日曜研究室

技術的な観点から日常を綴ります

[xv6 #49] Chapter 4 – Scheduling – Code: Pipes

テキストの59〜60ページ

本文

この章の最初のほうで扱ったシンプルなキューは、まるでおもちゃである。
しかし、xv6は読み込みと書き込みを同期化するためにsleepとwakeupを使うような本物のキューを2つ含む。
ひとつはIDEドライバで使われる。
プロセスはディスクへのリクエストをキューへ追加し、そしてsleepを呼ぶ。
割り込みハンドラはwakeupを使って、そのプロセスにそのリクエストが完了したことを通知する。

より複雑な実例としては、パイプの実装である。
我々は第0章でパイプのインターフェイスを見た。
パイプの一端に書きこまれたデータは、カーネルのバッファにコピーされ、そして他の一端から読み込めるようになる。
先の章では、パイプを取り巻くファイルシステムのサポートについて説明する。
しかし今は、pipewriteとpipereadの実装について見ていこう。

パイプはすべて、pipe構造体として表現される。
その構造体は、lockとバッファのためのdataというフィールドを含む。
nreadとnwriteというフィールドは、それぞれそのバッファから何バイト読み込んだか、何バイト書き込んだかを表す。
バッファは循環するようになっていて、buf[PIPESIZE-1]の後に書きこまれる次のバイトはbuf[0]である。
しかしnread、nwriteは循環しない。
この決まりごとは、バッファがいっぱいな状態（nwrite == nread+PIPESIZE）と、バッファが空である状態（nwrite == nread）の識別を可能にする。
しかしそれは、バッファのデータを参照するときに、単純にbuf[nread]とするのではなく、buf[nread % PIPESIZE]としなければならないということを意味する。（nwriteの場合も同じである）
2つの別のCPU上でpipereadとpipewriteが同時に呼び出されると仮定しよう。

pipe.c

```
001 #include "types.h"
002 #include "defs.h"
003 #include "param.h"
004 #include "mmu.h"
005 #include "proc.h"
006 #include "fs.h"
007 #include "file.h"
008 #include "spinlock.h"
009
010 #define PIPESIZE 512
011
012 struct pipe {
013   struct spinlock lock;
014   char data[PIPESIZE];
015   uint nread;     // number of bytes read
016   uint nwrite;    // number of bytes written
017   int readopen;   // read fd is still open
018   int writeopen;  // write fd is still open
019 };
020
021 int
022 pipealloc(struct file **f0, struct file **f1)
023 {
024   struct pipe *p;
025
026   p = 0;
027   *f0 = *f1 = 0;
028   if((*f0 = filealloc()) == 0 || (*f1 = filealloc()) == 0)
029     goto bad;
030   if((p = (struct pipe*)kalloc()) == 0)
031     goto bad;
032   p->readopen = 1;
033   p->writeopen = 1;
034   p->nwrite = 0;
035   p->nread = 0;
036   initlock(&p->lock, "pipe");
037   (*f0)->type = FD_PIPE;
038   (*f0)->readable = 1;
039   (*f0)->writable = 0;
040   (*f0)->pipe = p;
041   (*f1)->type = FD_PIPE;
042   (*f1)->readable = 0;
043   (*f1)->writable = 1;
044   (*f1)->pipe = p;
045   return 0;
046
047 //PAGEBREAK: 20
048  bad:
049   if(p)
050     kfree((char*)p);
051   if(*f0)
052     fileclose(*f0);
053   if(*f1)
054     fileclose(*f1);
055   return -1;
056 }
057
058 void
059 pipeclose(struct pipe *p, int writable)
060 {
061   acquire(&p->lock);
```

```
062   if(writable){
063     p->writeopen = 0;
064     wakeup(&p->nread);
065   } else {
066     p->readopen = 0;
067     wakeup(&p->nwrite);
068   }
069   if(p->readopen == 0 && p->writeopen == 0){
070     release(&p->lock);
071     kfree((char*)p);
072   } else
073     release(&p->lock);
074 }
075
076 //PAGEBREAK: 40
077 int
078 pipewrite(struct pipe *p, char *addr, int n)
079 {
080   int i;
081
082   acquire(&p->lock);
083   for(i = 0; i < n; i++){
084     while(p->nwrite == p->nread + PIPESIZE){  //DOC: pipewrite-full
085       if(p->readopen == 0 || proc->killed){
086         release(&p->lock);
087         return -1;
088       }
089       wakeup(&p->nread);
090       sleep(&p->nwrite, &p->lock);  //DOC: pipewrite-sleep
091     }
092     p->data[p->nwrite++ % PIPESIZE] = addr[i];
093   }
094   wakeup(&p->nread);  //DOC: pipewrite-wakeup1
095   release(&p->lock);
096   return n;
097 }
098
099 int
100 piperead(struct pipe *p, char *addr, int n)
101 {
102   int i;
103
104   acquire(&p->lock);
105   while(p->nread == p->nwrite && p->writeopen){  //DOC: pipe-empty
106     if(proc->killed){
107       release(&p->lock);
108       return -1;
109     }
110     sleep(&p->nread, &p->lock); //DOC: piperead-sleep
111   }
112   for(i = 0; i < n; i++){  //DOC: piperead-copy
113     if(p->nread == p->nwrite)
114       break;
115     addr[i] = p->data[p->nread++ % PIPESIZE];
116   }
117   wakeup(&p->nwrite);  //DOC: piperead-wakeup
118   release(&p->lock);
119   return i;
120 }
```

pipewriteは、まずパイプのロックを獲得することから始める。

そのロックは、読み書きバイト数、データ、そしてそれらに関係するインバリアントを保護する。

piperead も、まずロックの獲得を試みるが、不可能である。（今はpipewriteとpipereadを同時に実行してると仮定してるので）
こちらはロックが解放されるまでacquireのところで止まったままになる。
piperead が待ってる間、pipewriteは渡されたデータをループで1バイト毎（addr[0], addr[1], …, addr[n-1]）に処理し、それぞれpipeに追加する。
このループ中でバッファがいっぱいになる可能性がある。
この場合pipewriteは、データが準備できたという事をスリープ中の読み込み側に伝えるためにwakeupを呼び、そして、読み込み側がバッファからいくらかデータを取り出すのを待つために、&p->nwriteというチャンネルでスリープする。
sleepによって、このpipewriteを実行しているプロセスをスリープさせる処理の一環としてp->lockが解放される。

そしたらp->lockが獲得可能となるので、pipereadはそのロックを獲得し処理を開始する。
piperead は、p->nread != p->nwriteであることに気づき（pipewriteがスリープしたのはp->nwrite == p->nread+PIPESIZEになったからである）、最初のwhileループをスキップする。
それからpipereadは、パイプからデータをコピーし、nreadをコピーしたバイト数の分だけインクリメントする。
これでパイプにまた書き込めるようになったので、pipereadは呼び出し側に戻る前に、wakeupを呼び、スリープ中の書き込み側を起こす。
wakeupは、&p->nwriteというチャンネルでスリープ中のプロセスを探す。
そのプロセスは、pipewriteを実行中だったがバッファがいっぱいになったので止まってたプロセスである。
wakeupはそのプロセスをRUNNABLEとしてマークする。

パイプのコードは、読み込み側と書き込み側で別のスリープチャンネル（p->nreadとp->nwrite）を使う。
このことによって、（あまりありそうではないが）多くの読み込み手と書き込み手が同じパイプを待つような場合により効率的になる。
パイプのコードは、スリープする条件のチェックを行うループの内側でsleepを呼ぶ。
複数の読み手や書き手がある場合、最初のプロセスを除いてすべて起こされるが、まだ条件が偽であるので再度スリープする。

感想

パイプの実装についてです。
acquire/relese、sleep/wakeupを使ってるので、背景まで含めると結構複雑なコードだと思いますが、今まで学んだ分のおかげですんなり読めました。

そういう背景を除けば、このコードのキモはpipe構造体のdataフィールドのみ循環利用される事かなと思います。

書き込みと読み込みは必ず交互に実行されるわけではなく、また読み込みバイト数が書き込みバイト数を下回る可能性もあるということを考えると、循環以外の考え方はちょっと現実的じゃない気がします。

最後の方にも書かれてますが、sleepはその条件をチェックするループの内側で呼べという決まりごとがここでも守られてます。
なぜそうすべきか？ってのは、一言で言えば「起こされたときにまだ条件が整ってなければまたsleepすべきだから」ってところですかね。

カテゴリー: 技術 | タグ: xv6 | 投稿日: 2012/3/24 土曜日 [http://peta.okechan.net/blog/archives/1564] |