Unix 文件系统的使用

同济大学计算机系　操作系统课程讲稿　邓蓉　　2022-12-16　2023-12-28 修订

Part 1、打开文件结构

《Unix 文件系统的静态结构》介绍了文件系统的元数据，包括 SuperBlock，DiskInode 和 目录项（DirectoryEntry）。这些数据结构永久存放在磁盘上。SuperBlock 是文件系统的元数据；DiskInode 是文件（目录文件）的元数据。目录项是文件树的元数据。

系统正在使用的文件系统和磁盘文件，元数据有内存复本。CPU 访问元数据的内存复本对文件系统实施控制。内存中的元数据，用完写回磁盘。

下面以 Unix V6++为例，介绍文件系统的内存元数据，也就是 Unix 系统的打开文件结构。

一、内存索引节点表 和 内存索引节点

使用文件前，应该将其 DiskInode（磁盘索引节点）复制进内存供文件系统使用。文件使用过程中，如果发生写入操作 或 任何可能更新 DiskInode 状态的行为，内存中 DiskInode 复本会发生改变。文件使用完毕，需要将更新后的 DiskInode 写回磁盘。

1、内存索引节点表　m_Inode 数组，100 个元素。

```
class InodeTable
{
    /* static consts */
public:
    static const int NINODE  = 100; /* 内存Inode的数量 */
public:
    Inode m_Inode[NINODE];    /* 内存Inode数组 */
```

图 1、内存 inode 表

每个元素管理系统正在使用的一个文件，是 Unix 文件系统的内存 inode，用来存放该文件的 DiskInode 和 使用状态。多个进程访问同一个文件时，互斥共享同一个内存 inode。m_Inode 数组有 100 个内存 inode，文件系统最多可以同时访问 100 个不同的文件。

Unix/Linux 系统正常工作时，

- **/dev/tty1**，占用一个内存 inode。
- **主硬盘根目录文件**占用一个内存 inode。
  - FileManager 对象的 rootDirInode 指针指向这个内存 inode。任何进程，使用绝对路径搜索目录树时，从这个节点开始。
  - 关机时，关闭根目录，DisInode 写回磁盘。
  - 系统运行期间，这个内存 inode 不会释放。

- 正在上机的用户，**当前工作目录**占用一个内存 inode。Unix V6++进程，user 结构中，
  - 字段 u_cdir 指向这个内存 inode。
  - 字段 u_curdir 是一个字符串，是当前工作目录的绝对路径名。 Shell 进程读 u_curdir，生成命令行提示符。
  - 用户 logout，关闭当前工作目录，释放内存 inode。
  - cd 命令改变当前工作目录。
    - 关闭先前的当前工作目录文件，释放内存 inode。
    - 申请、占用一个新的内存 inode，打开新的当前工作目录文件。
- 进程 open 打开的每个磁盘文件，占用一个内存 inode。
  - close，释放内存 inode。
- 进程 creat 创建的新文件，占用一个内存 inode。
  - close，释放内存 inode。
- 目录搜索时
  - 进入一个目录文件，这个目录文件占用一个内存 inode。
  - 离开（进入子目录 或 目录搜索结束），释放该目录文件的内存 inode。
- exec 装载应用程序（可执行文件）
  - Unix V6（V6++），连续内存管理方式。exec 系统调用执行时，可执行文件占用一个内存 inode。
    exec 系统调用返回时，进程已装入应用程序的全部图像。释放内存 inode。
  - Linux，请求调页式虚拟存储器。exec 系统调用并不会装入应用程序的全部图像。
    exec 系统调用执行时，可执行文件占用一个内存 inode。进程 exit 时，内存 inode 才能释放。

  PS：Unix V6++系统正常运行，你看到命令行提示符的时候，可执行文件 shell 已经不占用内存 inode 了。。。
- 每个网络链接 socket，占用一个内存 inode。

2、内存索引节点（内存 inode） Inode 结构

```
class Inode
{
    short i_dev;            /* 外存inode所在存储设备的设备号 */
    int i_number;           /* 外存inode区中的编号 */

    unsigned int i_flag;    /* 状态的标志位，定义见enum INodeFlag */
    int i_count;            /* 引用计数 */

    int i_lastr;            /* 最近读过的逻辑块号，用于判断是否需要预读 */
    static int rablock;     /* 预读块的物理块号 */

    unsigned int i_mode;    /* 文件工作方式信息 */
    int i_nlink;            /* 文件联结计数，即该文件在目录树中不同路径名的数量 */
    short i_uid;            /* 文件所有者的用户标识数 */
    short i_gid;            /* 文件所有者的组标识数 */
    int i_size;             /* 文件大小，字节为单位 */
    int i_addr[10];         /* 用于文件逻辑块好和物理块好转换的基本索引表 */
};
```

图 2 Inode

蓝色字段。i_dev 是 inode 所在的磁盘，i_number 是 inode 编号。

绿色字段是内存 inode 的使用状态。

- i_flag，下面详述；
- i_count，引用计数。不严格地说，同一个文件，open 一次 i_count++；close 一次 i_count--。i_count== 0，Inode 已无人使用，可以同步回磁盘 & 回归空闲状态。
- i_lastr，rablock； for 预读。上次读的逻辑块号 和 预读块的物理块号。

i_flags 中的标识。关键的是 ILOCK, IWANTED 这一对；还有脏标识 IUPD。

每个使用中的文件占用 1 个内存 inode，访问该文件的所有进程共享。进程执行系统调用前，ILOCK 置 1 上锁 inode；系统调用执行完毕，解锁。等待 Inode 解锁的进程置 1 IWANTED 标识。Inode 解锁时，唤醒它们。

此外，Inode 被访问，IACC 置 1。

```
17    /* i_flag中标志位 */
18    enum INodeFlag
19    {
20        ILOCK = 0x1,        /* 索引节点上锁 */
21        IUPD  = 0x2,        /* 内存inode被修改过，需要更新相应外存inode */
22        IACC  = 0x4,        /* 内存inode被访问过，需要修改最近一次访问时间 */
23        IMOUNT = 0x8,       /* 内存inode用于挂载子文件系统 */
24        IWANT = 0x10,       /* 有进程正在等待该内存inode被解锁，清ILOCK标志时，要唤醒这种进程 *
25        ITEXT = 0x20        /* 内存inode对应进程图像的正文段 */
26    };
27
```

图 3 i_flags

黑色字段，是 DiskInode 字段复本。是文件静态属性。

DiskInode 的 d_atime 和 d_mtime 是文件的时间戳。内存 inode 没有这 2 个字段。进程 close 文件的时候，将系统当前时刻 time 写入 d_atime 和 d_mtime。其余字段，d_* 对应于 Inode 中的同名字段 i_*。

```
class DiskInode
{
    public:
    unsigned int  d_mode;/* 状态的标志位，定义见enum INodeFlag */
    int   d_nlink;/* 文件联结计数，即该文件在目录树中不同路径名的数量 */

    short d_uid;/* 文件所有者的用户标识数 */
    short d_gid;/* 文件所有者的组标识数 */

    int d_size;/* 文件大小，字节为单位 */
    int d_addr[10];/* 用于文件逻辑块好和物理块好转换的基本索引表 */

    int d_atime;/* 最后访问时间 */
    int d_mtime;/* 最后修改时间 */
};
```

目录文件的 Inode：新建、删除文件时 IUPD 置 1；目录搜索时，IACC 不动；ls 列目录，IACC 置 1。

二、系统打开文件表 和 打开文件控制块

1、系统打开文件表

系统打开文件表 m_File 是 File 结构数组，有 100 个元素。

- 每成功执行一次 open 系统调用，分配 1 个 File 结构，f_flag 是 open 的第 2 个参数，mode。
- 每成功执行一次 creat 系统调用，分配 1 个 File 结构，f_flag 是 FWRITE。

很多次 open 可以针对同一个 Inode。这些进程使用不同的 File 结构访问同一个文件，用自己 File 结构中的 f_flag 和 f_offset，对文件的不同位置独立实施读写操作。f_flag 是文件访问的类型，f_offset 是读写操作在文件中的起始地址。

```
class OpenFileTable
{
public:
    static const int NFILE   = 100;   /* 打开文件控制块 File 结构的数量 */
public:
    File* FAlloc();                 // 分配一个空闲的 File 结构
    void CloseF(File* pFile);       // 释放对 File 结构的引用
public:
    File m_File[NFILE];             // 系统打开文件表
};
```

2、打开文件控制块

内存 Inode 登记的是与文件结构相关的信息。每个使用中的文件只有一个内存 Inode。

文件系统允许不同的进程以不同的方式打开同一个文件，各自独立地访问文件中相同或不相同区域中的数据。有的进程读，有的进程写。文件访问流是文件一次访问的全过程，开始于 open 或 create 系统调用，终结于 close 系统调用。负责管理文件访问流的是打开文件控制块 File 结构。

- f_flag。记录 open 系统调用的打开方式，读打开还是写打开。读打开的文件，FREAD 标识为 1；写打开的文件，FWRITE 标识为 1；读写打开的文件，2 个标识都有。
- f_offset 是读写指针，表示文件流的当前读写位置。read/write 系统调用从这个位置顺序向后访问文件。对磁盘上的普通文件，f_offset 是文件中的偏移量。
- f_inode，指向一个内存 inode，这是被访问的文件。一个普通的磁盘文件，一个 tty 的输入或输出缓存 或是一个网络套接字（socket）。
- f_count。多个进程可能会共用同一个 File 结构，用相同的读写指针访问文件。f_count 是计数器。父进程创建子进程，之后这 2 个进程会共用一个 File 结构，f_count++。

```
class File
{
public:
    enum FileFlags
    {
        FREAD = 0x1,            /* 读请求类型 */
        FWRITE = 0x2,           /* 写请求类型 */
        FPIPE = 0x4             /* 管道类型 */
    };
public:
    unsigned int  f_flag;       /* 对打开文件的读、写操作要求 */
    int           f_count;      /* 当前引用该文件控制块的进程数量 */
    Inode*        f_inode;      /* 指向打开文件的内存 Inode 指针 */
    int           f_offset;     /* 文件当前读写位置 */
};
```

三、进程的打开文件表

User 结构中，字段 u_ofiles 是进程的打开文件表。

```
OpenFiles u_ofiles;
```

打开文件表的定义：

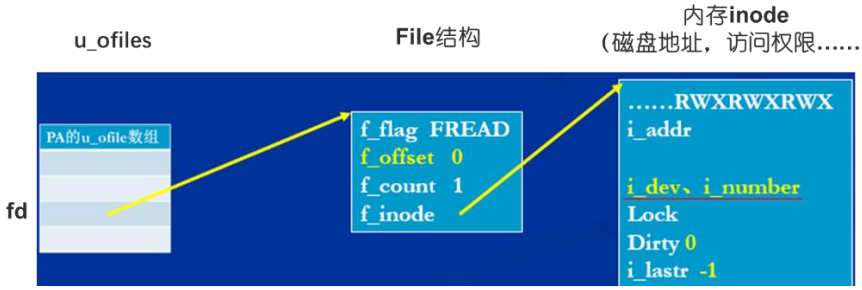
```
class OpenFiles
{
public:
    static const int NOFILES = 15;    /* 进程允许打开的最大文件数 */
public: ……
private:
    File *ProcessOpenFileTable[NOFILES];
};
```

ProcessOpenFileTable 数组，下标是打开文件描述符 fd，每个非空元素指向一个 File 结构。数组共有 15 个元素，进程可以同时访问最多 15 个文件。

0#文件，1#文件和 2#文件分别是进程的标准输入文件，标准输出文件和标准错误输出文件。默认标准输入文件中的字符是终端的键盘输入，写入标准输出文件的字符将显示在终端的屏幕上，写入标准错误输出文件的字符将以红色字体显示在终端的屏幕上。

四、打开文件结构，支持普通文件读写的内核数据结构

进程打开文件表，File 结构，内存 inode 组成打开文件结构。如图：



open/create 系统调用 建立打开文件结构，返回文件描述符 fd。read/write 系统调用利用 fd 使用打开文件结构，定位需要访问的文件数据。具体而言，查表 u_ofiles，定位 File 结构，

- 得到文件读写指针 f_offset；
- 文件的 inode。
  - 地址映射，需要使用混合索引结构，i_addr 是混合索引结构的树根。
  - i_size，文件长度。write 系统调用可能会增加文件长度、修正 i_size；read 系统调用需要这个字段判断 EOF。

五、超级块 g_spb

FileSystem.cpp 中定义的内核全局变量 g_spb 是磁盘超级块的复本。。

SuperBlock g_spb;

```
class SuperBlock
{   ……
public:
    int     s_isize;        /* 外存 Inode 区占用的盘块数 */
    int     s_fsize;        /* 盘块总数 */

    int     s_nfree;        /* 直接管理的空闲盘块数量 */
    int     s_free[100];    /* 直接管理的空闲盘块索引表 */

    int     s_ninode;       /* 直接管理的空闲外存 Inode 数量 */
    int     s_inode[100];   /* 直接管理的空闲外存 Inode 索引表 */

    int     s_flock;        /* 封锁空闲盘块索引表标志 */
    int     s_ilock;        /* 封锁空闲 Inode 表标志 */

    int     s_fmod;         /* 脏标记。卸载时，超级块写回磁盘 */
    int     s_ronly;        /* 本文件系统只能读出 */
    int     s_time;         /* 最近一次更新时间 */
    int     padding[47];    /* 填充使 SuperBlock 块大小等于 1024 字节，占据 2 个扇区 */
};
```

5.1 系统初始化时，启动磁盘，将主硬盘的 SuperBlock 读入 g_spb。

```
60 void FileSystem::LoadSuperBlock()
61 {
62     User& u = Kernel::Instance().GetUser();
63     BufferManager& bufMgr = Kernel::Instance().GetBufferManager();
64     Buf* pBuf;
65
66     for (int i = 0; i < 2; i++)
67     {
68         int* p = (int *)&g_spb + i * 128;
69
70         pBuf = bufMgr.Bread(DeviceManager::ROOTDEV, FileSystem::SUPER_BLOCK_SECTOR_NUMBER + i);
71
72         Utility::DWordCopy((int *)pBuf->b_addr, p, 128);
73
74         bufMgr.Brelse(pBuf);
75     }
76     if (User::NOERROR != u.u_error)
77     {
78         Utility::Panic("Load SuperBlock Error....!\n");
79     }
80
81     this->m_Mount[0].m_dev = DeviceManager::ROOTDEV;
82     this->m_Mount[0].m_spb = &g_spb;
83
84     g_spb.s_flock = 0;
85     g_spb.s_ilock = 0;
86     g_spb.s_ronly = 0;
87     g_spb.s_time = Time::time;
88 }
```

FileSystem::SUPER_BLOCK_SECTOR_NUMBER 值是 200，是 SuperBlock 在磁盘上的起始扇区号。代码的核心是 Line66~Line75 的循环。这里循环 2 次，依次将 200#扇区，201#扇区同步读入内存变量 g_spb。

5.2 系统正常运行时，文件系统使用 g_spb 中的空闲 inode 栈和空闲盘块号栈分配磁盘存储资源。具体而言，新建文件和子目录时，需要分配空闲 inode；删除文件和子目录时，需要回收空闲 inode。写文件如果导致文件长度增加，需要为新数据分配空闲盘块（磁盘数据块）；删除文件时，回收分配给这个文件的所有数据块。

这些操作会改变超级块的内容，置 1 g_spb 的脏标记 s_fmod。关机时，脏的超级块需要写回磁盘。

5.3 安全关机

36#系统调用执行安全关机操作，同步内存中尚未写回的文件系统更新。

shutdown 命令，应用程序 shutdown.c :

```
**************************************************************************
#include <stdio.h>
#include <sys.h>

int main1(int argc, char* argv[])
{
    if ( -1 == syncFileSystem() )
        printf("Error Happens During File System Update Progress...:(\n");
    else
        printf("File System Successfully Updated!\nYou can close Bochs now...\n");

    return 1;
```

```
}
```

36#系统调用的钩子函数：**********

```
int syncFileSystem()
{
	int res;
	__asm__ volatile ( "int $0x80":"=a"(res):"a"(36) );
	if ( res >= 0 )
		return res;
	return -1;
}
```

36#系统调用的入口函数：**********

```
int SystemCall::Sys_Sync()
{
	Kernel::Instance().GetFileSystem().Update();

	return 0; /* GCC likes it ! */
}
```

用来同步文件系统修改的 Update( )函数：**********

没来得及注释。。。

做 3 件事：脏 inode 刷回磁盘，脏数据块刷回磁盘，SuperBlock 刷回磁盘

六、基础函数

1、IGet(dev,number) 和 IPut(dev,number);

IGet 函数，锁住分配给<dev，number>的内存 Inode，同步读入 DiskInode ，i_count++。

2、NFLock( ) 和 NFrele( );

Inode 上锁和解锁。代码中的 this 指针指向函数处理的 Inode。

read、write 系统调用执行期间，fd 引用的内存 Inode 上锁。执行完毕，解锁。

read，write 系统调用执行期间，V6++上锁 inode

```
635 void Inode::NFlock()
636 {
637     User& u = Kernel::Instance().GetUser();
638
639     while( this->i_flag & Inode::ILOCK )
640     {
641         this->i_flag |= Inode::IWANT;
642         u.u_procp->Sleep((unsigned long)this, ProcessManager::PRIBIO);
643     }
644     this->i_flag |= Inode::ILOCK;
645 }
646
```

上锁内存inode
如果其它进程正在使用这个Inode，置IWANT标记，入睡等待

```
622
623 void Inode::NFrele()
624 {
625     /* 解锁pipe或Inode,并且唤醒相应进程 */
626     this->i_flag &= ~Inode::ILOCK;
627
628     if (this->i_flag & Inode::IWANT)
629     {
630         this->i_flag &= ~Inode::IWANT;
631         Kernel::Instance().GetProcessManager().WakeUpAll((unsigned long)this);
632     }
633 }
634
```

系统调用执行完毕解锁 Inode
IWANT为1，唤醒等待使用这个Inode的进程

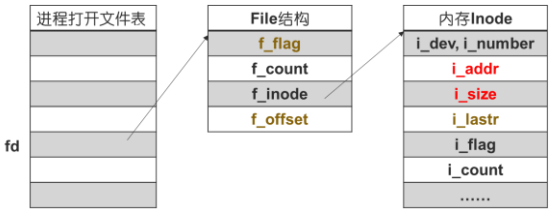
操作系统　电信学院计算机系 邓蓉　16

七、Open 系统调用的工作流程（要求熟练掌握）

第一步：线性目录搜索，找到目标文件的 DiskInode 号。（Namel 函数）
第二步：在内存 Inode 池中搜索目标 DiskInode。如果不命中，分配内存 Inode，启动磁盘读入 DiskInode。（Namel 函数→IGet 函数）
第三步：检测进程对目标文件的访问权限。（Access 函数）
第四步：在系统打开文件表中分配一个 File 结构，在进程打开文件表中分配一个空闲表项（下标是 fd），建立打开文件结构。（FAlloc 函数）
第五步：返回文件描述符 fd。

八、普通文件的读写操作：打开文件结构的使用，读写开销和 预读开销

访问普通文件之前，需要执行 open 或 creat 系统调用，为这个文件建立打开文件结构。如图：



打开文件结构建立完成之后，进程便可以通过 read/write 系统调用读写文件中的数据。读写操作的细节请复习巩固前面课程内容，结合今天的 PPT。现在我们考察文件访问的 IO 开销。

建立打开文件结构的主要开销 = 目录搜索的开销 + 将目标 DiskInode 读入内存 Inode（IO 一次）。

打开文件结构建立完成后，目标文件：

- 混合索引表的树根 d_addr 在内存里，是内存 Inode 中的 i_addr 字段。
- 其余索引块和数据块在磁盘上，访问时需要执行 IO 操作。

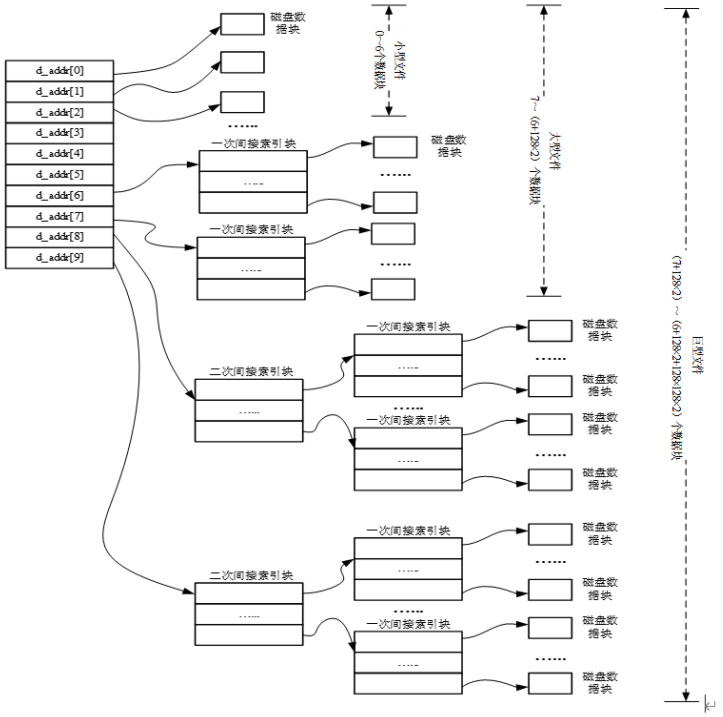


图 7.17：UNIX V6++文件索引结构

例 1：不考虑缓存。读 500#字节，IO 1 次，同步读入 i_addr[0]管理的 0#字符块。
答：读 5000#字节，IO 2 次，同步读入 i_addr[6]管理的一次间接索引块；之后，同步读入该索引块 3#字管理的 9#字符块。
读 n#字节（n>131K）。需要 IO 3 次。依次同步读入二次索引块、一次索引块和目标数据块。

例 2：Bmap( )地址映射

（1） 当前块是 3#逻辑块，当前块的物理块号？预读块的物理块号？预读，是否合算？
（2） 当前块是 5#逻辑块，当前块的物理块号？预读块的物理块号？预读，是否合算？
（3） 当前块是 100#逻辑块，当前块的物理块号？预读块的物理块号？预读，是否合算？
（4） 当前块是 133#逻辑块，当前块的物理块号？预读块的物理块号？预读，是否合算？

答：

（1） 当前块是 3#逻辑块，当前块的物理块号是 i_addr[3]。预读块的物理块号 i_addr[4]。预读不会引发索引块 IO，开销近似为 0，实施。rablkno= i_addr[4]。
（2） 当前块是 5#逻辑块，当前块的物理块号是 i_addr[5]。预读块的物理块号在 0#索引块里，是 0#字。计算预读块的物理块号很可能会引发索引块 IO、开销很大，不合算，放弃预读，rablkno=0。

(3) 当前块是 100#逻辑块，当前块的物理块号，0#索引块的 95#字。预读块的物理块号，0#索引块的 96#字。索引块已经在缓存池中。计算预读块的物理块号不会引发额外的索引块 IO、开销近似为 0，实施。读出 0#索引块的 96#字→rablkno 。

(4) 当前块是 133#逻辑块，当前块的物理块号是 0#索引块的 127#字。预读块的物理块号是 1#索引块的 0#字。预读不合算，rablkno=0，放弃。