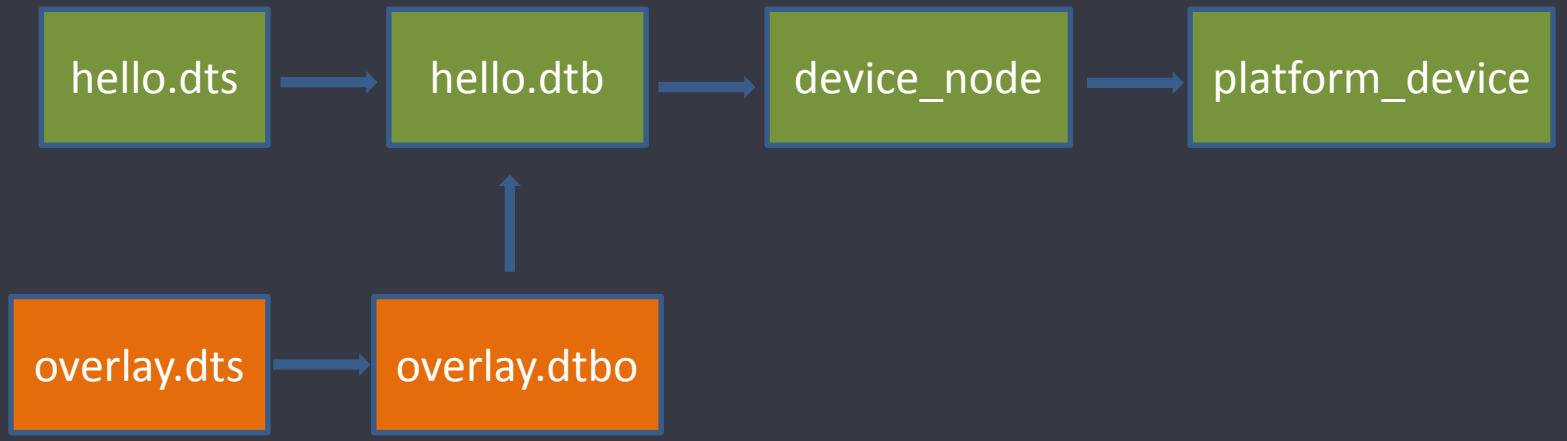
Linux内核编程12期
设备树的overlay与ConfigFS

主讲：王利涛

01 什么是设备树的overlay？

主讲：王利涛

- 设备树的overlay功能
  - 在系统runtime期间修改设备树
  - 设备树的编译、加载和运行



- 需求与现状
  - 外界插拔设备，无法在设备树中预先描述：耳机
  - 树莓派 + FPGA开发板
  - 基于I2C的温度传感器
  - 管脚的重新配置：PIN multiplexing
  - 修改bootcmd、分区
  - 内核mainline还没支持，不同平台各自实现

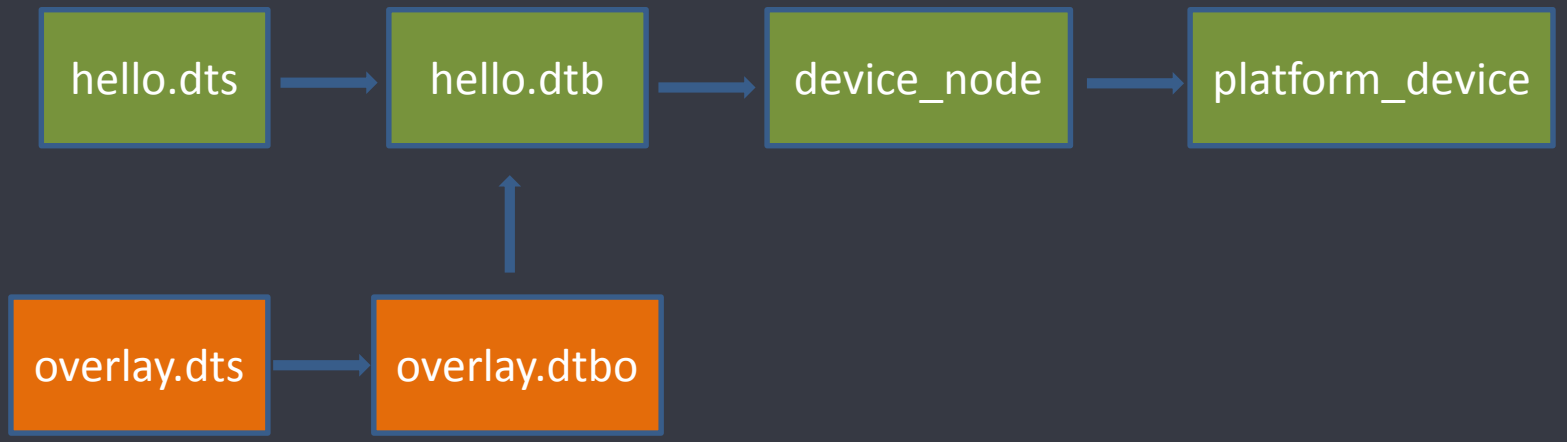
- 本期课程的主要内容
  - 在开发板上如何实现设备树的overlay功能
  - Configfs文件系统的配置与挂载
  - Configfs编程接口
  - 如何编写设备树 overlay文件
  - 设备树 overlay的编译和运行
  - 设备树overlay运行机制分析

适合哪些人学习：

嵌入式驱动工程师
嵌入式BSP工程师
嵌入式软件工程师
想从事嵌入式软件开发的同学

02 设备树overlay实现原理分析

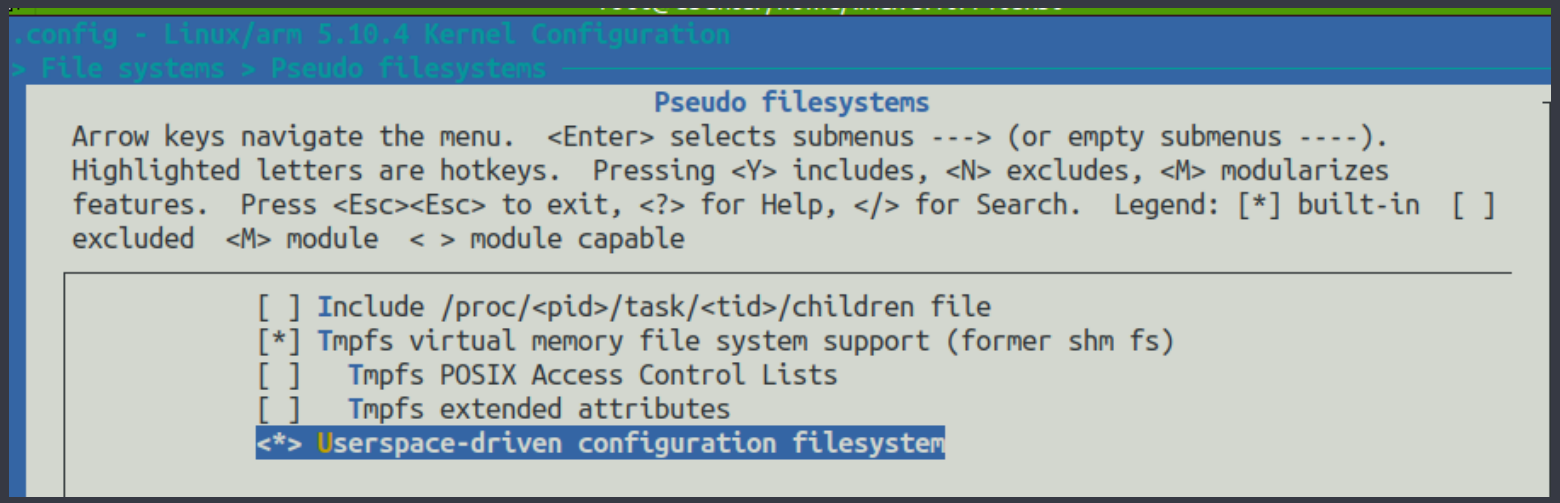
- 如何动态修改设备树？
  - 设备树的解析过程
  - 几个重要的函数
  - 如何将dtb文件加载到内核
  - 文件系统接口：proc、sysfs、configfs
  - changset：增改删



03 ConfigFS的编译与挂载

- ConfigFS文件系统简介
  - 基于RAM的内核对象管理器：config_item
  - 内核对象在用户空间的接口，类似proc/sysfs
  - 亮点：用户可以在用户空间创建内核对象
  - 源码：linux-5.10.4/fs/configfs

- 内核编译配置与挂载

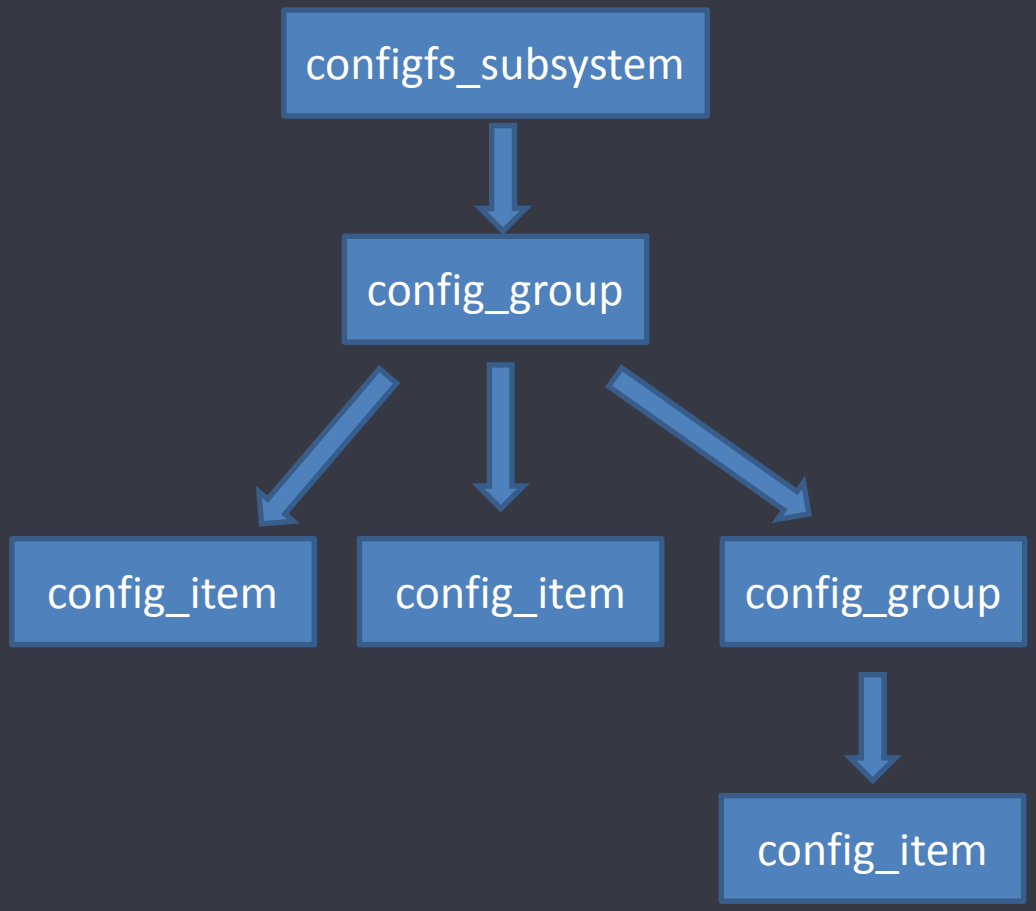


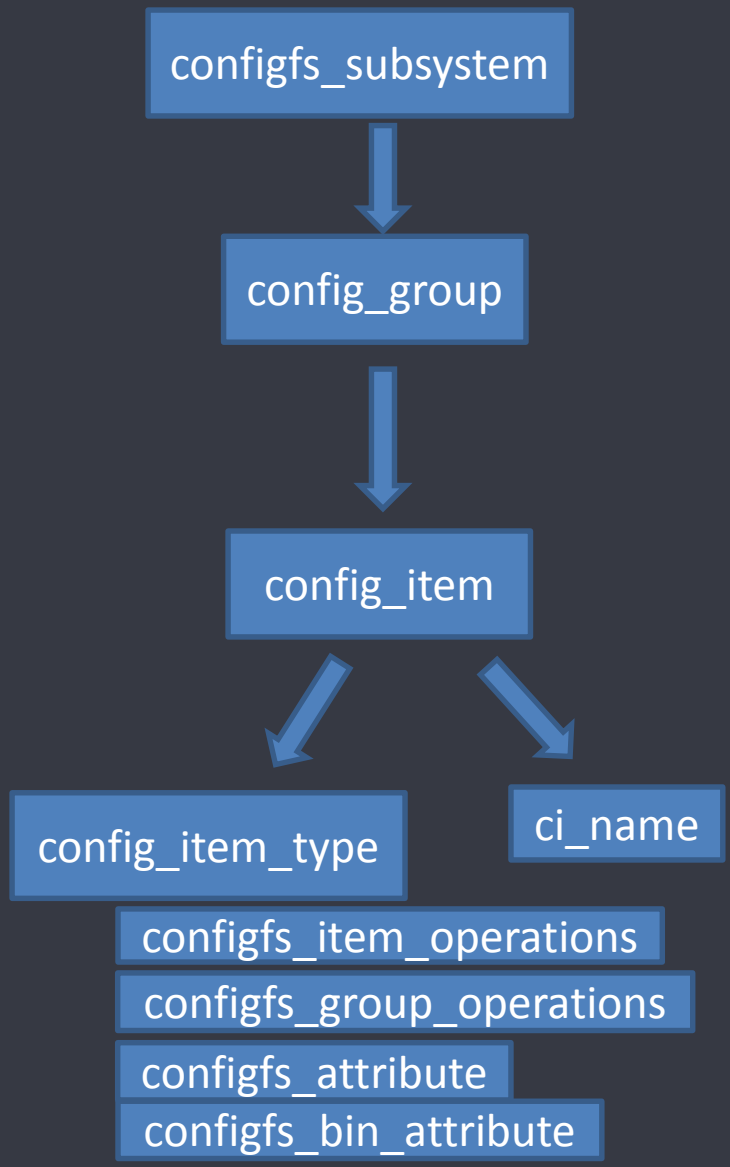

```
# mount -t configfs none /sys/kernel/config
```

```
1  #device     mount-point type    options     dump    fsck order
2  proc        /proc       proc    defaults       0    0
3  tmpfs       /tmp        tmpfs   defaults       0    0
4  sysfs       /sys        sysfs   defaults       0    0
5  tmpfs       /dev        tmpfs   defaults       0    0
6  var         /dev        tmpfs   defaults       0    0
7  ramfs       /dev        ramfs   defaults       0    0
8  debugfs     /sys/kernel/debug   debugfs     defaults     0    0
9  configfs    /sys/kernel/config  configfs    defaults     0    0
10 tmpfs       /log        tmpfs   defaults       0    0
```

04 ConfigFS的核心数据结构

- 几个关键的核心数据结构





- 子系统、容器和config_item

```
struct configfs_subsystem {
        struct config_group  su_group;
        struct mutex         su_mutex;
};
struct config_group {
        struct config_item     cg_item;
        struct list_head       cg_children;
        struct configfs_subsystem   *cg_subsys;
        struct config_group       **default_groups;
         struct list_head          group_entry;
 };
 struct config_item {
    char                    *ci_name;
    char                     ci_namebuf[CONFIGFS_ITEM_NAME_LEN];
    struct kref                    ci_kref;
    struct list_head               ci_entry;
    struct config_item             *ci_parent;
    struct config_group            *ci_group;
    struct config_item_type        *ci_type;
    struct dentry                  *ci_dentry;
 };
```

- 属性和方法

```
 struct config_item_type {
        struct module                   *ct_owner;
        struct configfs_item_operations *ct_item_ops;
        struct configfs_group_operations *ct_group_ops;
        struct configfs_attribute       **ct_attrs;
        struct configfs_bin_attribute   **ct_bin_attrs;
};

struct configfs_attribute {
        const char      *ca_name;       // 属性名字
        struct module   *ca_owner;      // 属性所属模块
        umode_t         ca_mode;        // 访问权限
        ssize_t (*show)(struct config_item *, char *);      // 属性的读写方法
        ssize_t (*store)(struct config_item *, const char *, size_t);
};
```

05 通过ConfigFS加载二进制文件

- 二进制属性和方法

```
 struct config_item_type {
        struct module                   *ct_owner;
        struct configfs_item_operations     *ct_item_ops;
        struct configfs_group_operations    *ct_group_ops;
        struct configfs_attribute           **ct_attrs;
        struct configfs_bin_attribute       **ct_bin_attrs;
};

struct configfs_bin_attribute {
        struct configfs_attribute cb_attr;          /* std. attribute */
        void *cb_private;                           /* for user       */
        size_t cb_max_size;                         /* max core size  */
        ssize_t (*read)(struct config_item *, void *, size_t);
        ssize_t (*write)(struct config_item *, const void *, size_t);
};
```

06 创建ConfigFS子目录

- 创建目录：make_item

```
struct config_item {
        char                            *ci_name;
        char                            ci_namebuf[CONFIGFS_ITEM_NAME_LEN];
        const struct config_item_type   *ci_type;
};
struct config_item_type {
        struct module                           *ct_owner;
        struct configfs_item_operations     *ct_item_ops;
        struct configfs_group_operations    *ct_group_ops;
        struct configfs_attribute           **ct_attrs;
        struct configfs_bin_attribute       **ct_bin_attrs;
 };
struct configfs_item_operations {
        void (*release)(struct config_item *);
        int (*allow_link)(struct config_item *src, struct config_item *target);
        void (*drop_link)(struct config_item *src, struct config_item *target);
};
struct configfs_group_operations {
        struct config_item *(*make_item)(struct config_group *group, const char *name);
        struct config_group *(*make_group)(struct config_group *group, const char *name);
        int (*commit_item)(struct config_item *item);
        void (*disconnect_notify)(struct config_group *group, struct config_item *item);
        void (*drop_item)(struct config_group *group, struct config_item *item);
};
```

07 创建多级ConfigFS子目录

- 创建目录：make_group

```
struct config_group {
        struct config_item      cg_item;
        struct list_head        cg_children;
        struct configfs_subsystem   *cg_subsys;
        struct config_group     **default_groups;
        struct list_head        group_entry;
};
struct config_item_type {
        struct module                           *ct_owner;
        struct configfs_item_operations     *ct_item_ops;
        struct configfs_group_operations    *ct_group_ops;
        struct configfs_attribute           **ct_attrs;
        struct configfs_bin_attribute     **ct_bin_attrs;
};
struct configfs_group_operations {
        struct config_item *(*make_item)(struct config_group *group, const char *name);
        struct config_group *(*make_group)(struct config_group *group, const char *name);
        int (*commit_item)(struct config_item *item);
        void (*disconnect_notify)(struct config_group *group, struct config_item *item);
        void (*drop_item)(struct config_group *group, struct config_item *item);
};
```

- 层次结构的构建
  - config_group->cg_children
  - config_item->ci_parent

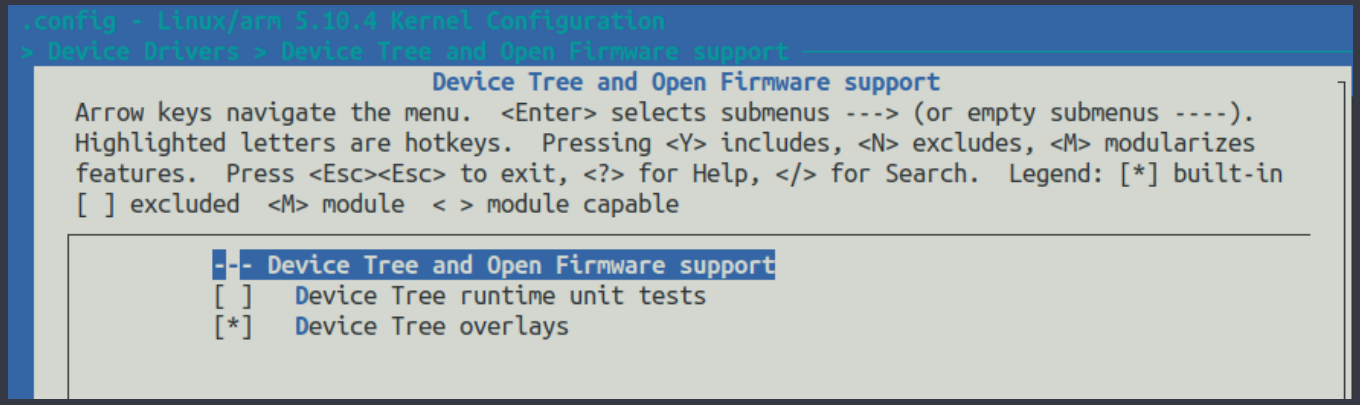
08 ConfigFS mkdir过程分析

- 当我们在用户空间mkdir时…
  - 源码目录：fs/configfs/dir.c： configfs_mkdir
  - 调用config_item_type实现的方法
    - » 执行make_group分支：创建configfs_group
    - » 执行make_item分支：创建item
  - 创建目录
    - make_group分支：
      - » configfs_attach_group ->
      - » configfs_attach_item 创建目录
    - make_item分支：
      - » configfs_attach_item ->
      - » configfs_create_dir(item, dentry, frag) 创建目录

- 小结
  - 若一个config_item_type中，定义了make_group方法，创建目录时会调用该方法
  - 没有定义make_group方法，创建目录时，会调用make_item方法
  - 两者都没定义，无法创建子目录
  - 若实现了make_group方法，会创建子group，每个子config_group对应一个目录，有各自的config_item
  - 每个 config_group下可以创建多个同级子目录，子目录对应的 config_item 使用链表管理
  - 层级关系通过 config_group->cg_children 和 config_item->ci_parent 指针构建

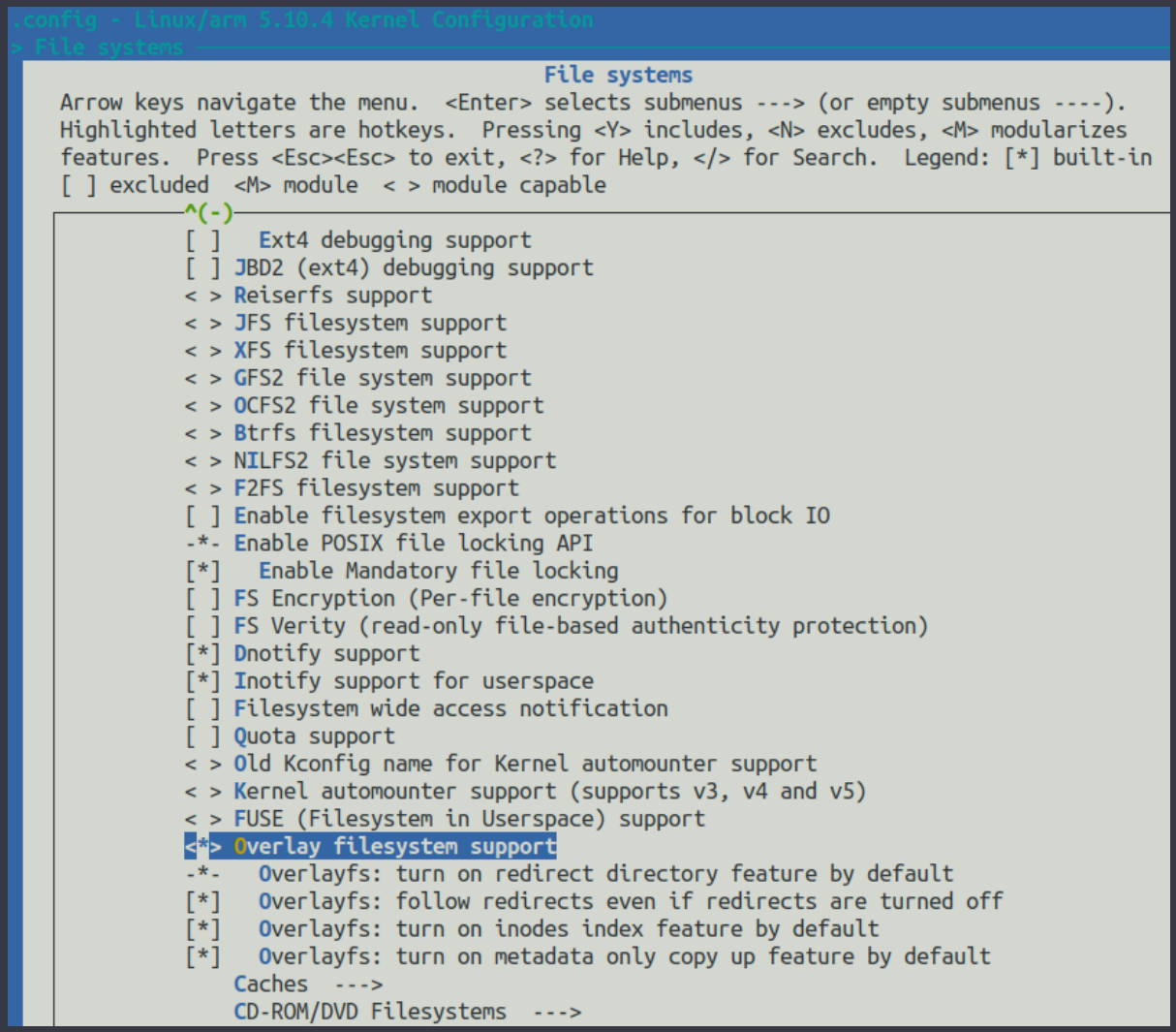
09 实现设备树的overlay功能

- 如何实现设备的overlay？
  - 内核配置
  - 加载
  - 解析
  - 驱动源码分析
  - 编译运行




```
# insmod demo.ko
# cd sys/kernel/config/device-tree/overlays
```

- 设备树的overlay内核配置
  - CONFIG_OF_OVERLAY：使能overlay功能
  - CONFIG_OVERLAY_FS



10 向设备树动态添加节点（上）

- 编写第一个设备树overlay文件
  - overlay语法
  - 编译
  - 加载

```
# insmod demo.ko
# cd sys/kernel/config/device-tree/overlays
# mkdir hello
# ls hello
  dtbo status
# cat /hello.dtbo > hello/dtbo
# echo 1 > hello/status
# ls /proc/device-tree
  hello
```

```
# echo 1 > hello/status
# rmdir hello
# rmmod demo.ko
```

11 向设备树动态添加节点（下）

- 反编译设备树的overlay文件
  - 反编译后的设备树语法分析

```
/plugin/;
/ {
        /* set of per-platform overlay manager properties */

        fragment@0 {
                target = <&target-label>; /* or target-path */

                __overlay__ {
                        /* contents of the overlay */
                };
        };

        fragment@1 {
                /* second overlay fragment... */
        };
};
```

12 设备树overlay加载过程分析

- 加载过程分析
  - 顶层目录的创建
  - 用户动态创建目录
  - 加载dtb文件过程
  - 驱动源码分析

13 设备树overlay解析过程分析

- 内核如何解析新加载的dtb文件
  - 控制开关：status
  - 申请内存，将dtbo展开为device_node
  - 处理label/phandle引用，找到真正的node
  - 创建和处理overlay changeset：增改删
  - 内核API

of_fdt_unflatten_tree：将 dtb 展开为 device_node
of_resolve_phandles ：动态调整phandle值
of_overlay_create ： Creates and applies an overlay
of_overlay_destroy

14 同时加载多个设备树overlay

- 实验
  - 同时加载多个dtbo文件
  - 动态修改某个属性