日曜研究室

技術的な観点から日常を綴ります

[xv6 #36] Chapter 3 – Locking – Modularity and recursive locks

テキストの46ページ

本文

システムは、モジュール式の抽象概念を用いてクリーンさを保つよう設計される。
呼び出される側がその個々の機能性をどう実装するかについて、呼び出し側が知る必要がない場合、そのモジュール式の抽象概念を用いることは最善の方法である。
ロックは、このモジュール性を損なう。
例えば、あるCPUがロックを保持している場合、そのロックを再度獲得しようとするような関数fはどんなものでも呼ぶことは出来ない。
fから戻るまで呼び出し側はロックを解放することが出来ないので、fが同じロックを獲得しようとする場合、それは永遠にスピン（ロック解放待ちのループ）し、デッドロック状態になる。

呼び出し側と呼び出される側が使ってるロックを隠蔽できるようにする、透過的な解決方法はない。
ひとつは汎用的で透過的だが、不十分な解決方法として、再帰ロック（recursive locks）というものがあり、呼び出される側が呼び出し側で既に保持されたロックを再度獲得する事を可能にする。
この解決方法の問題点は、再帰ロックはインバリアントの保護には使えないということである。
insert関数がacquire(&listlock)を呼んだとき（前々回のサンプルコード参照。一応今回も再掲しておきます。）、リスト操作中に他の関数はないので、ロックを保持している関数が他に無いと仮定でき、一番重要な点として、全てのリストのインバリアントを保持できる。
再帰ロックを使うシステムでは、insert関数がacquire関数を呼んだ後でも何も仮定出来ることはない。
あるinsert関数の呼び出し側がすでにロックを保持し、そしてリストのデータ構造を編集中であっても、おそらくacquireは成功する。
その場合インバリアントは保持されたりされなかったりする。
リストはもはやインバリアントを保護しない。
ロックは、違うCPU同士から保護するように、呼び出し側と呼び出される側同士から保護するためだけに重要である。

再帰ロックは、その特徴を諦める。

前々回のサンプルコード

```
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 }
```

理想的で透過的な解決方法はないので、我々は関数の設計について熟考しなければならない。
プログラマは、関数fが必要とするロックを保持している間、その関数fを呼び出さないように配置しなければならない。
ロックは、我々の抽象概念のなかへロック自身を無理やり押し込む。

感想

原文の言い回しが微妙で難しかったので、今回は特に訳に自信がないです。

まとめると、
通常のロックは（ロックを要求する他の関数によって）、release無しで連続acquireすることが出来ない（永久スピンになる）ので、その回避策として再帰ロックというものがあるけど、それはそれでインバリアントの保護を保証出来ず、本末転倒で使えないので、通常のロックを使いながらロックを多重に要求するようなコードパスが発生しないように慎重に設計しましょうという話かと思います。

ではなぜ再帰ロックはインバリアントを保護できないか？
それについてはあまりピンと来てませんが、以下のようなことかなと思います。
通常のロックの場合、複数のCPUで同じデータ構造を同時にごちゃまぜに弄る可能性を排除出来るのはもちろん、（呼び出し先でロックを獲得出来れば呼び出し元がロックを獲得してない事が明らかなので）呼び出し元がインバリアントを一時的に壊すような操作中である可能性も排除出来ます。
一方再帰ロックの場合だと、呼び出し先がロックを獲得出来ても、実は呼び出し元もロックを獲得済みでインバリアントを一時的に壊す操作の途中かもしれません。
そうなると呼び出し元の実行が終わったあともインバリアントは壊れたままになる可能性があるとい

うことかなと思います。
再帰ロックでインバリアントが壊れる場合、複数のCPUやプロセスやスレッドで同じデータを弄って壊すパターンとはちょっと違いますが、呼び出し先の意図しない順番で処理が実行される事になります。

ってことかなぁと。
多分再帰ロックを使ってても、プログラマが気をつければ（机上の空論だけど）、データが壊れる可能性がない事を全体的には保証出来るかもしれませんが、多分個別の関数単位では保証出来ないんだろうなと思います。
ほんとかな〜。

再帰ロックでインバリアントを保護できない理由を誰か明快に教えてください(´；ω；｀)

カテゴリー: 技術 | タグ: xv6 | 投稿日: 2012/3/12 月曜日 [http://peta.okechan.net/blog/archives/1426] |