日曜研究室

技術的な観点から日常を綴ります

[xv6 #3] Chapter 0 – Operating system interfaces – Code: File descriptors

テキストの10〜12ページ

概要

ファイルディスクリプタそのものは（小さな）整数値であり、カーネルが管理する（プロセスで読み書きするための）オブジェクトである。
プロセスはファイルやディレクトリやデバイスを開いたり、パイプを生成したり、既に存在するディスクリプタを複製したりする事によってファイルディスクリプタを得るだろう。
内部的に、xv6のカーネルはプロセスごとのテーブル（どのプロセスもファイルディスクリプタ用の領域を個別に持ち、それはゼロから始まる）へのインデクスとしてファイルディスクリプタを使う。
お約束として、プロセスはファイルディスクリプタ”0″（標準入力）から読み込み、ファイルディスクリプタ”1″（標準出力）へ書きこむ。
エラーメッセージはファイルディスクリプタ”2″（標準エラー出力）に書きこむ。
いずれ分かるが、シェルはI/Oリダイレクションやパイプラインを実装するためのそのお約束を利用している。
シェルは、コンソールの標準のファイルディスクリプタとして、必ず3つのファイルディスクリプタを開く。
（前回載せたsh.cのmain関数の最初のwhile文を参照）

readシステムコールとwriteシステムコールは、オープン済みのファイルのファイルディスクリプタを受け取りバイト列を読み込んだり書き込んだりする。
read(fd, buf, n)は最大nバイトのデータをファイルディスクリプタfdから読み込み、bufへコピーし、実際に読み込んだバイト数を返す。
各ファイルディスクリプタはそれぞれ個別にオフセット値を持つ。
readは現在のファイルオフセットから読み、読み込んだ分だけオフセットを進める。
その次に呼んだreadは最初のreadが返した部分の次の部分を返す。
これ以上読み込むデータがなければ、readはファイルの終わりを示すためゼロを返す。

write(fd, buf, n)はbufからファイルディスクリプタfdにnバイト書きこみ、実際に書きこまれたバイト数を返す。
nバイトより少なく書き込まれるのはエラーが起きたときだけである。（ほんとか？）
readのように、writeは現在のファイルオフセットから書き込み、書き込んだ分だけオフセットを進める。
その後writeを呼び出す度に前回書き終わった次の箇所から書き込む。

次のプログラムの断片（catコマンドの重要な部分）は、標準入力から標準出力へデータをコピーするコードである。
エラーが起きたら、標準エラー出力へメッセージを書き込む。

```
01 char buf[512];
02 int n;
03
04 for(;;){
05   n = read(0, buf, sizeof buf);
06   if(n == 0)
07     break;
08   if(n < 0){
09     fprintf(2, "read error\n");
10     exit();
11   }
12   if(write(1, buf, n) != n){
13     fprintf(2, "write error\n");
14     exit();
15   }
16 }
```

とても重要な点として、このコードの断片はどこから読み込んでるか（ファイルからなのかコンソールからなのかパイプからなのか）を関知しない点が挙げられる。
同じく書き込みに関してもどこに書き込むか（コンソールなのかファイルなのかそれとも他の何かなのか）を関知しない。
“0”が入力で”1”が出力というファイルディスクリプタの習慣的な決まりごとが、catのシンプルな実装を可能にしている。

closeシステムコールはファイルディスクリプタを解放する。
そうすることで、そのファイルディスクリプタ（番号）を後でopenやpipeやdupというシステムコールで再利用できる。
新しくファイルディスクリプタを割り当てる場合、常に現在のプロセスの使用されてないディスクリプタの中で一番小さいものが選ばれる。

ファイルディスクリプタとforkは相互に作用しあって、I/Oリダイレクションの実装を簡単にしている。
forkは親プロセスのファイルディスクリプタテーブルもコピーする。
なので子プロセスは親が開いてるファイルと全く同じものを開いた状態で開始される。
execシステムコールは呼び出したプロセスのメモリを置き換えるが、ファイルテーブルは保持したままにする。

このforkとexecの振る舞いによって、forkした後いったんファイルを閉じて別のファイルを開きexecする事によって、I/Oリダイレクションが可能となる。（文章だけではピンと来ないけど以下のコードそのままの事を書いてある）
以下にシェルが”cat <input.txt”というコマンドを実行する場合と同じような動きをするコードを簡単に示す。

```
1  char *argv[2];
2  argv[0] = "cat";
3  argv[1] = 0;
4  if(fork() == 0) {
5    close(0);
6    open("input.txt", O_RDONLY);
7    exec("cat", argv);
8  }
```

子プロセスがファイルディスクリプタ”0″（”0″はファイルディスクリプタの最小値）を閉じた後、openによってinput.txtを読み込みモードで新しく開く。
そしてcatはinput.txtを指し示すファイルディスクリプタ”0″とともに実行される。
（catからはinput.txtの内容が標準入力に入力されたように見える）

xv6のシェルのI/Oリダイレクションのためのコードは、まさにこの方法で動作している。
（前回載せたsh.cのruncmd関数内のswitchのcase REDIR:の部分を参照）
この時点（case REDIR:の時点）で、既にフォークしていて、子プロセスのruncmdは新しいプログラムをロードするためにexecを呼ぶ。
forkとexecの分離がなぜいいアイデアなのか、ここでハッキリさせよう。
この分離によって、子プロセスが目的のプログラムを実行する前に、環境を調節する事が可能となる。

forkはファイルディスクリプタテーブルをコピーするにもかかわらず、ファイルディスクリプタのオフセット値は親と子の間で共有される。
この例を考えてみよう。

```
1  if(fork() == 0) {
2    write(1, "hello ", 6);
3    exit();
4  } else {
5    wait();
6    write(1, "world\n", 6);
7  }
```

これを実行すると、ファイルディスクリプタ”1″に割り当てられたファイルに”hello world”というデータが書き出されるだろう。
親プロセスのwrite（waitがあるおかげで必ず子プロセスのwriteの後に実行される）は、子プロセスが書き込み終わったところから書き込みはじめる。
この振る舞いによって、複数のコマンドの結果を一つに（ごちゃ混ぜにすることなく）まとめる事が可能となる。
例えば”(echo hello; echo world)>output.txt”のように。

dupシステムコールは既存のファイルディスクリプタを複製し、元のファイルディスクリプタが指し示すI/Oオブジェクトと同じ物を指し示す新しいファイルディスクリプタを返す。
両方のファイルディスクリプタは、forkで複製されるときと同じくオフセット値を共有する。
以下のコードは、ファイルへ”hello world”と書き込む別の方法である。

```
1 fd = dup(1);
2 write(1, "hello ", 6);
3 write(fd, "world\n", 6);
```

forkやdupで複製された時は、それぞれのファイルディスクリプタでオフセット値が共有される。
しかし、同じファイルをopenしただけの場合はオフセット値は共有されない。
dupによって”ls existing-file non-existing-file > tmp1 2>&1″のようなコマンドが可能となる。
“2>&1″は、コマンドに「ファイルディスクリプタ”2″は”1″の複製ですよ」という事を伝える。
存在してるファイルの名前と存在しないファイルによるエラーメッセージの両方はtmp1に出力される。
xv6のシェルは標準エラー出力のI/Oリダイレクションはサポートしてないが、実装は可能である。

ファイルディスクリプタは、何に接続してるかの詳細（書き込み先等がファイルかデバイスかコンソールかパイプか）を隠蔽すとても便利な概念である。

感想

ファイルディスクリプタが詳細を隠蔽するので実装が楽だよという話。
実際にcat.c見たら、テキストに載ってたコードの断片とは書き方が違うけど、全部で40行足らずでホントにシンプルでした。

forkでファイルディスクリプタテーブルも複製されるけど、execでは上書きされないとあります。
これによって、forkしてそのままexecした場合は親プロセスと同じファイルディスクリプタテーブルを引き継いで動作可能だし、forkした後一旦ファイルディスクリプタを閉じて別のものをopenしexecすれば（用語が適切かどうか怪しいですが）そのあたりをオーバーライド出来ると。

うまく表現できませんが、forkとexecで影響を受けるメモリの範囲にズレを作ることで柔軟性を確保してるんだなぁ。

あと、forkやdupで複製されたファイルディスクリプタはオフセットを共有する。コレ大事だと思います。
知らずにプログラム書いたら絶対ハマリそう。

しかし概要とか書きながら全部訳してるので全然概要じゃないですね。
まぁ翻訳が超怪しいので、本気にしないでねという意味で概要って付けとくのもいいかなとは思いますが、そのうちフォーマットを変えるかもしれません。

カテゴリー: 技術 | タグ: xv6 | 投稿日: 2012/2/8 水曜日 [http://peta.okechan.net/blog/archives/1226] |
