日曜研究室

技術的な観点から日常を綴ります

[xv6 #4] Chapter 0 – Operating system interfaces – Code: Pipes

テキストの12〜14ページ

本文

pipeは、ファイルディスクリプタの組（一方は読み込み、もう一方は書き込み）であり、プロセス間通信用にカーネルで提供される小容量のバッファである。
パイプの一方の端からデータを書きこむと、もう一方の端からそのデータを読み込むことが出来る。
パイプによってプロセス間通信が可能となる。

次のコードは、標準入力の代わりにパイプの読み込み側を割り当て、wcというプログラムを実行する。

```
01 int p[2];
02 char *argv[2];
03
04 argv[0] = "wc";
05 argv[1] = 0;
06
07 pipe(p);
08 if(fork() == 0) {
09   close(0);
10   dup(p[0]);
11   close(p[0]);
12   close(p[1]);
13   exec("/bin/wc", argv);
14 } else {
15   write(p[1], "hello world\n", 12);
16   close(p[0]);
17   close(p[1]);
18 }
```

このプログラムでは、新しいパイプを生成するためにpipeを呼んでいる。
pipeを呼ぶと、配列pに読み込み用と書き込み用のファイルディスクリプタがセットされる。
forkした後、親プロセスと子プロセスの両方はパイプを参照するファイルディスクリプタの組を持つことになる。

子プロセスは、パイプの読み込み側のファイルディスクリプタが0になるように複製する。
（close(0)した直後にdup(p[0])してるので、パイプの読み込み側はファイルディスクリプタ0に割当てられる）
そしてpが保持するファイルディスクリプタを二つとも閉じ、そしてwcを実行する。
wcがその処理の中で標準入力から読み込もうとするとき、それはパイプから読み取ることになる。
親プロセスは、パイプの書き込み側から書き込み、そしてその後両方のファイルディスクリプタを閉じる。

もしデータが準備できてない場合、パイプに対するreadは、データが書き込まれるか、パイプの書き込み側を参照する”全ての”ファイルディスクリプタ閉じられるまで待つ。
後者の場合（書き込み側を参照するファイルディスクリプタが全て閉じられた場合）、ちょうどファイルの末尾まで読んでしまったときと同じように、readは0を返す。
この「もう新しいデータが来ないとハッキリするまでreadがブロックされる」という事実は、wcを実行する前にパイプの書き込み側を閉じる重要なひとつの理由である。
もしwcがパイプの書き込み側を参照するファイルディスクリプタを持ったままだと、wcはend-of-fileを見ることはないだろう。（フリーズする）
（パイプの書き込み側を参照するファイルディスクリプタが全て閉じられなければEOFにならないので、親プロセス側だけではなく子プロセス側のも閉じなければならないってこと）

xv6のシェルは、上のコードと似た作法でパイプを実装している。
（実際のsh.c runcmd関数のパイプ関連の部分。前々回も載せてるけどまた載せときます。）

```
01 case PIPE:
02   pcmd = (struct pipecmd*)cmd;
03   if(pipe(p) < 0)
04     panic("pipe");
05   if(fork1() == 0){
06     close(1);
07     dup(p[1]);
08     close(p[0]);
09     close(p[1]);
10     runcmd(pcmd->left);
11   }
12   if(fork1() == 0){
13     close(0);
14     dup(p[0]);
15     close(p[0]);
16     close(p[1]);
17     runcmd(pcmd->right);
18   }
19   close(p[0]);
20   close(p[1]);
21   wait();
22   wait();
23   break;
```

まず、左側のコマンドと右側のコマンドを接続するためにパイプを生成する。
（左側・右側というのは例えば”ls | wc”というコマンドの場合はlsとwcを指す）
そして左側のコマンド・右側のコマンドそれぞれを実行するためにruncmdを呼び、その2つのコマン

ドの終了を待つためにwaitを2回呼ぶ。
例えば”a | b | c”というコマンドを実行した場合、まず左側のコマンドは”a”、右側のコマンドは”b | c”として2つの子プロセスにフォークし、さらに右側のコマンド”b | c”に対応するプロセスは左側”b”、右側”c”として2つの子プロセスにフォークする。
つまり、シェルがこのようなコマンドを実行するとき、プロセスのツリーが形成される。
このツリーの末端は最小単位のコマンドであり、末端以外のノード（内部ノード）は左右のコマンドに対応する子プロセスが終了するのを待つようなプロセスである。
原理的には、内部ノードにパイプの左端を受け持たせることが出来るが、そうすると実装は複雑になる。

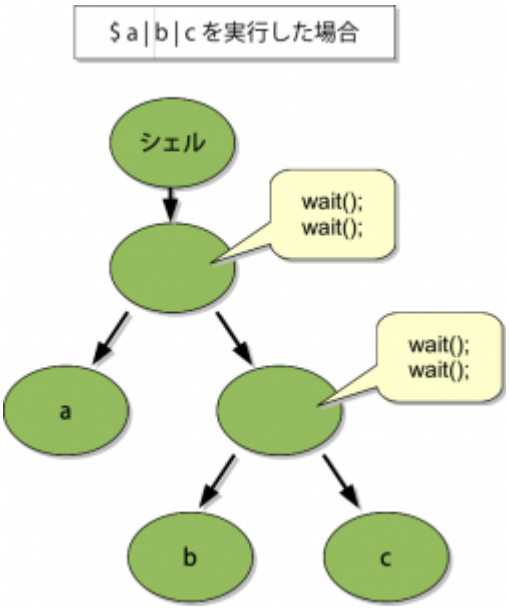
パイプは、一時ファイルと比べ特に優れてる、という風には見えないかもしれない。
パイプライン”echo hello world | wc”はパイプ無しで”echo hello world >/tmp/xyz; wc </tmp/xyz”のように書ける。
しかしパイプと一時ファイルには重要な違いが最低でも3つある。
一つ目、パイプは処理が終わると自動で消去される。
ファイルへのリダイレクションだと、処理が終わったあとシェルは注意深く一時ファイルを消さなければならなくなる。
二つ目、パイプは大きなデータストリームを任意に通すことが出来る。
ファイルへのリダイレクションを使ってると、全部のデータを保管できる空き容量がディスクにあるか気にしなければならなくなる。
三つ目、パイプは同期通信が出来る。
2つのプロセスは、writeでデータが送られるまでreadした側がブロックされる仕組みを利用してお互いに同期的にメッセージを送受信できる。

感想

パイプです。
かなり乱暴に言えば、見た目的にはシェルでコマンドを連携させるときに使う縦棒ですね。
あれを使うプログラムをどう実装するかの概要です。

ファイルディスクリプタは一つのパイプの一つの端に対していくつも生成することができます。
例では親プロセスと子プロセスで、同じパイプの端を参照するファイルディスクリプタをそれぞれ持ってます。
その参照してるファイルディスクリプタを全部closeしないとread側にEOFが伝わらないってところは重要かと思います。
知らずにプログラム書いたらなんでフリーズするんだろうって頭を悩ませることになりますね。
Python等でパイプを使うプログラムをいくつか書いたことがありますが、read側でブロックしまくりで頭を悩ませた覚えがあります。
こういう仕組みを知ってたらもっとすんなり行けたのかなぁ。

途中プロセスのツリーの所の文章が分かりづらいですが、絵にすると多分こんな感じのことかなと思います。



カテゴリー: 技術 l タグ: xv6 l 投稿日: 2012/2/9 木曜日 [http://peta.okechan.net/blog/archives/1234] l