日曜研究室

技術的な観点から日常を綴ります

[xv6 #5] Chapter 0 – Operating system interfaces – Code: File system

テキストの14〜15ページ

本文

xv6は、データファイル（解釈されていない生のバイト列）と、ディレクトリ（他のデータファイルやディレクトリを含む）を提供する。
xv6はディレクトリを、特別な種類のファイルとして実装している。
ディレクトリは、rootと呼ばれる特殊なディレクトリから始まるツリーの中に配置される。
/a/b/c のようなパスは、ルートディレクトリ"/"の中にあるaというディレクトリの中にあるbというディレクトリの中にあるcと名付けられたファイル（もしくはディレクトリ）を指し示す。
"/"から始まってないパスは、そのプロセスの現在のディレクトリ(current directory)を起点に相対的に評価される。
現在のディレクトリは、chdirシステムコールで変更出来る。
以下の2つのコードは同じファイルを開く。

```
1 chdir("/a");
2 chdir("b");
3 open("c", O_RDONLY);
```

```
1 open("/a/b/c", O_RDONLY);
```

最初のコードは、プロセスの現在のディレクトリを"/a/b"に変更する。
2つめのコードは、プロセスの現在のディレクトリを変更しない。

新しいファイルやディレクトリを作るためのシステムコールがいくつかある。
mkdirはディレクトリを作成する。
openをO_CREATEフラグで呼ぶと新しいファイルを作成する。
mknodで新しいデバイスファイルを作成する。
以上3つ全てについて次で例示する。

```
1 mkdir("/dir");
```

```
2 fd = open("/dir/file", O_CREATE|O_WRONLY);
3 close(fd);
4 mknod("/console", 1, 1);
```

mknodはファイルシステムの中にファイルを作るが、そのファイルは中身を持たない。
しかしながら、そのファイルのメタデータにはそれがデバイスファイルであるという情報がマークされ、カーネルデバイスを一意に識別するためのメジャーそしてマイナーデバイス番号（mknodの引数の後ろ二つに対応）が記録される。
プロセスが後でそのファイルを開いたとき、カーネルはreadやwriteのシステムコールを、ファイルシステムへreadしたりwriteしたりする代わりに、カーネルデバイスの実装へすり替える。

fstatは、ファイルディスクリプタが指し示しているオブジェクトについての情報を取り出す。
stat.hに定義されているstat構造体で結果が返ってくる。
（ちなみに以下はstat.hの一部ではなく全てです）

```
01 #define T_DIR  1   // Directory
02 #define T_FILE 2   // File
03 #define T_DEV  3   // Special device
04
05 struct stat {
06   short type;  // Type of file
07   int dev;     // Device number
08   uint ino;    // Inode number on device
09   short nlink; // Number of links to file
10   uint size;   // Size of file in bytes
11 };
```

ファイル名は、ファイル自体とは別になっている。（stat構造体にはnameなんてメンバはない）
1つのファイル（inode）は複数の名前（link）を持つことが出来る。
linkシステムコールは、既存のファイルなどと同じinodeを指し示すようなファイルシステム上の別名を作成する。
次のコードは、aというファイルを作成し、bという別名をつける。

```
1 open("a", O_CREATE|O_WRONLY);
2 link("a", "b");
```

aに対する読み書きは、bに対する読み書きと同じである。
どのinodeも一意のinode番号で識別される。
上のコードが実行されたあと、fstatの結果を調べるとaとbが同じ内容を参照してるということが分かる。
つまり両方とも同じinode番号（ino）が返ってくるはずである。
そしてnlinkには2が設定されているはずである。

unlinkシステムコールはファイルシステムから名前を取り除く。
ファイルのinodeとディスクスペースが解放されるのは、そのファイルへのリンク数がゼロになったとき、かつそのファイルを参照するファイルディスクリプタがないときだけである。
なので上記のコードの最後に

```
1 unlink("a");
```

を追加しても、bとしてアクセス可能なinodeとその内容は消えない。

```
1 fd = open("/tmp/xyz", O_CREATE|O_RDWR);
2 unlink("/tmp/xyz");
```

このコードは、一時的なinodeを作成するために理にかなった方法である。
プロセスがfdをcloseしたとき、もしくはプロセスが終了したときに、自動的に一時ファイルが消去される。

ファイルシステムを操作するxv6のコマンドは、mkdir, ln, rm等のようにユーザレベルのプログラムとして実装されている。
このデザインによって、新しいユーザーコマンドによってシェルを拡張する事が可能となる。
今となっては、この設計の利点は明らかだが、Unixが生まれた時代に設計された他のシステムは、しばしばそのようなコマンドをシェルの中に組み込んである。（そしてシェル自体もユーザ空間ではなくカーネル空間で実行されるようになってたりする。）

1つの例外がcdで、シェルの中に組み込まれている。
（sh.cのmain関数。二つ目のwhile文の中の最初のif文に注目）

```
01 int
02 main(void)
03 {
04   static char buf[100];
05   int fd;
06
07   // Assumes three file descriptors open.
08   while((fd = open("console", O_RDWR)) >= 0){
09     if(fd >= 3){
10       close(fd);
11       break;
12     }
13   }
14
15   // Read and run input commands.
16   while(getcmd(buf, sizeof(buf)) >= 0){
17     if(buf[0] == 'c' && buf[1] == 'd' && buf[2] == ' '){
18       // Clumsy but will have to do for now.
19       // Chdir has no effect on the parent if run in the child.
20       buf[strlen(buf)-1] = 0;  // chop \n
21       if(chdir(buf+3) < 0)
22         printf(2, "cannot cd %s\n", buf+3);
23       continue;
24     }
25     if(fork1() == 0)
26       runcmd(parsecmd(buf));
27     wait();
28   }
29   exit();
30 }
```

cdはシェル自身のカレントディレクトリを変えなければいけないということがその理由である。
もしcdが通常のコマンドと同じように実行されると、シェルは子プロセスをforkし、子プロセスがcd

を実行する訳で、これでは子プロセスのカレントディレクトリが変わるだけで、親プロセス（シェル）のカレントディレクトリは変わらないからである。

感想

cdコマンドのシェルへの組み込まれ具合にちょっと驚きました。

それ以外は知ってることばかりだったので特にアレです。

ファイルディスクリプタが存在するとunlinkしても消えず、closeされた段階で自動的に消えるというのは便利かなと思います。

次回はChapter 0の最後の節です。
短くコードもないみたいなのでサクっと読めそうですが、さらにその次のChapter 1がかなりハードウェア寄り内容みたいで理解できるか不安です。
がまぁ一日一節と行かなくてもなんとかぼちぼちやっていければと思っております。

カテゴリー: 技術 | タグ: xv6 | 投稿日: 2012/2/11 土曜日 [http://peta.okechan.net/blog/archives/1241] |