日曜研究室

技術的な観点から日常を綴ります

[xv6 #66] Chapter 5 – File system – File descriptor layer

テキストの74ページ

本文
Unixのインターフェイスの一つのクールな側面は、多くのリソース（コンソールのようなデバイスやパイプやもちろん実際のファイルも）をファイルとして表現しているところである。
ファイルディスクリプタの層は、この特徴を達成するための層である。

xv6は、第0章で見たように、各プロセスが開いてるファイル（ファイルディスクリプタ）のテーブルを各プロセスごとに与える。
開いているそれぞれのファイルは、inodeやパイプ、それにI/Oオフセットを加え、それらをラップするfile構造体として表現される。
1回のopenシステムコールの呼び出しごとに、新しいファイル（新しいfile構造体）を作成する。
複数のプロセスが同じファイルをお互い無関係に開いた場合、それぞれのインスタンスは違うI/Oオフセットを持つだろう。
一方、一つの開いているファイル（file構造体）は、あるプロセスのファイルテーブルに複数回現れる可能性があり、また、複数のプロセスに渡っても同じことが起きる可能性が当然ある。
あるプロセスがopenシステムコールを呼び出したら、dupシステムコールを使ってエイリアスが作成されるか、もしくはforkシステムコールを使って子プロセスに共有されるということが起こるだろう。
リファレンスカウントrefによって、あるファイルが個別に開かれている回数が追跡される。
一つのファイルは、読み込みもしくは書き込み、またはその両方のために開くことが出来る。
readableフィールドとwritableフィールドはこれを追跡する。

file.h

```
01 struct file {
02   enum { FD_NONE, FD_PIPE, FD_INODE } type;
03   int ref; // reference count
04   char readable;
05   char writable;
06   struct pipe *pipe;
```

```
07   struct inode *ip;
08   uint off;
09 };
10
11
12 // in-core file system types
13
14 struct inode {
15   uint dev;           // Device number
16   uint inum;          // Inode number
17   int ref;            // Reference count
18   int flags;          // I_BUSY, I_VALID
19
20   short type;         // copy of disk inode
21   short major;
22   short minor;
23   short nlink;
24   uint size;
25   uint addrs[NDIRECT+1];
26 };
27
28 #define I_BUSY 0x1
29 #define I_VALID 0x2
30
31 // device implementations
32
33 struct devsw {
34   int (*read)(struct inode*, char*, int);
35   int (*write)(struct inode*, char*, int);
36 };
37
38 extern struct devsw devsw[];
39
40 #define CONSOLE 1
```

システム内で開かれている全てのファイルは、グローバルなファイルテーブルであるftableに保持される。
そのファイルテーブルはファイルを割り当てるための関数（filealloc）や、参照の複製を作成するための関数（filedup）や、参照を解放するための関数（fileclose）や、データを読み書きするための関数（fileread, filewrite）を持つ。

file.c

```
001 #include "types.h"
002 #include "defs.h"
003 #include "param.h"
004 #include "fs.h"
005 #include "file.h"
006 #include "spinlock.h"
007
008 struct devsw devsw[NDEV];
009 struct {
010   struct spinlock lock;
011   struct file file[NFILE];
012 } ftable;
013
014 void
015 fileinit(void)
016 {
```

```
017   initlock(&ftable.lock, "ftable");
018 }
019
020 // Allocate a file structure.
021 struct file*
022 filealloc(void)
023 {
024   struct file *f;
025
026   acquire(&ftable.lock);
027   for(f = ftable.file; f < ftable.file + NFILE; f++){
028     if(f->ref == 0){
029       f->ref = 1;
030       release(&ftable.lock);
031       return f;
032     }
033   }
034   release(&ftable.lock);
035   return 0;
036 }
037
038 // Increment ref count for file f.
039 struct file*
040 filedup(struct file *f)
041 {
042   acquire(&ftable.lock);
043   if(f->ref < 1)
044     panic("filedup");
045   f->ref++;
046   release(&ftable.lock);
047   return f;
048 }
049
050 // Close file f.  (Decrement ref count, close when reaches 0.)
051 void
052 fileclose(struct file *f)
053 {
054   struct file ff;
055
056   acquire(&ftable.lock);
057   if(f->ref < 1)
058     panic("fileclose");
059   if(--f->ref > 0){
060     release(&ftable.lock);
061     return;
062   }
063   ff = *f;
064   f->ref = 0;
065   f->type = FD_NONE;
066   release(&ftable.lock);
067
068   if(ff.type == FD_PIPE)
069     pipeclose(ff.pipe, ff.writable);
070   else if(ff.type == FD_INODE){
071     begin_trans();
072     iput(ff.ip);
073     commit_trans();
074   }
075 }
076
077 // Get metadata about file f.
078 int
079 filestat(struct file *f, struct stat *st)
080 {
```

```
081   if(f->type == FD_INODE){
082     ilock(f->ip);
083     stati(f->ip, st);
084     iunlock(f->ip);
085     return 0;
086   }
087   return -1;
088 }
089
090 // Read from file f.  Addr is kernel address.
091 int
092 fileread(struct file *f, char *addr, int n)
093 {
094   int r;
095
096   if(f->readable == 0)
097     return -1;
098   if(f->type == FD_PIPE)
099     return piperead(f->pipe, addr, n);
100   if(f->type == FD_INODE){
101     ilock(f->ip);
102     if((r = readi(f->ip, addr, f->off, n)) > 0)
103       f->off += r;
104     iunlock(f->ip);
105     return r;
106   }
107   panic("fileread");
108 }
109
110 //PAGEBREAK!
111 // Write to file f.  Addr is kernel address.
112 int
113 filewrite(struct file *f, char *addr, int n)
114 {
115   int r;
116
117   if(f->writable == 0)
118     return -1;
119   if(f->type == FD_PIPE)
120     return pipewrite(f->pipe, addr, n);
121   if(f->type == FD_INODE){
122     // write a few blocks at a time to avoid exceeding
123     // the maximum log transaction size, including
124     // i-node, indirect block, allocation blocks,
125     // and 2 blocks of slop for non-aligned writes.
126     // this really belongs lower down, since writei()
127     // might be writing a device like the console.
128     int max = ((LOGSIZE-1-1-2) / 2) * 512;
129     int i = 0;
130     while(i < n){
131       int n1 = n - i;
132       if(n1 > max)
133         n1 = max;
134
135       begin_trans();
136       ilock(f->ip);
137       if ((r = writei(f->ip, addr + i, f->off, n1)) > 0)
138         f->off += r;
139       iunlock(f->ip);
140       commit_trans();
141
142       if(r < 0)
143         break;
144       if(r != n1)
```

```
145           panic("short filewrite");
146         i += r;
147       }
148       return i == n ? n : -1;
149     }
150     panic("filewrite");
151   }
```

最初の3つは、おなじみの手順に従う。
filealloc関数は、参照されていないファイル（f->ref == 0）を捜すためにファイルテーブルをスキャンし、新しい参照を返す。
filedup関数は、参照カウントをインクリメントする。
fileclose関数は、参照カウントをデクリメントする。
ファイルの参照カウントがゼロに達したら、fileclose関数は、file構造体の裏にあるパイプやinodeを解放する。
どちらを解放するかはfile構造体のtypeフィールドの値に従う。

filestat, fileread, filewrite関数は、ファイルに対するstat, read, writeの実装である。
filestat関数は、inodeに対してのみ機能し、stati関数を呼ぶ。
fileread, filewrite関数は、ファイルが開かれたときのモードにその操作が許可されているかチェックし、それからパイプやinodeの実装を呼び出す。
もし、ファイルがinodeを表現していたら、filereadとfilewriteは、I/Oオフセットの値を、操作のためのオフセットとして使い、そのオフセット値を読み書きした分だけ進める。
パイプにはオフセットの概念はない。
inodeの関数は、その呼び出し側によるロックの制御を要求することを思い出せ。（filestat関数などのilock, iunlockの呼び出し部分）
inodeをロックすることは、読み込みや書き込みのオフセット値の更新がアトミックに行われるという便利な副作用ももたらす。
なので、同じファイルに対して同時に複数の書き込みが発生しても、お互いのデータが上書きされることはない。
が、最終的にはその書き込みはごちゃ混ぜになるだろう。

感想

ファイルディスクリプタの実装についてです。

ファイルディスクリプタでは、パイプとinodeが統合されると書かれています。
inodeの層で実際のファイルとデバイスが統合されるので、最終的に実際のファイル、デバイス、パイプが”ファイル”として扱われることになります。

コード自体は、今まで見てきたコードと同じような感じで、注意深く読まないと勘違いするような部分は特になさげです。
それだからか、この節は一つ一つの関数はサラッと説明されています。

タイトルにも”Code: “が入ってないですね。

次の節で、xv6自体の説明は終わりです。
（その後、Real worldの節とExercisesの節がありますけどね）

さらに付録A, Bがあるのですが、これは以前読むかどうかは未定と書いた覚えがあります。
ザッと目を通したところ、教育用OSの枠内にとどまらない実際の世界の話も書かれてるみたいで、
なかなか面白そうなので読んでみたいと思います。

カテゴリー: 技術 | タグ: xv6 | 投稿日: 2012/4/23 月曜日 [http://peta.okechan.net/blog/archives/1666] |