_end[];
static initcall_entry_t *initcall_levels[] __initdata = {
                __initcall0_start,
                __initcall1_start,
                __initcall2_start,
                __initcall3_start,
                __initcall4_start,
                __initcall5_start,
                __initcall6_start,
                __initcall7_start,
                __initcall_end,
};
```

• 一些关键的信息

arch/arm/kernel/vmlinux.lds：

```
__initcall_start = .; KEEP(*(.initcallearly.init))
__initcall0_start = .; KEEP(*(.initcall0.init)) KEEP(*(.initcall0s.init))
__initcall1_start = .; KEEP(*(.initcall1.init)) KEEP(*(.initcall1s.init))
__initcall2_start = .; KEEP(*(.initcall2.init)) KEEP(*(.initcall2s.init))
__initcall3_start = .; KEEP(*(.initcall3.init)) KEEP(*(.initcall3s.init))
__initcall4_start = .; KEEP(*(.initcall4.init)) KEEP(*(.initcall4s.init))
__initcall5_start = .; do_one_initcallKEEP(*(.initcall5.init)) KEEP(*(.initcall5s.init))
__initcallrootfs_start = .; KEEP(*(.initcallrootfs.init)) KEEP(*(.initcallrootfss.init))
__initcall6_start = .; KEEP(*(.initcall6.init))
```

module_init(hello_init)展开后：

```
module_init(hello_init)
__initcall(hello_init)
device_initcall(hello_init)
__define_initcall(hello_init, 6)
___define_initcall(hello_init, 6, .initcall6)

        .section  ".initcall6.init","a"
        __initcall_hello_init6:
        .long hello_init - .
        .previous
```

20 使用模块机制分析内核

- 内核中的模块

宏名	调用次数	优先级
pure_initcall	9	0
core_initcall	206	1
postcore_initcall	143	2
arch_initcall	474	3
subsys_initcall	689	4
fs_initcall	122	5
rootfs_initcall	8	rootfs
device_initcall	389	6
late_initcall	246	7
module_init	3158	Loadable or 6

宅学部落

专注嵌入式、Linux精品教程

更多信息

王利涛老师个人店：https://wanglitao.taobao.com/
嵌入式在线教程网：www.zhaixue.cc
嵌入式技术交流群：

宅学部落02群：398294860
宅学部落03群：559671596
宅学部落04群：528718820

欢迎关注公众号：