日曜研究室

技術的な観点から日常を綴ります

[xv6 #34] Chapter 3 – Locking – Race conditions

テキストの43〜45ページ

本文

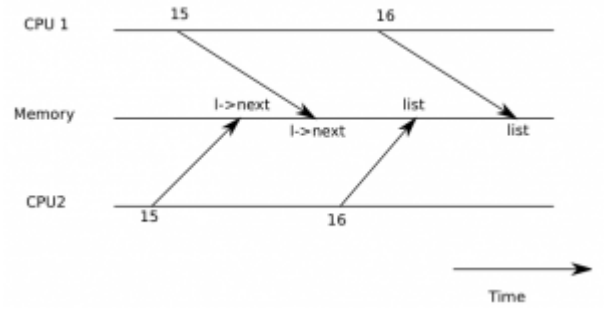
なぜロックが必要か、その例として、xv6におけるIDEのディスクのように、いくつかのプロセッサが一つのディスクを共有する場合を考える。
ディスクドライバは、未実行のディスクへのリクエストのリンクリストを管理（ide.cのidequeueは現在読み書き中のバッファを指し、idequeue->qnextは次に処理されるべきバッファを指す。）し、プロセッサは、同時にそのリストへ新しいリクエストを追加するだろう（ide.cのiderw関数を参照）。
もし、リクエストの同時発生がありえないのならば、そのリンクリストは以下のように実装出来る。

```
01 struct list {
02   int data;
03   struct list *next;
04 };
05
06 struct list *list = 0;
07
08 void
09 insert(int data)
10 {
11   struct list *l;
12
13   l = malloc(sizeof *l);
14   l->data = data;
15   l->next = list;
16   list = l;
17 }
```

この実装の正しさを調べる事は、データ構造とアルゴリズムの授業における典型的な課題である。
この実装は、正しいと証明できるが、少なくともマルチプロセッサ上ではそうではない。
2つの別のCPUが、同時にinsert関数を実行した場合、どちらかが16行目を実行する前に、どちらも15行目を実行するという事態が起きる可能性がある。（図3-1参照）
この事態が発生した場合、リストの2つのノードのnextには、listの古い値がセットされるだろう。
16行目でlistへの2つの代入が発生したとき、2つ目の代入が1つ目の代入を上書きする。

そのノードへ最初に割り当てられた値は失われるだろう。
この種の問題は、競合状態（レースコンディション, race condition）と呼ばれる。
処理過程におけるその問題は、それに関わる2つのCPUの厳密なタイミングに依存し、それらのメモリ操作は、メモリシステムによって実行され、その結果として復元するのが難しくなる。
例えば、insert関数をデバッグするときprint文を追加する事は、この処理過程を再現できなくするのに十分なほど、その実行のタイミングを変えてしまうだろう。

図3-1 処理過程の例



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
34   release(&idelock);
35 }
```

この処理過程を避ける典型的な方法は、ロックを使うことである。
ロックは排他を確実にする。
そうすることで、一度にひとつのCPUだけがinsert関数を実行出来る。
そうなると上で説明したような競合状態は起きなくなる。
上のコードのロックを使った正しいバージョンは、いくつかの行を追加しただけである。
（追加というコメントが付いてる行が追加した行です。）

```
01 struct list {
02   int data;
03   struct list *next;
04 };
05
06 struct list *list = 0;
07 struct lock listlock; // 追加
08
09 void
10 insert(int data)
11 {
12   struct list *l;
13
14   acquire(&listlock); // 追加
15   l = malloc(sizeof *l);
16   l->data = data;
17   l->next = list;
18   list = l;
19   release(&listlock); // 追加
20 }
```

我々が、「ロックがデータを保護している」という言うとき、正確には「ロックが、データへ適用するインバリアントのいくつかの集合を保護している」という事を意味する。
インバリアントとは、一連の操作を通して管理されているデータ構造の属性である。
典型的に、ひとつの操作は、操作が開始されたときに真となるインバリアントに依存するふるまいを正す。
操作は、一時的にそのインバリアントに違反するが、終了するまえにそれらを回復しなければならない。
例えばリンクリストの場合、そのインバリアントは、リスト内の最初のノードを指すlistと、次のノードを指す各ノードのnextフィールドである。
insertの実装は、一時的にこのインバリアントを侵害する。
mallocの行は、リストの新しい要素であるlを、リストの最初のノードになるよう生成する。
しかしこの時点では、lの次へのポインタはまだ次のノードを差してはおらず（それはl->next = list;の行で行われる）、listはまだlを差してはいない（それはlist = l;の行で行われる）。
我々が上記で調査した競合状態は、一時的に侵害されている間、そのリストのインバリアントに依存したコードを2番目のCPUが実行することによって、発生した。
ロックを正しく使えば、一度にひとつのCPUだけがデータ構造上で操作することを確実にでき、その結果、データ構造のインバリアントが中途半端な状態のときに、他のCPUがそのデータ構造を操作することはなくなるだろう。

感想

ロックの使い方を軽くといったところでしょうか。
サンプルのコード自体は、ユーザ空間のプログラムでロックを使うときのコードと全く同じですね。

インバリアントは訳が難しいのでそのままですが、ここでいうインバリアントはデータ構造とか形式とか一貫性とかそういった感じの意味かと思います。

今回、特に最後の段落が難しいです。
このテキストやってると、抽象的で言い回しも微妙でtypoがあってとても分かりづらい節や段落にたまに出くわします。
担当者の違いでしょうか。

カテゴリー: 技術 | タグ: xv6 | 投稿日: 2012/3/10 土曜日 [http://peta.okechan.net/blog/archives/1415] |