日曜研究室

技術的な観点から日常を綴ります

[xv6 #67] Chapter 5 – File system – Code: System calls

テキストの74ページ

本文

低レイヤが提供する関数を使って実装される多くのシステムコールは些細なものである。（sysfile.cを見よ）
その中で注目に値するシステムコールがいくつかある。

sys_linkとsys_unlinkシステムコールは、ディレクトリを変更し、inodeへの参照を生成したり消去したりする。
それらはトランザクションの便利さを示すいい例である。
sys_linkシステムコールは、その引数、oldとnewという文字列変数を取り出すところからはじめる。
oldは存在し、かつディレクトリではないと仮定し、sys_linkはそのip->nlinkをインクリメントする。
それからsys_linkは、newの親ディレクトリと最後のパス要素を探すためにnameiparent関数を呼び、そしてoldのinodeを参照する新しいディレクトリエントリを作成する。
newの親ディレクトリは存在している必要があり、oldのinodeと同じデバイス上になければならない。
inode番号は、一つのディスク上でのみ一意だからである。
以上のルールに沿わないようなエラーが起きた場合、sys_linkは途中まで行った操作を元に戻して、ip->nlinkをデクリメントしなければならない。

sys_linkは複数のディスクブロックを更新する必要があるので、トランザクションはその実装を単純化するが、どのような順番でブロックが更新されるかについては我々が心配する必要はない。
最終的には成功するか失敗するかのどちらかである。
例えば、トランザクション無しだと、リンクを作成するまえにip->nlinkを更新するときに、一時的にファイルシステムが危険な状態になり、その間にクラッシュが起きると、大破壊がもたらされるだろう。
トランザクションを使えば、このような事について心配する必要はなくなる。

sys_linkは、既存のinodeのための新しい名前を作成する。
create関数は、新しいinodeのための新しい名前を作成する。
create関数は、ファイル生成に関わる3つのシステムコールの処理を一般化したものである。
openシステムコールがO_CREATEフラグとともに呼ばれると、通常の新しいファイルを生成し、mkdirシステムコールは新しいディレクトリを作成し、mkdevシステムコールは新しいデバイスファイルを作成する。
sys_linkのように、create関数は、親ディレクトリのinodeを得るためにnameiparent関数を呼ぶことから初める。
それからdirlookup関数を使って、名前がすでに存在していないかチェックする。
名前がすでに存在していた場合、create関数の振る舞いはどのシステムコールに呼ばれたかに依存する。
名前がすでに存在しているという事実は、openでは、mkdirやmkdevとは違った意味を持つ。
createがopenのために呼ばれ（type == T_FILE）、名前がすでに存在し、それが通常のファイルだった場合、openは成功として扱い、createもそれに従う。
そうでなければエラーとなる。
名前が存在しない場合、createはiallocを使って新しいinodeを割り当てる。
新しいinodeがディレクトリである場合、createは”.”と”..”でそのエントリを初期化する。
最後に、そのデータが正常に初期化されたら、createはその親ディレクトリにそのディレクトリへのリンクを作成する。
createは、sys_linkのように、同時に2つのinode（ipとdp）のロックを保持する。
inode ipは新たに割り当てられたものなので、デッドロックは起こりえない。
まずipをロックし、それからdpをロックしようとするような他のプロセスは存在しない。

createを使うと、sys_open, sys_mkdir, sys_mknodの実装が簡単になる。
sys_openがその中では一番複雑である。
なぜなら新しいファイルを生成することは、それが出来ることの一部に過ぎないからである。
openにO_CREATEフラグが渡された場合、createを呼ぶ。
それ以外の場合は、nameiを呼ぶ。
createはロック済みのinodeを返すが、nameiはそうじゃないので、sys_openはそのinodeを自分自身でロックする。
これは、対象のinodeがディレクトリかつ読み込み専用で開かれてるだけかどうかをチェックするにはよい箇所である。
いずれにしても、inodeが得られたと仮定し、sys_openはファイルとファイルディスクリプタを割り当て、それからファイルのメタデータを設定する。
このファイルは、現在のプロセスのテーブルにしか存在しないので、初期化途中のファイルに他のプロセスがアクセスすることはないということに注意せよ。

第4章では、ファイルシステムの説明の前に、パイプの実装について説明した。
sys_pipe関数は、パイプの組を生成する方法を提供する事によって、ファイルシステムの実装への橋渡しをしている。
sys_pipeの引数は、2つの整数の領域を指すポインタであり、新しい2つのファイルディスクリプタを

記録する場所になる。
sys_pipeはパイプを割り当て、ファイルディスクリプタにそのパイプを設定する。

sysfile.c

```
001 #include "types.h"
002 #include "defs.h"
003 #include "param.h"
004 #include "stat.h"
005 #include "mmu.h"
006 #include "proc.h"
007 #include "fs.h"
008 #include "file.h"
009 #include "fcntl.h"
010
011 // Fetch the nth word-sized system call argument as a file
    descriptor
012 // and return both the descriptor and the corresponding struct file.
013 static int
014 argfd(int n, int *pfd, struct file **pf)
015 {
016   int fd;
017   struct file *f;
018
019   if(argint(n, &fd) < 0)
020     return -1;
021   if(fd < 0 || fd >= NOFILE || (f=proc->ofile[fd]) == 0)
022     return -1;
023   if(pfd)
024     *pfd = fd;
025   if(pf)
026     *pf = f;
027   return 0;
028 }
029
030 // Allocate a file descriptor for the given file.
031 // Takes over file reference from caller on success.
032 static int
033 fdalloc(struct file *f)
034 {
035   int fd;
036
037   for(fd = 0; fd < NOFILE; fd++){
038     if(proc->ofile[fd] == 0){
039       proc->ofile[fd] = f;
040       return fd;
041     }
042   }
043   return -1;
044 }
045
046 int
047 sys_dup(void)
048 {
049   struct file *f;
050   int fd;
051
052   if(argfd(0, 0, &f) < 0)
053     return -1;
054   if((fd=fdalloc(f)) < 0)
055     return -1;
```

```
056   filedup(f);
057   return fd;
058 }
059
060 int
061 sys_read(void)
062 {
063   struct file *f;
064   int n;
065   char *p;
066
067   if(argfd(0, 0, &f) < 0 || argint(2, &n) < 0 || argptr(1, &p, n) <
    0)
068     return -1;
069   return fileread(f, p, n);
070 }
071
072 int
073 sys_write(void)
074 {
075   struct file *f;
076   int n;
077   char *p;
078
079   if(argfd(0, 0, &f) < 0 || argint(2, &n) < 0 || argptr(1, &p, n) <
    0)
080     return -1;
081   return filewrite(f, p, n);
082 }
083
084 int
085 sys_close(void)
086 {
087   int fd;
088   struct file *f;
089
090   if(argfd(0, &fd, &f) < 0)
091     return -1;
092   proc->ofile[fd] = 0;
093   fileclose(f);
094   return 0;
095 }
096
097 int
098 sys_fstat(void)
099 {
100   struct file *f;
101   struct stat *st;
102
103   if(argfd(0, 0, &f) < 0 || argptr(1, (void*)&st, sizeof(*st)) < 0)
104     return -1;
105   return filestat(f, st);
106 }
107
108 // Create the path new as a link to the same inode as old.
109 int
110 sys_link(void)
111 {
112   char name[DIRSIZ], *new, *old;
113   struct inode *dp, *ip;
114
115   if(argstr(0, &old) < 0 || argstr(1, &new) < 0)
116     return -1;
117   if((ip = namei(old)) == 0)
```

```
118     return -1;
119
120   begin_trans();
121
122   ilock(ip);
123   if(ip->type == T_DIR){
124     iunlockput(ip);
125     commit_trans();
126     return -1;
127   }
128
129   ip->nlink++;
130   iupdate(ip);
131   iunlock(ip);
132
133   if((dp = nameiparent(new, name)) == 0)
134     goto bad;
135   ilock(dp);
136   if(dp->dev != ip->dev || dirlink(dp, name, ip->inum) < 0){
137     iunlockput(dp);
138     goto bad;
139   }
140   iunlockput(dp);
141   iput(ip);
142
143   commit_trans();
144
145   return 0;
146
147 bad:
148   ilock(ip);
149   ip->nlink--;
150   iupdate(ip);
151   iunlockput(ip);
152   commit_trans();
153   return -1;
154 }
155
156 // Is the directory dp empty except for "." and ".." ?
157 static int
158 isdirempty(struct inode *dp)
159 {
160   int off;
161   struct dirent de;
162
163   for(off=2*sizeof(de); off<dp->size; off+=sizeof(de)){
164     if(readi(dp, (char*)&de, off, sizeof(de)) != sizeof(de))
165       panic("isdirempty: readi");
166     if(de.inum != 0)
167       return 0;
168   }
169   return 1;
170 }
171
172 //PAGEBREAK!
173 int
174 sys_unlink(void)
175 {
176   struct inode *ip, *dp;
177   struct dirent de;
178   char name[DIRSIZ], *path;
179   uint off;
180
181   if(argstr(0, &path) < 0)
```

```
182     return -1;
183   if((dp = nameiparent(path, name)) == 0)
184     return -1;
185
186   begin_trans();
187
188   ilock(dp);
189
190   // Cannot unlink "." or "..".
191   if(namecmp(name, ".") == 0 || namecmp(name, "..") == 0)
192     goto bad;
193
194   if((ip = dirlookup(dp, name, &off)) == 0)
195     goto bad;
196   ilock(ip);
197
198   if(ip->nlink < 1)
199     panic("unlink: nlink < 1");
200   if(ip->type == T_DIR && !isdirempty(ip)){
201     iunlockput(ip);
202     goto bad;
203   }
204
205   memset(&de, 0, sizeof(de));
206   if(writei(dp, (char*)&de, off, sizeof(de)) != sizeof(de))
207     panic("unlink: writei");
208   if(ip->type == T_DIR){
209     dp->nlink--;
210     iupdate(dp);
211   }
212   iunlockput(dp);
213
214   ip->nlink--;
215   iupdate(ip);
216   iunlockput(ip);
217
218   commit_trans();
219
220   return 0;
221
222 bad:
223   iunlockput(dp);
224   commit_trans();
225   return -1;
226 }
227
228 static struct inode*
229 create(char *path, short type, short major, short minor)
230 {
231   uint off;
232   struct inode *ip, *dp;
233   char name[DIRSIZ];
234
235   if((dp = nameiparent(path, name)) == 0)
236     return 0;
237   ilock(dp);
238
239   if((ip = dirlookup(dp, name, &off)) != 0){
240     iunlockput(dp);
241     ilock(ip);
242     if(type == T_FILE && ip->type == T_FILE)
243       return ip;
244     iunlockput(ip);
245     return 0;
```

```
246   }
247
248   if((ip = ialloc(dp->dev, type)) == 0)
249     panic("create: ialloc");
250
251   ilock(ip);
252   ip->major = major;
253   ip->minor = minor;
254   ip->nlink = 1;
255   iupdate(ip);
256
257   if(type == T_DIR){  // Create . and .. entries.
258     dp->nlink++;  // for ".."
259     iupdate(dp);
260     // No ip->nlink++ for ".": avoid cyclic ref count.
261     if(dirlink(ip, ".", ip->inum) < 0 || dirlink(ip, "..", dp->inum)
< 0)
262       panic("create dots");
263   }
264
265   if(dirlink(dp, name, ip->inum) < 0)
266     panic("create: dirlink");
267
268   iunlockput(dp);
269
270   return ip;
271 }
272
273 int
274 sys_open(void)
275 {
276   char *path;
277   int fd, omode;
278   struct file *f;
279   struct inode *ip;
280
281   if(argstr(0, &path) < 0 || argint(1, &omode) < 0)
282     return -1;
283   if(omode & O_CREATE){
284     begin_trans();
285     ip = create(path, T_FILE, 0, 0);
286     commit_trans();
287     if(ip == 0)
288       return -1;
289   } else {
290     if((ip = namei(path)) == 0)
291       return -1;
292     ilock(ip);
293     if(ip->type == T_DIR && omode != O_RDONLY){
294       iunlockput(ip);
295       return -1;
296     }
297   }
298
299   if((f = filealloc()) == 0 || (fd = fdalloc(f)) < 0){
300     if(f)
301       fileclose(f);
302     iunlockput(ip);
303     return -1;
304   }
305   iunlock(ip);
306
307   f->type = FD_INODE;
308   f->ip = ip;
```

```
309   f->off = 0;
310   f->readable = !(omode & O_WRONLY);
311   f->writable = (omode & O_WRONLY) || (omode & O_RDWR);
312   return fd;
313 }
314
315 int
316 sys_mkdir(void)
317 {
318   char *path;
319   struct inode *ip;
320
321   begin_trans();
322   if(argstr(0, &path) < 0 || (ip = create(path, T_DIR, 0, 0)) == 0){
323     commit_trans();
324     return -1;
325   }
326   iunlockput(ip);
327   commit_trans();
328   return 0;
329 }
330
331 int
332 sys_mknod(void)
333 {
334   struct inode *ip;
335   char *path;
336   int len;
337   int major, minor;
338
339   begin_trans();
340   if((len=argstr(0, &path)) < 0 ||
341      argint(1, &major) < 0 ||
342      argint(2, &minor) < 0 ||
343      (ip = create(path, T_DEV, major, minor)) == 0){
344     commit_trans();
345     return -1;
346   }
347   iunlockput(ip);
348   commit_trans();
349   return 0;
350 }
351
352 int
353 sys_chdir(void)
354 {
355   char *path;
356   struct inode *ip;
357
358   if(argstr(0, &path) < 0 || (ip = namei(path)) == 0)
359     return -1;
360   ilock(ip);
361   if(ip->type != T_DIR){
362     iunlockput(ip);
363     return -1;
364   }
365   iunlock(ip);
366   iput(proc->cwd);
367   proc->cwd = ip;
368   return 0;
369 }
370
371 int
372 sys_exec(void)
```

```
373 {
374   char *path, *argv[MAXARG];
375   int i;
376   uint uargv, uarg;
377
378   if(argstr(0, &path) < 0 || argint(1, (int*)&uargv) < 0){
379     return -1;
380   }
381   memset(argv, 0, sizeof(argv));
382   for(i=0;; i++){
383     if(i >= NELEM(argv))
384       return -1;
385     if(fetchint(proc, uargv+4*i, (int*)&uarg) < 0)
386       return -1;
387     if(uarg == 0){
388       argv[i] = 0;
389       break;
390     }
391     if(fetchstr(proc, uarg, &argv[i]) < 0)
392       return -1;
393   }
394   return exec(path, argv);
395 }
396
397 int
398 sys_pipe(void)
399 {
400   int *fd;
401   struct file *rf, *wf;
402   int fd0, fd1;
403
404   if(argptr(0, (void*)&fd, 2*sizeof(fd[0])) < 0)
405     return -1;
406   if(pipealloc(&rf, &wf) < 0)
407     return -1;
408   fd0 = -1;
409   if((fd0 = fdalloc(rf)) < 0 || (fd1 = fdalloc(wf)) < 0){
410     if(fd0 >= 0)
411       proc->ofile[fd0] = 0;
412     fileclose(rf);
413     fileclose(wf);
414     return -1;
415   }
416   fd[0] = fd0;
417   fd[1] = fd1;
418   return 0;
419 }
```

感想

ファイルシステム関連のシステムコールの実装についてです。

原文でも明記されてなかったりするので、システムコールという名称と関数という名称の使い分けが曖昧な部分があります。
まぁ関数のほうがより一般的な概念なのでどちらも関数と言っておけば間違いないと思いますが、システムコールに関しては、SYSCALLマクロで生成された（例えば）openのようなものだけをそう呼ぶのか、その実体である（例えば）sys_openまで含めるのかどうかが微妙なところです。

カテゴリー: 技術 | タグ: xv6 | 投稿日: 2012/4/25 水曜日 [http://peta.okechan.net/blog/archives/1673] |