linux-VFS文件系统结构分析

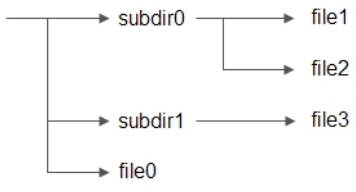
VFS是Linux非常核心的一个概念，linux下的大部分操作都要用到VFS的相关功能。这里从使用者的角度，对VFS进行了简单说明。使用者不但需要知道Linux下有哪些文件操作的函数，还需要对VFS的结构有一个比较清晰的了解，才能更好的使用它。例如hard link 与symbolic，如果没有VFS结构的相了解，就无法搞清楚如何使用它们。

本文首先是建立了一个简单的目录模型，然后介绍该目录在VFS的结构，最终总结出如何使用各个文件操作函数。

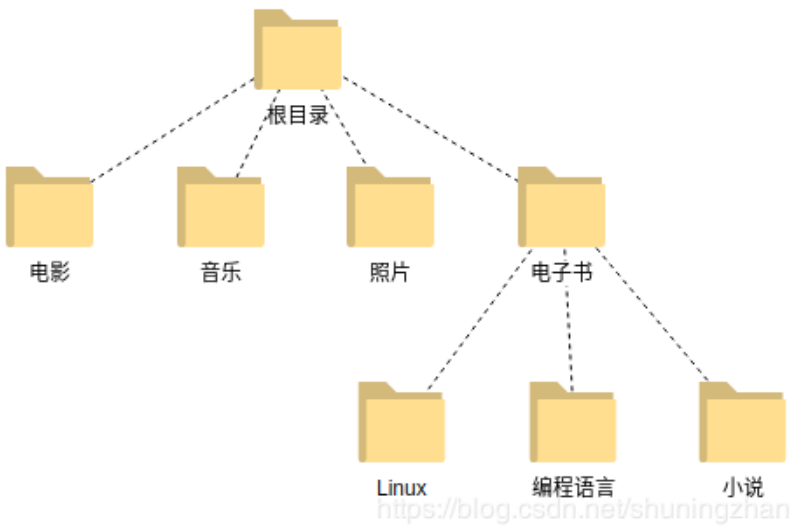
本着简单使用的原则，主要使用了分析加猜测的方法。鉴于本人水平有限，文中不免会有些错误。欢迎各位读者理性阅读，大胆批判。您的批判是我进步的动力。

1. 目录模型

以下面的目录为例。

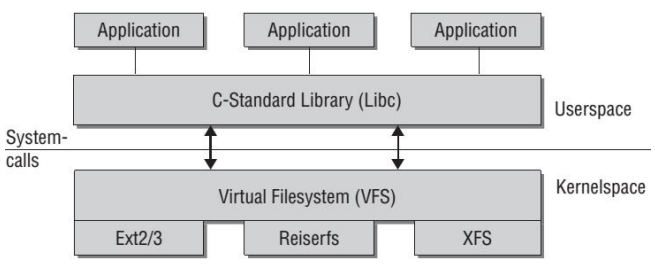


dir为第一级目录，dir中有subdir0与subdir1两个子目录与一个文件file0。“subdir0”中有两个文件file1与file0。subdir1中有一个文件file3。我们这里给出一个更加形象的插图：



2 VFS的概念

VFS是Linux中的一个虚拟文件文件系统，也称为虚拟文件系统交换层（Virtual Filesystem Switch）。它为应用程序员提供一层抽象，屏蔽底层各种文件系统的差异。如下图所示：



不同的文件系统，如Ext2/3、XFS、FAT32等，具有不同的结构，假如用户调用open等文件IO函数去打开文件，具体的实现会非常不同。为了屏蔽这种差异，Linux引入了VFS的概念。相当于是Linux自建了一个新的贮存在内存中的文件系统。所有其他文件系统都需要先转换成VFS的结构才能为用户所调用。

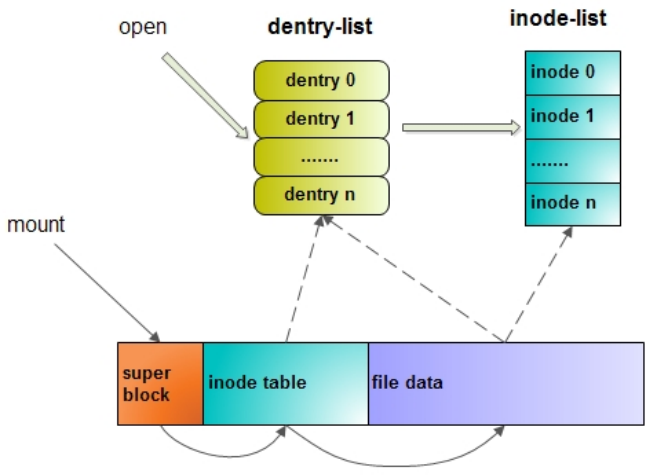
3 VFS的构建

所谓VFS的构建就是加载实际文件系统的过程，也就是mount被调用的过程。如下图所示，以mount一个ext2的文件系统为例。

这是一个经过简化的Ext2磁盘结构，只是用于说明用它构建VFS的基本过程。

mount命令的一般形式为：mount /dev/sdb1 /mnt/mysdb1

/dev/sdb1是设备名，/mnt/mysdb1是挂载点。



VFS文件系统的基本结构是dentry结构体与inode结构体。

Dentry代表一个文件目录中的一个点，可以是目录也可以是文件。

Inode代表一个在磁盘上的文件，它与磁盘文件一一对应。

Inode与dentry不一定一一对应，一个inode可能会对应多个dentry项。（hard link）

Mount时，linux首先找到磁盘分区的super block，然后通过解析磁盘的inode table与file data，构建出自己的dentry列表与indoe列表。

需要注意的是，VFS实际上是按照Ext的方式进行构建的，所以两者非常相似（毕竟Ext是Linux的原生文件系统）。

比如inode节点，Ext与VFS中都把文件管理结构称为inode，但实际上它们是不一样的。Ext的inode节点在磁盘上；VFS的inode节点在内存里。Ext-inode中的一些成员变量其实是没有用的，如引用计数等。保留它们的目的是为了与vfs-node保持一致。这样在用ext-inode节点构造vfs-inode节点时，就不需要一个一个赋值，只需一次内存拷贝即可。

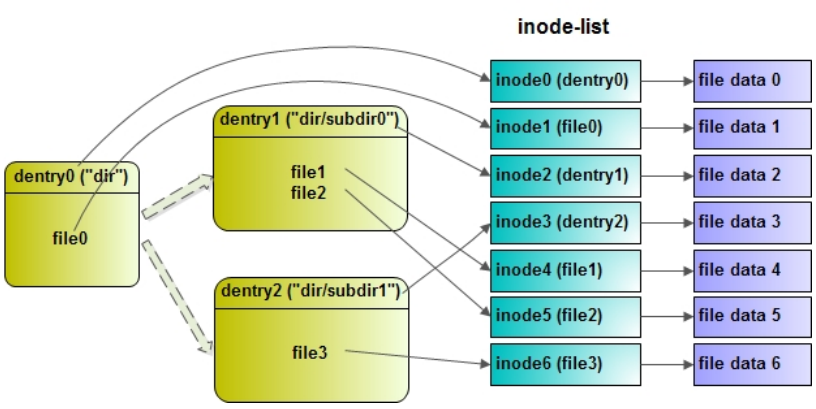
如果是非EXT格式的磁盘，就没有这么幸运了，所以mount非EXT磁盘会慢一些。

4. VFS的结构

构建出VFS文件系统后，下一步是把第一节中提到的目录模型映射到VFS结构体系中。

上文提到了VFS主要由denty与inode构成。Dentry用于维护VFS的目录结构，每个dentry项就代表着我们用ls时看的的一项（每个目录和每个文件都对应着一个dentry项）。Inode为文件节点，它与文件一一对应。Linux中，目录也是一种文件，所以dentry也会对应一个inode节点。

下图是第一节中的目录模型在VFS中的结构。



5 Dentry cache

每个文件都要对应一个inode节点与至少一个dentry项。假设我们有一个100G的硬盘，上面写满了空文件，那个需要多少内存才能重建VFS呢?

文件最少要占用1个block（一般是4K）。假一个dentry与一个inode需要100byte，则dentry与inode需要占用1/40的空间。100G硬盘则需要2.5G空间。最近都开始换装1T硬盘了，需要 25G的内存才能放下inode与dentry，相信没有几台电脑可以承受。

为了避免资源浪费，VFS采用了dentry cache的设计。

当有用户用ls命令查看某一个目录或用open命令打开一个文件时，VFS会为这里用的每个目录项与文件建立dentry项与inode，即“按需创建”。然后维护一个LRU（LeastRecently Used）列表，当Linux认为VFS占用太多资源时，VFS会释放掉长时间没有被使用的dentry项与inode项。

需要注意的是：这里的建立于释放是从内存占用的角度看。从Linux角度看，dentry与inode是VFS中固有的东西。所不同的只是VFS是否把dentry与inode读到了内存中。对于Ext2/3文件系统，构建dentry与inode的过程非常简单，但对于其他文件系统，则会慢得多。

了解了Dentry cache的概念，才能明白为何下面会有两种定位文件的方式。

6 无denty时定位文件

因为上面提到的Denty Cache，VFS并不能保证随时都有dentry项与inode项可用。下面是无dentry项与inode项时的定位方式。

为了简化问题，这里假设已经找到了dir的dentry项（找到dentry的过程会在后面讲解）。

首先，通过dir对应的dentry0找到inode0节点，有了inode节点就可以读取目录中的信息。其中包含了该目录包含的下一级目录与文件文件列表，包括name与inode号。实际上用ls命令查看的就是这些信息。 “ls -i”会显示出文件的inode号。

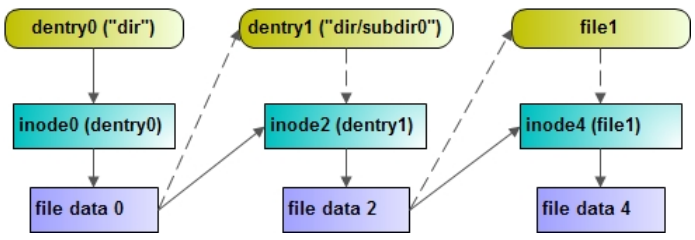
```
> ls -i
975248 subdir0  975247 subdir1   975251 file0
```

然后，根据通过根据subdir0对应的inode号重建inode2，并通过文件数据（目录也是文件）与inode2重建subdir0的dentry节点：dentry1。

```
> ls -i
975311 file1 975312 file2
```

接着，根据file1对应的inode号重建inode4，并通过文件数据与inode4重建file1的dentry节点。

最后，就可以通过inode4节点访问文件了。



注意：文件对应的inode号是确定的，只是inode结构体需要重新构造。

7　有dentry时定位文件

一旦在Dentry cache中建立了dentry项，下次访问就很方便了。

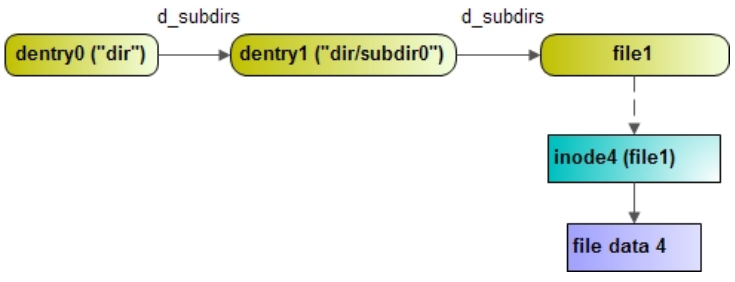
Dentry中的一个关键变量是d_subdirs，它保存了下一级目录的列表，用于快速定位文件。

首先，在代表dir目录的dentry0的d_subdirs中查找名字为“subdir0”的dentry项，找到了dentry1。

然后在dentry1中查找名字为“file1”的dentry项，然后找到了file1对应的dentry项，

最后通过file1对应的dentry项获得file1对应的inode4。

与无dentry项时比较，有dentry项时的操作精简了许多。

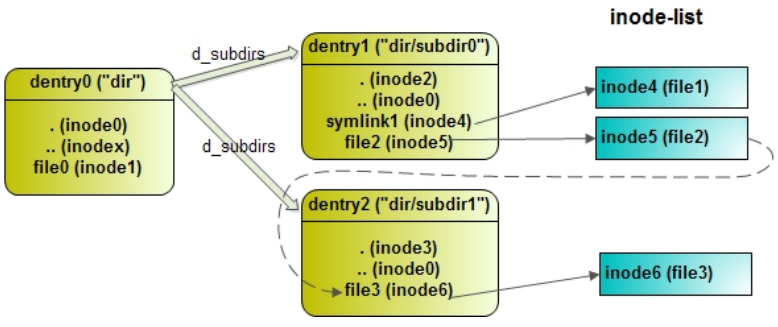


8　Symbolic link

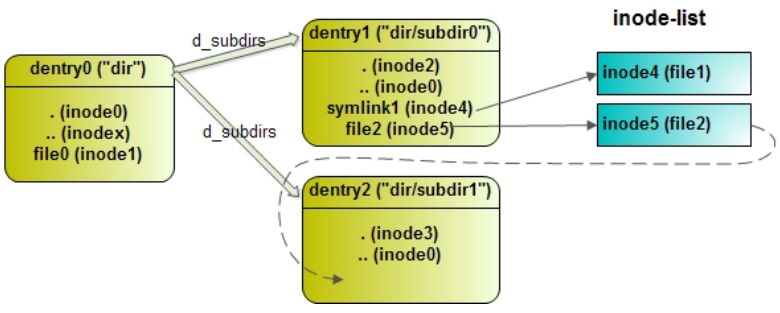
建立symboliclink的命令为： ln -s 源文件目标文件

Linux中的symbolic link类似于Windows系统中的快捷方式。如下图所示，symlink1是指向file1的symbolic link。symlink1本身也是文件，因此有自己独立的inode节点。symlink中实际存储的是源文件的相对路径。

大部分文件操作会直接对symbolic link指向的目标进行操作，比如open（“symlikn1”），实际上打开的是file3。



如果file3不在会发生什么事情呢？open函数照样会按照symlink1中的文件路径打开文件。但file3不存在，因此会报错说文件不存在。



9　hard link

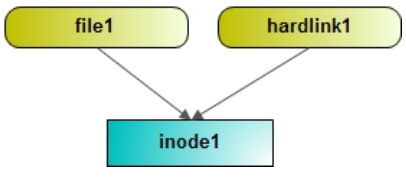
Linux除了symbolic link，还有hard link的概念。

Hard link建立实际上是dentry项的一个拷贝，它们都指向同一个inode节点。当我们使用write改写file1的内容时，hardlink1的内容也会被改写，因为所以实际上它们是同一个文件。

如下图所示，hardlink1是file1的一个hard link。它们都指向同一个inode1节点。Inode1中有一个计数器，用于记录有几个dentry项指向它。删除任意一个dentry项都不会导致inode1的删除。只有所有指向inode1的dentry都被删除了，inode1才会被删除。

他们实际

从某种意义上讲，所有dentry项都是hard link。

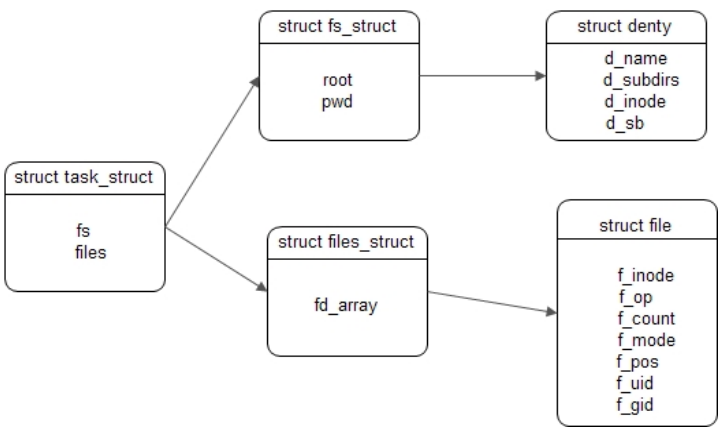


10 进程对文件的管理

进程控制块task_struct中有两个变量与文件有关：fs与files。

files中存储着root与pwd两个指向dentry项的指针。用户定路径时，绝对路径会通过root进行定位；相对路径会的通过pwd进行定位。（一个进程的root不一定是文件系统的根目录。比如ftp进程的根目录不是文件系统的根目录，这样才能保证用户只能访问ftp目录下的内容）

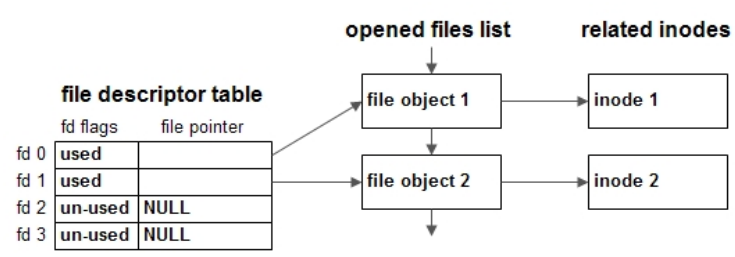
fs是一个file object列表，其中每一个节点对应着一个被打开了的文件。当进程定位到文件时，会构造一个file object，并通过f_inode 关联到inode节点。文件关闭时（close），进程会释放对应对应file object。File object中的f_mode是打开时选择的权限，f_pos为读写位置。当打开同一个文件多次时，每次都会构造一个新的file object。每个file object中有独立的f_mode与f_pos。



11 open的过程

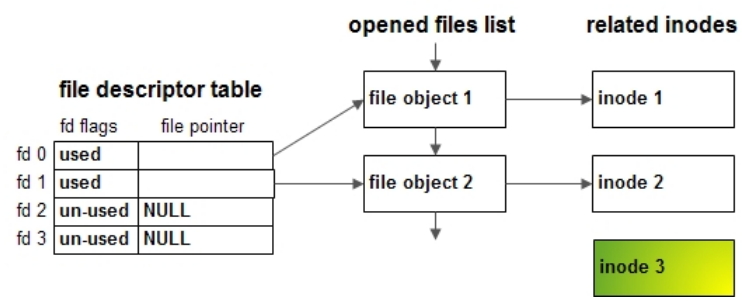
打开文件涉及到里一系列的结构调整，这里分步骤进行说明：

首先建立一个文件管理结构，如下图所示，该进程已经打开了两个文件，接下来我们再打开一个新文件。



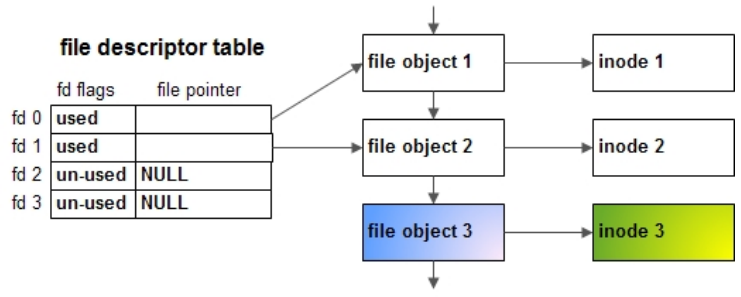
第一步：找到文件；

从上文中能定位到我们文件的inode节点，找到了inode节点也就找到了文件。



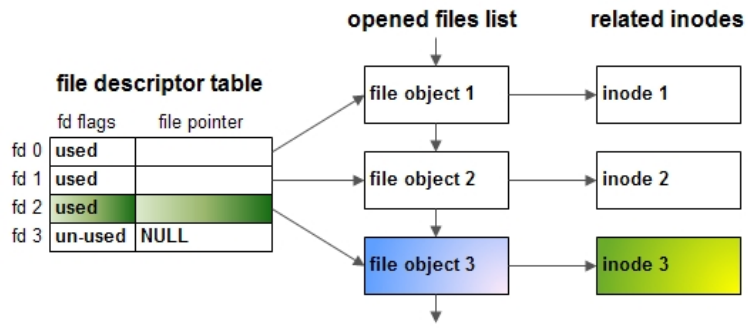
第二步：建立file object；

建立一个新的file object对象，放入file object对象列表，并把它指向inode节点。



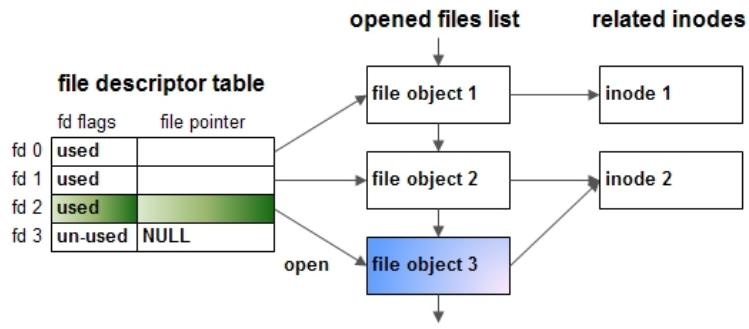
第三步：建立file descriptor

file descriptor就是进程控制块task_struct中files中维护的fd_array。因为是数组，所以file descriptor实际上已经预先分配好空间了，这里这是需要把某个空闲的file descriptor与file object关联起来。这个file descriptor在数组中的索引号就是open文件时得到的文件fd。

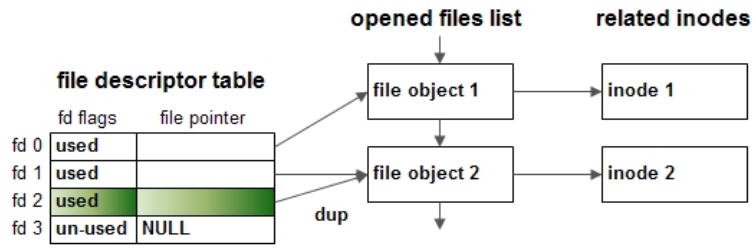


12 open与dup

同一个文件是可以open多次的，结构如下图所示。每次open都会建立一个新的file descriptor与file object。然后指向同一个文件的inode节点。下图中，假设open的文件与fd1指向的是同一个文件，则新创建的file object 2与fd1的file object 2会指向同一个inode2节点。



Linux还提供了dup功能，用于复制file descriptor。使用dup不会建立新的非file object，所以新建立的file descriptor会与原filedescriptor同时指向同一个file object。下图中，我们通过dup(fd1)得到了fd2，则fd2与fd1指向了同一个file object2。

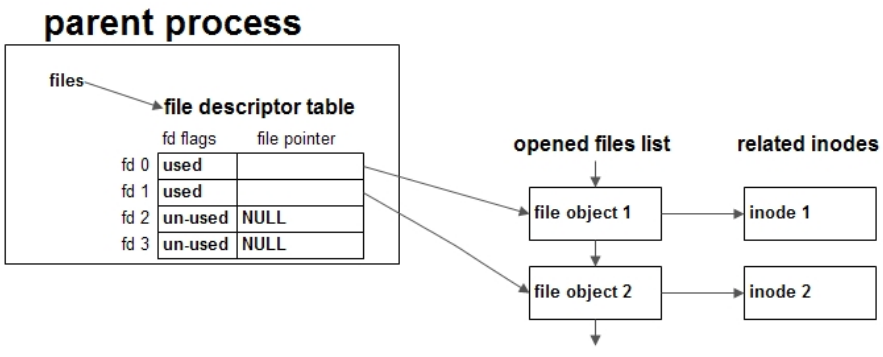


两次open后由于会生成新的object，所以文件读写属性、文件读写位置（f_pos）等信息都是独立的。使用dup复制filedescriptor后，由于没有独立的object，所以修改某个fd的属性或文件读写位置后，另一个fd也会随之变化。

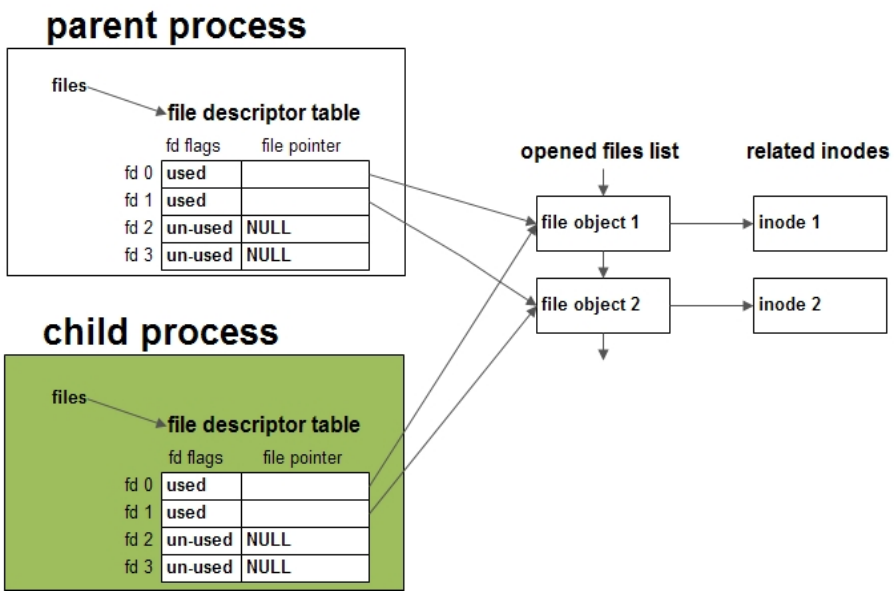
13 Fork对打开文件的影响

Dup的操作与fork一个子进程时的操作类似。

下图是已有父进程的文件结构：



使用fork后的结构如下。同样是没有创建新的file object，因此当对parent process中的fd1进行文件指针的移动时（如读写），childprocess中的fd1也会受影响。也即是说opened files list不是进程的一部分，因此不会被复制。Opened files list应该是一个全局性的资源链表，进程维护的是一个指针列表fd table，所以被复制的只是指针列表fd table，而不是opened files list。



14 文件操作函数解析

通过上面的分析，可以对各个函数的作用域与使用方式有更清晰的了解。下面列出了常用的文件操作：

函数名	作用对象	说明
creat	dentry, inode	创建文件时会创建新的dentry与inode
open	file object	如果文件不存在，且有O_CREAT参数，则会先调用creat
close	file object	删除file object，但不会删除文件。
state/lstate	inode	读取inode的内容。如果目标是symbolic link，stat会读取symbolic link指向的内容；lstat则会读取symbolic link文件本身。
chmod	file object	改变file object中的f_mode
chown/lchown	file object	改变file object中的f_uid与f_gid
truncate	inode	改变文件长度。
read	file object	读文件会改变file object中的f_pos
write	file object，inode	写文件改变file object中的f_pos的同时也会改变文件内容与更新修改时间。
dup	file object	建立一个新的file descriptor，指向同一个file object项
seek/lseek	file object	改变file object中的f_pos
link	dentry	创建新的dentry项，指向同一个inode节点。
unlink	dentry	删除一个dentry项。如果该dentry指向的inode节点没有被其他dentry项使用，则删除inode节点与磁盘文件。
rename	dentry	修改dentry相中的d_name
readlink	----------	read无法读取symbolic link 文件的内容，需要使用readlink读取
symlink	dentry, inode	作用与creat类似，但创建的文件属性为symbolic link。

注：磁盘文件与inode节点一一对应，所以在表中不再单独列出磁盘文件。

参考文件：

Advanced Programming in the UNIX Environment (3rd) *W. Richard Stevens & Stephen A. Rago*
Understanding the Linux Kernel (3rd) *Daniel P. Bovet & Marco Cesati*