Linux 驱动开发进阶

深入剖析 LINUX 驱动源码

李山文 第一版

野生技术协会 | 野生技术协会大厅

前　言

在《Linux 驱动开发入门》和《Linux 块设备驱动开发入门》中简单讲解了 Linux 的字符设备驱动和块设备驱动的大体开发流程和基本的一些框架。然而这仅仅是驱动开发的开始，因为在 Linux 操作系统中，整个源码非常庞大，由于驱动程序运行在内核态，因此其开发过程中对问题进行定位也变得异常困难，一个小小的错误可能就导致内核崩溃。因此我们必须对 Linux 内核有一个清晰的认识才能在驱动开发过程中尽可能减少不必要的问题以及如何定位问题。

本书将深入讲解 Linux 驱动开发所涉及到的内核相关知识，尽可能用图示的形式展现其执行过程。

第一章 文件系统

在 Linux 中，有一种“一切皆文件”的设计理念，对于应用程序而言，应用开发者关心最多的便是设备节点，这些设备节点就是文件，因此我们有必要对文件的有一个非常清晰的认识。Linux 中的文件系统是非常复杂的，因为在 Linux 中，不仅仅只有一种文件系统，还有非常多的不同种类的文件系统，我们熟悉的便是 ext2、ext3、ext4、sysfs、procfs、jfs、vfat、fat32 等等，那 Linux 又是如何支持这么多文件系统的呢？下面我们来深入了解其背后的原理。

1.1 应用程序执行过程

在 Linux 中，用户程序是如何运行的呢？或者说我们写一个应用程序，例如 hello.exe，那么这个程序是如何运行的呢？这个程序的在执行过程中到底做了哪些操作呢？

首先我们会通过编辑器编写 hello.c 文件，然后用编译器将其编译为可执行文件，这时，我们就可以利用./hello.exe 来执行该文件了，在执行该命令后，当前 shell 会创建一个子进程，创建的过程使用 fork 函数，该子进程中会调用 execve 系统调用函数来执行我们的 hello.exe 文件，此时 execve 系统调用会出发软中断来让内核执行系统调用_system_call 函数，该函数需要将用户程序中的信息全部压入到栈中，其中包括数据段，代码段等上下。

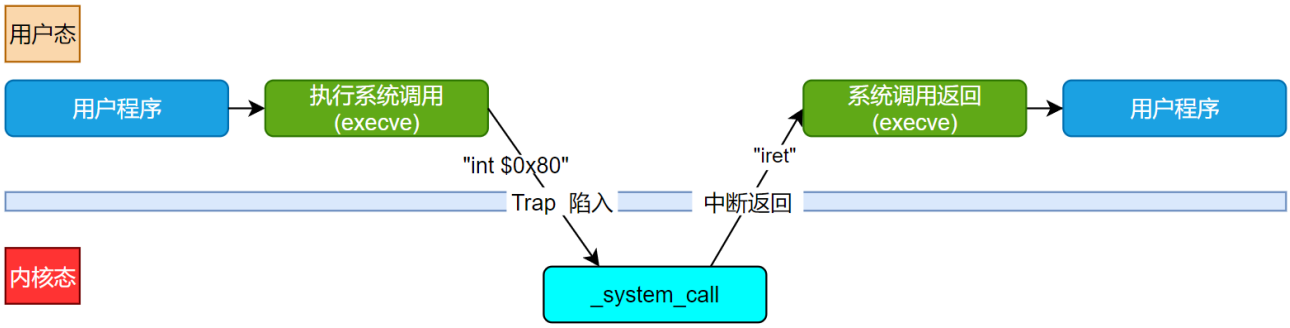

图 1-1 系统调用简化过程

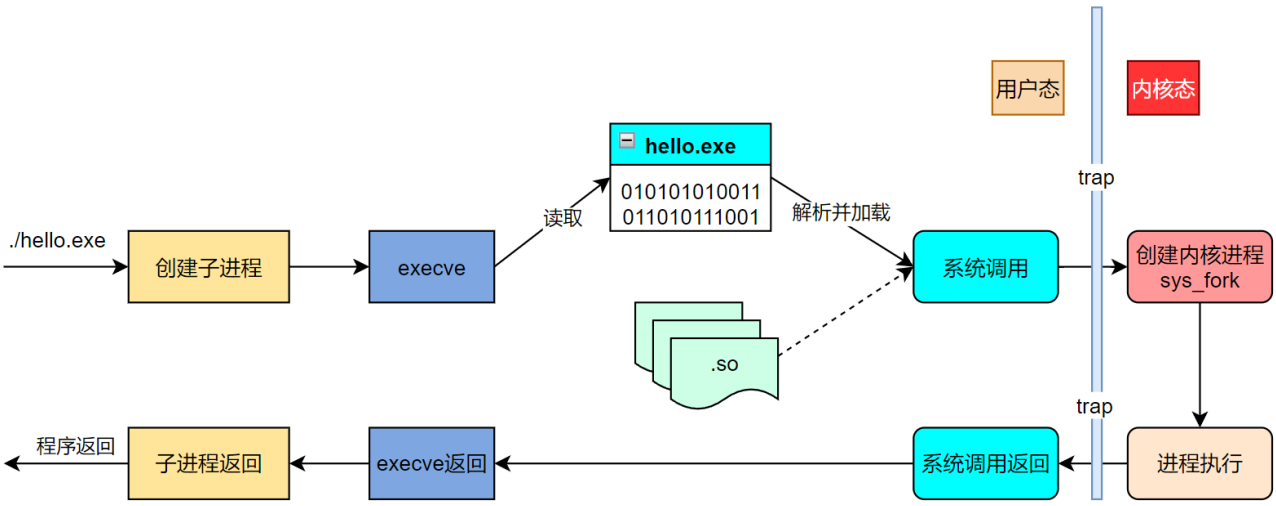

图 1-2 用户应用程序执行过程

可以看到，实际上用户的进程并没有 CPU 的权限，或者应用程序并不能直接执行，而是需要先使用 execve 系统调用来解析可执行文件并出发软中断（trap），此时内核会调用内核态的系统调用函数即__system_call 函数来为该可执行文件建立必要的环境，即创建了一个内核进程，然后再执行该进程。执行完毕后，即系统调用返回，此时整个应用程序就运行完毕。实际上整个过程中比上图要复杂的多。

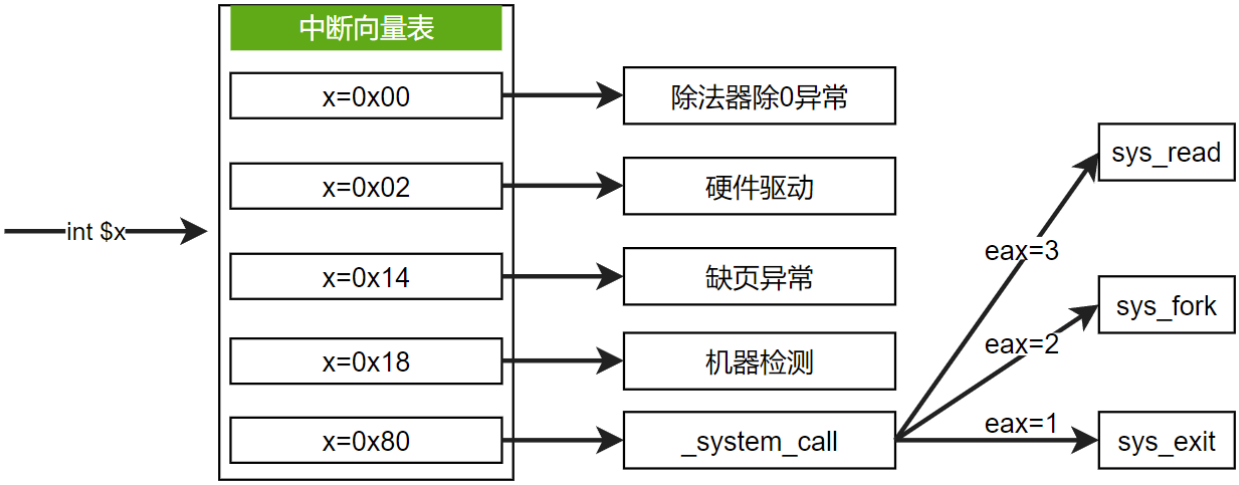

图 1-3 i386 下 Linux 的中断向量异常

如上图所示，对于 i386 机器而言，Linux 将中断向量[1]进行了定义，每个中断都有其特定的中断服务函数，每个中断服务函数执行各自的异常处理。系统调用使用的是 0x80 中断偏移，而在中断返回时，需要使用 iret 命令来结束中断。

网上一个博客总结了一个图，如下：

1

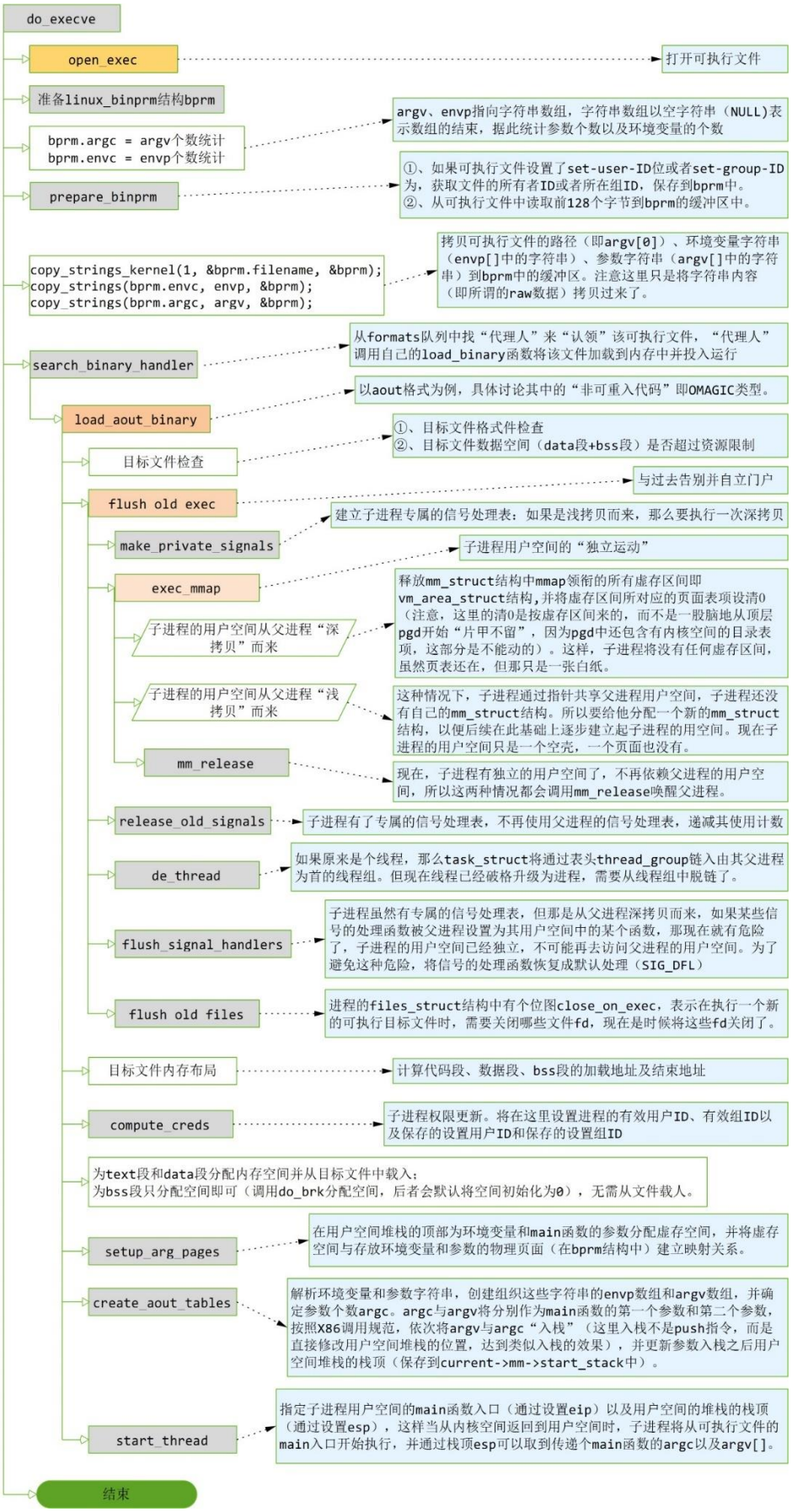

图 1-4 execve 执行流程

1.2 VFS 虚拟文件系统

在块设备驱动中，我们已经对 VFS 有了一些了解，VFS 是一种虚拟文件系统，该文件系统并不是真正的文件系统，而是提供了一个标准的接口。如下图所示，在应用程序中，我们会调用 open，write，read 等系统调用函数，这些函数在执行之后会进入系统调用，从而调用 Linux 内核中的相关接口函数，那什么是系统调用呢？系统调用和普通的函数调用非常相似，区别仅仅在于，系统调用由操作系统核心提供，运行于内核态；而普通的函数调用由函数库或用户自己提供，运行于用户态。

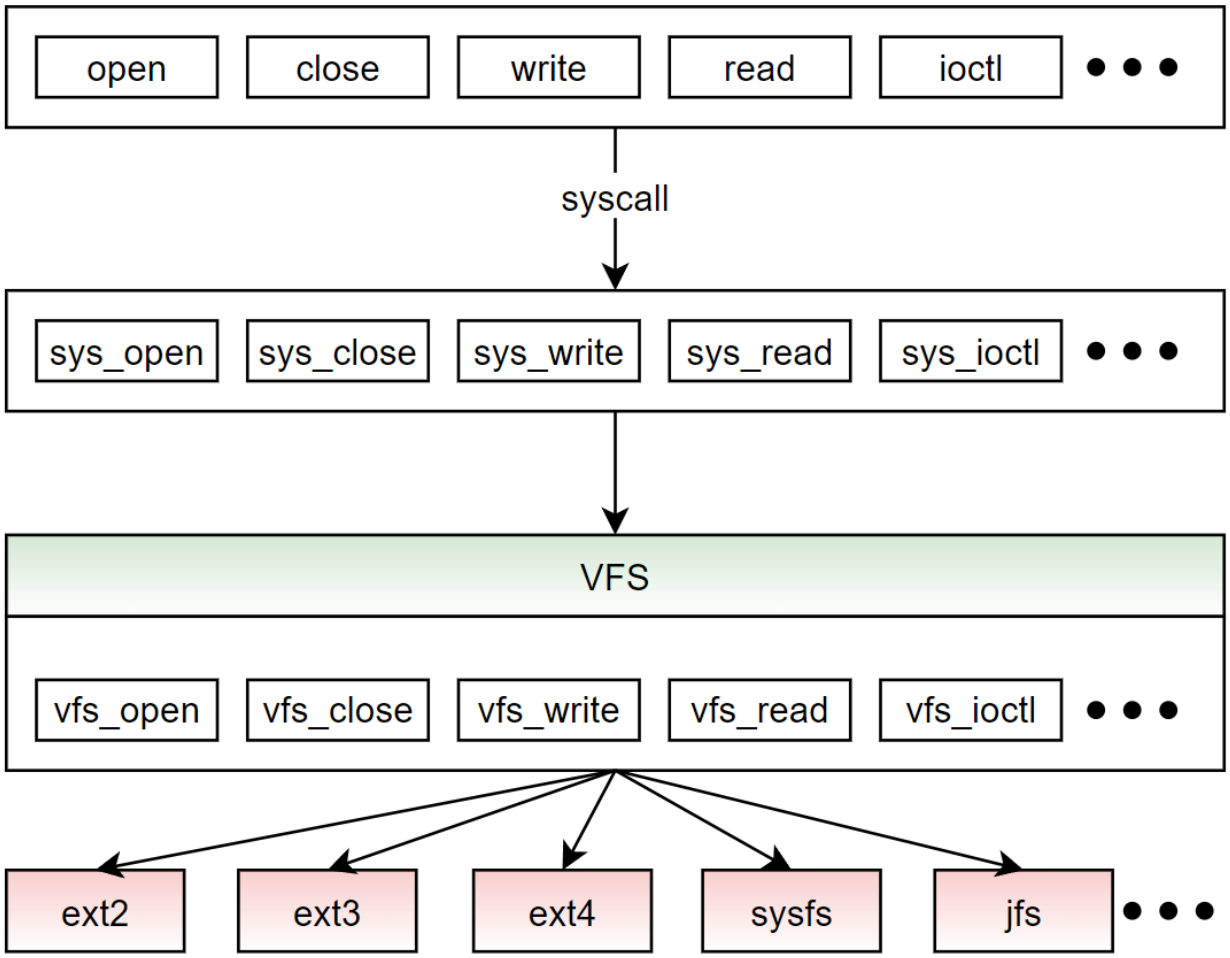

图 1-5 文件系统中 VFS 的作用

从上图可以看到，用户程序中的一个 open 系统调用函数就经过了很多次调用。我们再来看下 file 和 inode 这两个非常重要的结构（具体源码可以查看 include/linux/fs.h 文件）。

首先在应用程序中，我们会使用 open 函数来打开一个文件，如下图

```
int fd=open(“/dev/led”,O_RDWR);
```

上面这个函数就是系统调用中的一个 open 函数，该调用函数打开 dev 目录下的 led 设备文件，fd 是该函数的返回值，即文件描述符，这个值是一个 int 类型的，要了解该值的意义，下面我们来简单了解下什么是文件描述符。

1.2.1 Linux 文件系统结构

文件最终需要存放在 ROM 中的，这样掉电就不会丢失，目前常见的 ROM 有磁盘、机械硬盘、固态硬盘等等。但是对于数据而言，我们不能直接写入到硬盘中，而是需要按照固定的格式进行组织，组织好数据结构才能写到存储介质中。因此文件系统就出现了，为了更高效的管理文件，文件系统采用的是链式存储结构，即不是按顺序存储的，这样做的好处是因为链式存储在文件搬移和擦除上相比于顺序存储有着无法比拟的优势，因此目前几乎所有的文件系统都是基于链式结构的。

对于 Linux 文件系统而言，由于其需要支持多种不同的文件系统，因此引入了虚拟文件系统的概念，其中最重要的便是四个非常重要的结构：

- 超级块（super block）
- i 节点（inode）
- 目录项（dentry）
- 文件（file）

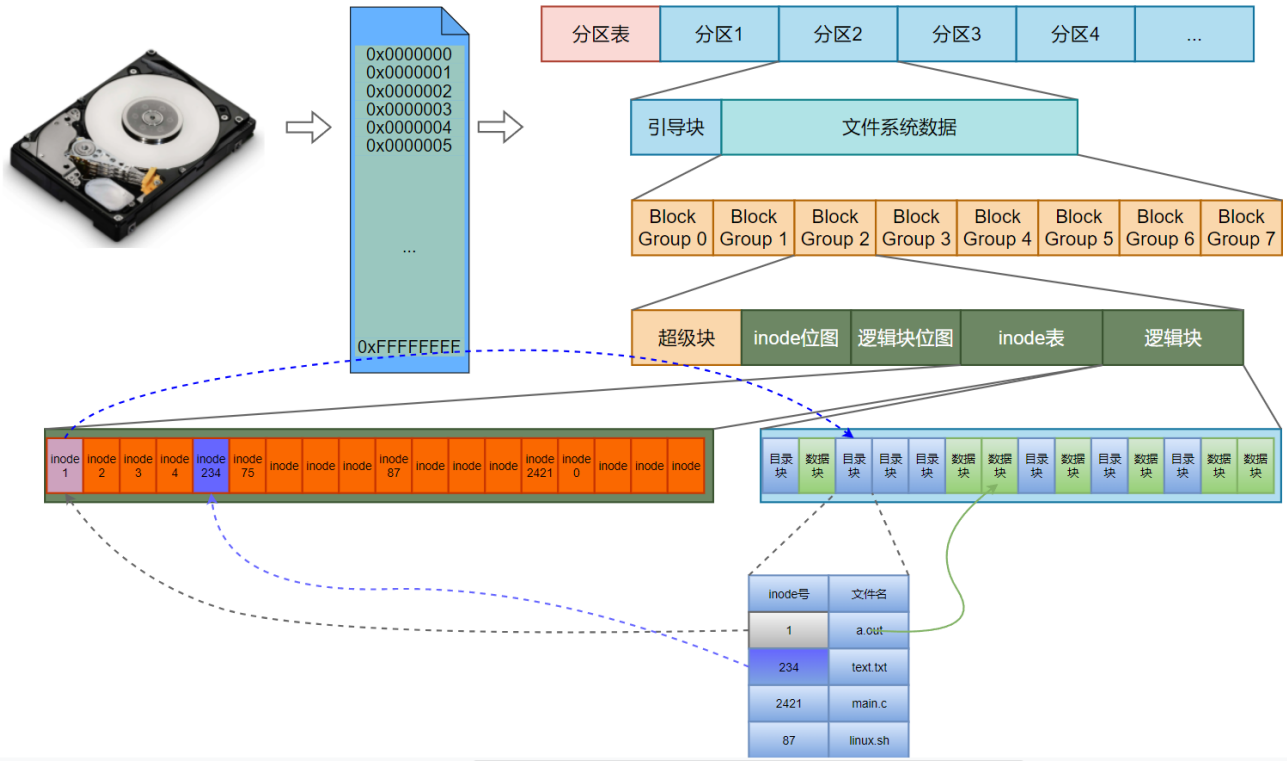

图 1-10 Linux 文件系统结构

1.2.2 文件描述符（fd）

fd 即 file descriptor 的简写，文件描述符是内核为了高效管理已被打开的文件所创建的索引，用于指代被打开的文件，对文件所有 I/O 操作相关的系统调用都需要通过文件描述符。那么在应用程序中，使用 open 系统调用函数到底做了哪些操作呢？

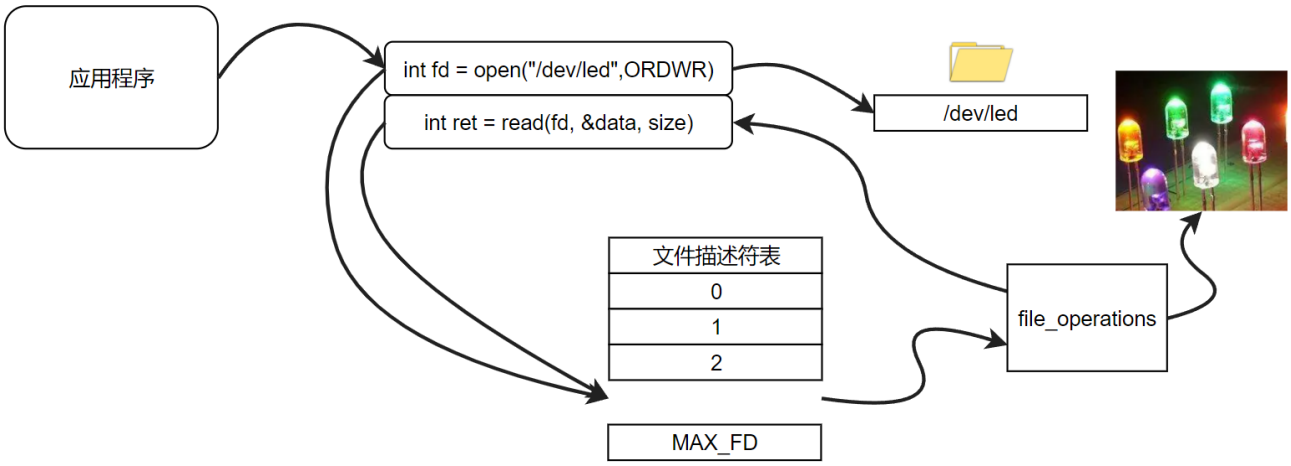

图 1-6 读写文件中的描述符

当应用程序中使用 open 系统调用时，此时程序会通过软中断陷入内核态，并执行相应的程序，首先会根据 open 函数传进的第一个参数找到 led 设备文件的位置，同时内核中的线程会创建一个文件描述符表，用来记录存放文件指针，在 Linux 内核中也存在一张很大的文件描述符表，该表和每个进程独立的文件描述符表不同，该表是系统级的，其中存放的信息更加全面，而对于每个进程独立的文件描述符表仅仅只是该系统级的一个索引。

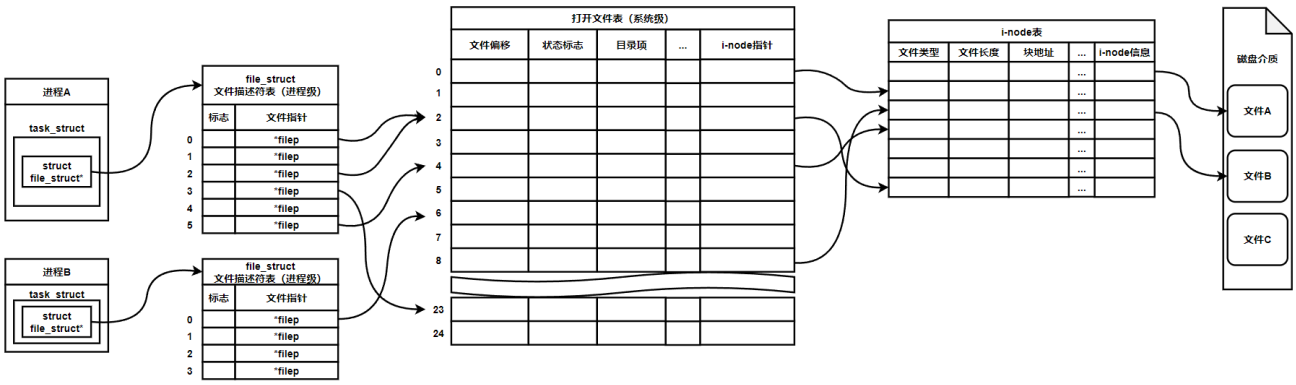

图 1-7 文件打开过程

对于进程级的文件描述符表而言，该表是在进程创建的时候会生成，同时进程中有一个 task_struct 结构体，该结构体包含了 struct file_struuct 指针，该指针就是指向进程级的文件描述符表，这样每个进程在打开文件的时候都会在文件描述符表中添加索引，同时将 filep 指针指向系统级的文件描述符表，而系统级的文件描述符表会非常大，其中包含着文件类型、文件标志、目录项、i-node 指针等。其中需要重点说明下 i-node 指针，实际上在 Linux 操作系统中的每个文件都对应这一个 i-node，该 i-node 记录着文件的一些信息，例如文件类型、文件长度、文件的物理存放地址等。因此所有的文件读写都要先去找对应的 i-node 表，然后找到文件的具体物理地址就可以对其进行读写了，这个过程是文件系统来完成的，用户是感觉不的。

实际上上面的整个过程都只是简化的流程，真是情况比上图要复杂的多，现在我们深入源码来看下具体的过程。每创建一个进程都会包含一个 task_struct 结构体，该结构体中含有 file_struct *files 指针，该指针指向一个 fdtable 结构体，该结构体指向 struct_file 中的某一个 fd，然后 fd 指向该进程中具体的 struct file 结构体。struct file 结构体中包含这 f_inode 指针和 f_ops 指针，f_inode 指针指向 inode 结构体，该结构体对应着硬盘上的一个文件，记录着文件的信

息。f_ops 指针指向设备驱动程序中的 file_operations 结构体。

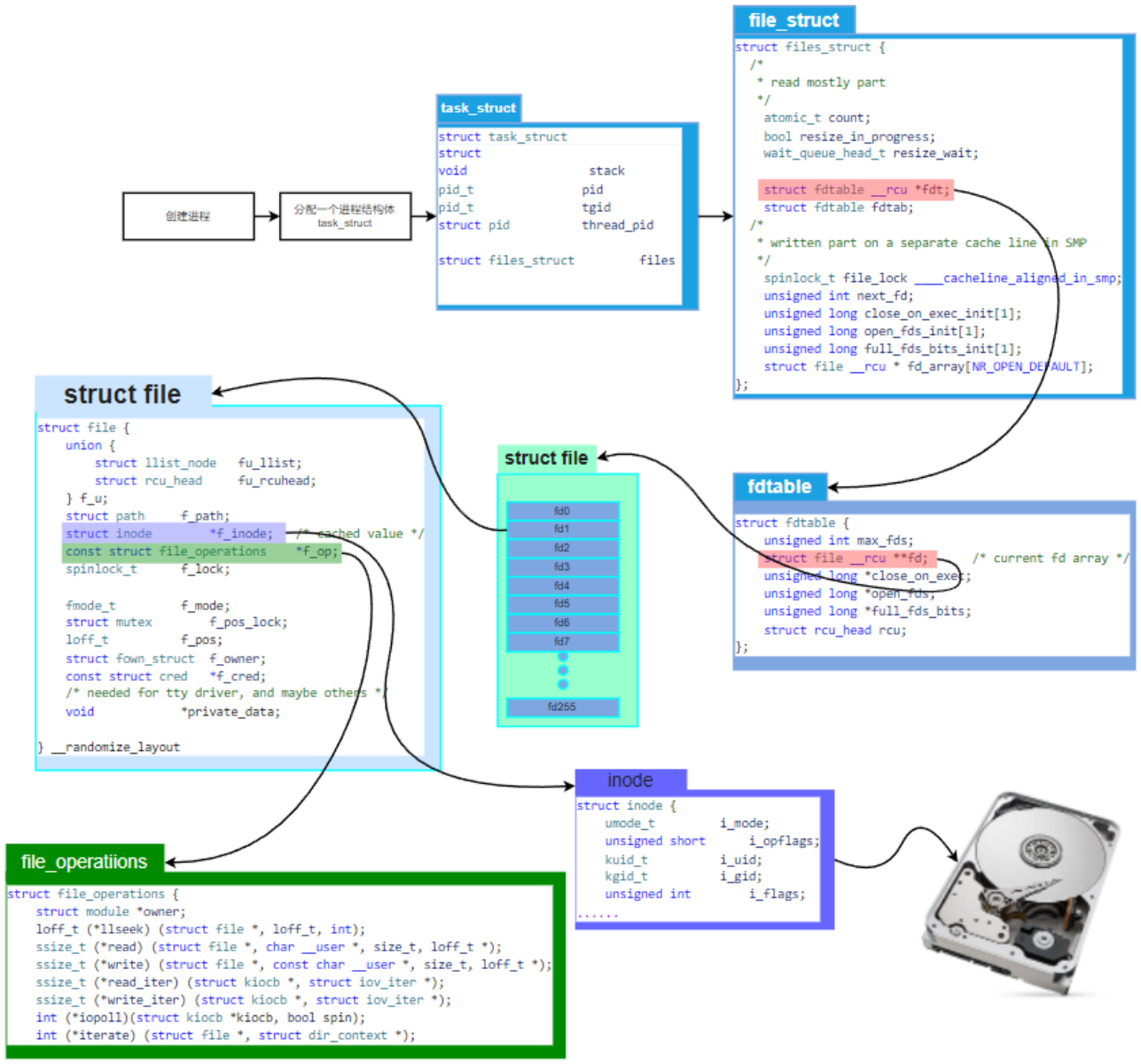

图 1-8 文件打开中的涉及到的数据结构

我们观察下发现 struct file 结构体中还有一个 f_path 结构体，该结构体也是非常重要的，当我们打开一个文件时，我们会将文件的路径保存到该结构体中，这样我们使用文件句柄的时候就可以知道文件的路径了。下面我们来详细看下这个 f_path 里面是啥，如下所示，下面时 f_path 的具体结构体：

```
struct path {
    struct vfsmount *mnt;
    struct dentry *dentry;
} __randomize_layout;
```

该结构体有两个成员，一个时 vfsmount *mnt 和 dentry *dentry，前者是虚拟文件系统的挂载点，前面我们说过，Linux 支持的文件系统非常多，每个文件系统的数据结构也不同，因

此为了能够准确的知道当前文件到底挂载在哪个文件系统上，我们需要将文件的挂载点保存下来。dentry 是一个目录项，即记录着该文件在该挂载点的目录路径。需要注意的是没文件名并不保存在 inode 中，而是保存在目录项 dentry 中。

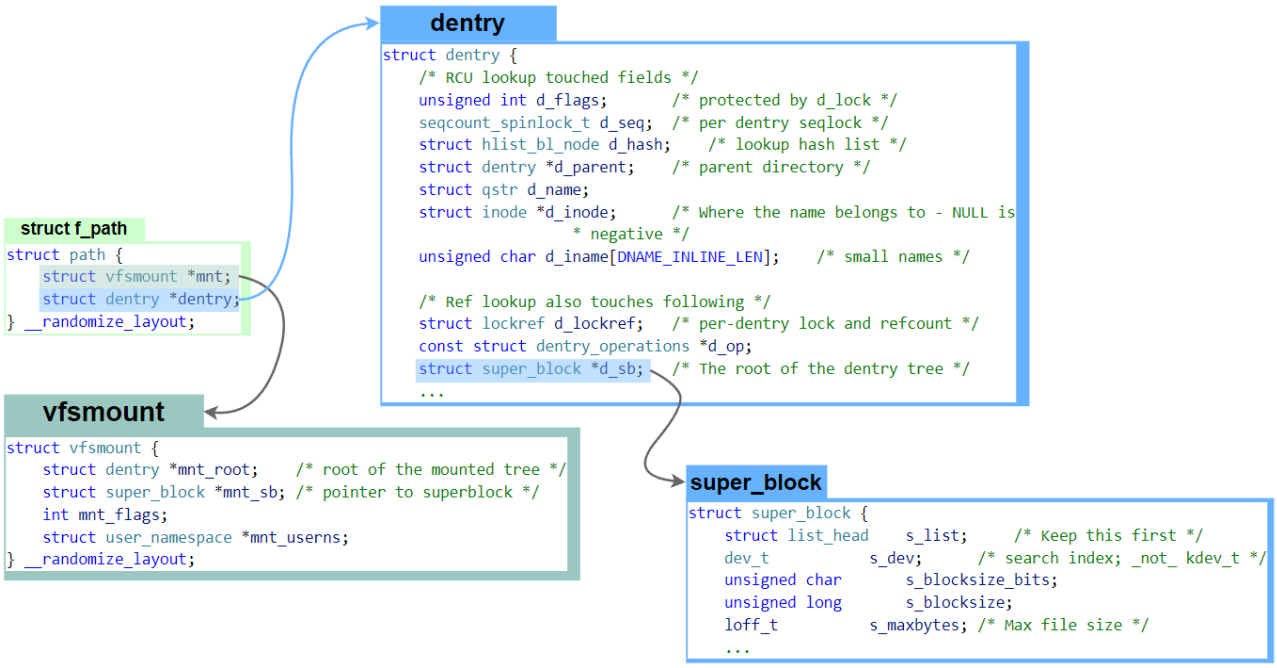

图 1-9 目录项

第二章 热插拔设备

在《Linux 驱动开发入门》中详细讲解了 sysfs 文件系统的使用，该文件系统与 Linux 的设备模型进行了绑定，很大程度上提高了 Linux 结构，同时也为驱动开发者提供了一种 Debug 手段。

实际上 sysfs 最重要的目的并不是给驱动开发者提供调试用，其最重要的目的是该文件系统可以很好的解决电源管理、热插拔设备。

第三章 内存管理