日曜研究室

技術的な観点から日常を綴ります

[xv6 #30] Chapter 2 – Traps, interrupts, and drivers – Code: Disk driver

テキストの39〜41ページ

本文

IDEデバイスは、PCの標準IDEコントローラに接続されたディスクへのアクセスを提供する。
IDEは、今となってはSCSIやSATAのような流行からは外れてしまっているが、そのインターフェイスは単純で、個別のハードウェアの詳細に立ち入る必要なくドライバ全体の構築に集中させてくれる。

ディスクドライバは、バッファと呼ばれるデータの構造体（buf構造体）としてディスクのセクタを表現する。
どのバッファも、個別のディスクデバイス上の一つのセクタの内容を表す。
devフィールドとsectorフィールドは、それぞれデバイス番号とセクタ番号を表し、dataフィールドはディスクのセクタの内容のメモリ上のコピーである。

buf.h

```
01 struct buf {
02   int flags;
03   uint dev;
04   uint sector;
05   struct buf *prev; // LRU cache list
06   struct buf *next;
07   struct buf *qnext; // disk queue
08   uchar data[512];
09 };
10 #define B_BUSY  0x1  // buffer is locked by some process
11 #define B_VALID 0x2  // buffer has been read from disk
12 #define B_DIRTY 0x4  // buffer needs to be written to disk
```

flagsフィールドは、メモリとディスクの関連を追跡するためにある。
B_VALIDフラグは、データが読み込まれたことを意味し、B_DIRTYフラグはデータをディスクへ書きだす必要がある事を意味する。

B_BUSYフラグはロックのためのフラグであり、そのバッファをどこかのプロセスが使っていてその他のプロセスは使うべきではない事を指し示す。
バッファにB_BUSYフラグがセットされたとき、我々は「そのバッファはロックされいる」と言う。

カーネルは、起動時にmain関数からideinit関数を呼ぶことによってディスクドライバを初期化する。
ideinit関数は、picenable関数とioapicenable関数を呼び、IDE_IRQ割り込みを有効にする。
picenable関数は、ユニプロセッサ上で割り込みを有効にする。
ioapicenable関数は、マルチプロセッサ上で割り込みを有効にするが、最後のCPU（ncpu-1）についてのみである。
つまりプロセッサが2つあるシステムでは、CPU 1がディスク割り込みを制御する。

ide.cのideinit関数

```
01 void
02 ideinit(void)
03 {
04   int i;
05
06   initlock(&idelock, "ide");
07   picenable(IRQ_IDE);
08   ioapicenable(IRQ_IDE, ncpu - 1);
09   idewait(0);
10
11   // Check if disk 1 is present
12   outb(0x1f6, 0xe0 | (1<<4));
13   for(i=0; i<1000; i++){
14     if(inb(0x1f7) != 0){
15       havedisk1 = 1;
16       break;
17     }
18   }
19
20   // Switch back to disk 0.
21   outb(0x1f6, 0xe0 | (0<<4));
22 }
```

次に、ideinit関数は、ディスクハードウェアを探る。
それはidewait関数の呼び出しによりはじまり、ディスクがコマンドを受け付けれる状態になるまで待つ。
PCのマザーボードは、I/Oポート0x1f7でディスクハードウェアの状態ビットを提供する。
idewait関数は、ビジービット（IDE_BSY）がクリアされ準備完了ビット（IDE_DRDY）がセットされるまで、その状態ビットをポーリングする。

ide.cのidewait関数

```
01 // Wait for IDE disk to become ready.
02 static int
03 idewait(int checkerr)
04 {
05   int r;
06
```

```
07    while(((r = inb(0x1f7)) & (IDE_BSY|IDE_DRDY)) != IDE_DRDY)
08      ;
09    if(checkerr && (r & (IDE_DF|IDE_ERR)) != 0)
10      return -1;
11    return 0;
12  }
```

それが終われば、ディスクコントローラは準備完了となり、ideinitはそこから提供される多くのディスクをチェックすることが出来るようになる。
ideinitはディスク0があると仮定する（なぜならブートローダとカーネルの両方はディスク0から読み込まれてるので）が、ディスク1があるかどうかはチェックしなければならない。
ideinitは、ディスク1を選択するためにI/Oポート0x1f6へ書き込み、そしてディスクの状態ビットが準備完了状態を表すようになるまで待つ。
もしそうでなければ、ideinitは、そのディスクはないという事にする。

ideinitの後、バッファのキャッシュがiderw関数を呼ぶまで、ディスクは使用されない。
iderw関数は、フラグによってロック状態となっているバッファを更新する。
もしB_DIRTYがセットされていれば、iderwはそのバッファの内容をディスクへ書き込む。
もしB_VALIDがセットされていなければ、iderwはディスクからバッファへ読み込む。

ide.cのiderw関数

```
01  // Sync buf with disk.
02  // If B_DIRTY is set, write buf to disk, clear B_DIRTY, set B_VALID.
03  // Else if B_VALID is not set, read buf from disk, set B_VALID.
04  void
05  iderw(struct buf *b)
06  {
07    struct buf **pp;
08
09    if(!(b->flags & B_BUSY))
10      panic("iderw: buf not busy");
11    if((b->flags & (B_VALID|B_DIRTY)) == B_VALID)
12      panic("iderw: nothing to do");
13    if(b->dev != 0 && !havedisk1)
14      panic("iderw: ide disk 1 not present");
15
16    acquire(&idelock);  // DOC:acquire-lock
17
18    // Append b to idequeue.
19    b->qnext = 0;
20    for(pp=&idequeue; *pp; pp=&(*pp)->qnext)  // DOC:insert-queue
21      ;
22    *pp = b;
23
24    // Start disk if necessary.
25    if(idequeue == b)
26      idestart(b);
27
28    // Wait for request to finish.
29    // Assuming will not sleep too long: ignore proc->killed.
30    while((b->flags & (B_VALID|B_DIRTY)) != B_VALID){
31      sleep(b, &idelock);
32    }
33
```

```
  release(&idelock);
}
```

ディスクのアクセスには、典型的にはミリ秒単位の時間がかかる。
それはプロセッサにとっては長い時間である。
ブートローダは、読み込みコマンドをディスクに発行し、そしてデータが準備出来るまで状態ビットを繰り返し読む。
このポーリングもしくはビジーウェイトは、他にもっといい方法を持たないブートローダでは仕方がない。
一方OS起動後では、別のプロセスを実行し、そのプロセスでディスク操作が完了したときに割り込みを受けて制御するという、より効率的な方法が使える。
iderwは後者の手法を採用していて、キューで未実行のディスクへのリクエストのリストを保持し、各リクエストが終わったときに、それを探すために割り込みを使う。
iderwがリクエストのキューを管理するとはいえ、シンプルなIDEディスクコントローラは、一度に一つの操作しか制御できない。
ディスクドライバは、ディスクハードウェアへキューの一番前にあるバッファを送るという一定の仕事を管理する。
他は、ディスクハードウェアの仕事を単純に待つだけである。

iderwは、キューの最後にバッファbを追加する。（Append b to idequeue.とコメントが付いてる部分）
もしそのバッファがキューの一番前なら、iderwはidestart関数を使ってディスクハードウェアへそれを送信しなければならない。
そうじゃなければ、バッファは一度開始され、その前にあるバッファによって処理される。
（ここ文章が分かりづらいけど、特定のバッファに関するディスク操作が完了したときに実行されるideintr（後で説明）内で、キューが空じゃなければ続けてその後のバッファを処理するようになっているということみたい。）

ide.cのidestart関数

```
// Start the request for b.  Caller must hold idelock.
static void
idestart(struct buf *b)
{
  if(b == 0)
    panic("idestart");

  idewait(0);
  outb(0x3f6, 0);  // generate interrupt
  outb(0x1f2, 1);  // number of sectors
  outb(0x1f3, b->sector & 0xff);
  outb(0x1f4, (b->sector >> 8) & 0xff);
  outb(0x1f5, (b->sector >> 16) & 0xff);
  outb(0x1f6, 0xe0 | ((b->dev&1)<<4) | ((b->sector>>24)&0x0f));
  if(b->flags & B_DIRTY){
    outb(0x1f7, IDE_CMD_WRITE);
    outsl(0x1f0, b->data, 512/4);
  } else {
    outb(0x1f7, IDE_CMD_READ);
```

```
20    }
21  }
```

idestart関数は、バッファのデバイスとセクタのための読み込み・書き込みどちらも発行する。
読み込みか書き込みかはフラグで見分ける。
もし操作が書き込みなら、idestartはこの時点でデータを供給しなければならず（outsl(0x1f0, b->data, 512/4); のところ）、そして割り込みはデータがディスクに書き込まれたことを示す。
もし操作が読み込みなら、割り込みはデータが準備完了であることを示し、そして割り込みハンドラはそれを読み込む。
iderwはIDEデバイスについての詳細な知識を持ち、正しいポートに正しい値を書き込むことに注意。
もし、一連のoutb命令に間違いがあったら、IDEは我々がやって欲しいこととは違う何かを行うだろう。
それらの詳細を正しく知ることは、デバイスドライバを書くことがなぜチャレンジングであるかの一つの理由である。

キューへリクエストを追加したら、もし必要ならそのキューは開始される。
そしてiderwは結果を待つ。
上で説明したように、ポーリングはCPUの使い方としては非効率である。
その代わり、iderwは、操作の完了を示すフラグが割り込みハンドラによってバッファに記録されるまで、sleep関数で待つ。
このプロセスがスリープしてる間、xv6はCPUに仕事をさせるために他のプロセスをスケジュールするだろう。

最終的に、ディスクは操作を完了し割り込みを発生させる。
trap関数は、その割り込みを制御させるためにideintr関数を呼ぶ。
ideintr関数は、どの操作が起きたのか知るためにキューの最初のバッファを調べる。
もし、バッファが読み込み完了でディスクコントローラがデータを準備して待ってるなら、ideintrは、inslを使ってバッファへデータを読み込む。
ideintrは、B_VALIDをセットし、B_DIRTYをクリアし、バッファ上で寝ているどこかのプロセスを起こす。
（Wake process waiting for this bufのコメントの部分）
これでバッファが準備出来たことになる。
最後にideintrは、次に待ってるバッファをディスクへ送らなければならない。

trap.cのtrap関数のディスク割り込みの処理の部分

```
1  case T_IRQ0 + IRQ_IDE:
2    ideintr();
3    lapiceoi();
4    break;
```

ide.cのideintr関数

```
01  // Interrupt handler.
```

```
02 void
03 ideintr(void)
04 {
05   struct buf *b;
06
07   // Take first buffer off queue.
08   acquire(&idelock);
09   if((b = idequeue) == 0){
10     release(&idelock);
11     // cprintf("spurious IDE interrupt\n");
12     return;
13   }
14   idequeue = b->qnext;
15
16   // Read data if needed.
17   if(!(b->flags & B_DIRTY) && idewait(1) >= 0)
18     insl(0x1f0, b->data, 512/4);
19
20   // Wake process waiting for this buf.
21   b->flags |= B_VALID;
22   b->flags &= ~B_DIRTY;
23   wakeup(b);
24
25   // Start disk on next buf in queue.
26   if(idequeue != 0)
27     idestart(idequeue);
28
29   release(&idelock);
30 }
```

感想

ディスクドライバの実装です。

キューが空のときに追加された場合しかidestartしないと説明があって（コードもそうなってる）、キューが空じゃないときに追加されたリクエストはどうなるのよ、と最初混乱しましたが、割り込みハンドラ側で芋づる式にキューを次々と処理していく（1バッファごとに割り込みが発生）ようになってて納得しました。

実際のOSでは、キューを高機能化して、例えば出来るだけセクタが連続するようにバッファを並びかえたり、電源のブチ切りでなるべくデータが失われないような工夫があったりするんじゃなかろうかと思います。

カテゴリー: 技術 | タグ: xv6 | 投稿日: 2012/3/6 火曜日 [http://peta.okechan.net/blog/archives/1402] |