OS 第 2 回課題レポート

1510151 柳 裕太

2017 年 7 月 21 日

1 課題 1: 和訳

1.1 デッドロックの条件

4 つの条件がデッドロック発生を引き起こす。[C+71]

- **相互排除:** 複数スレッドが必要なリソースの排他制御を要求するとき (例: 単一スレッドがロックの所有権を獲得する)
- **保持&待ち:** 複数スレッドが追加リソースを待つ間に、(例: これが、彼らがリソースの獲得を望んでいるように見える) 割り当てられたリソースを保持する (例: これが、彼らが既にリソースを獲得したもののように見える)
- **先取なし:** 複数リソース (例: ロック) を、所有されているスレッドから強制消去することができない
- **循環待機:** 次のスレッドが必要としている、1 つあるいはそれ以上のリソース (例: ロック) を持つスレッドらによる、循環連鎖が存在する

もしも 4 つのうちいずれかのの条件に出くわしていないのなら、デッドロックは発生することができない。それゆえ、我々はまずデッドロックを回避する技術調べる…これらの各戦術が、これらの条件の 1 つを発生から回避することを求めており、それゆえ、これが 1 つのデッドロック問題に対する対応方法となっている。

1.2 防止策

1.2.1 循環待機

概ね、多くの実践的な防止技術 (そして、確かにその 1 つはよく採用されている) は、あなたに循環待機を決して引き起こさないようなロックするコードを書かせるだろう。最もわかりやすくそれを遂行する方法は、たった 2 つのロックを取り入れたシステム (L1 と L2) で、あなたは常に L1 が L2 の前に獲得させることで、デッドロックを防止することができる。厳格な順番こそが、環状待機発生なしを確証させ、それこそがデッドロックなしとなる。

もちろんのこと、更に複雑なシステム内では、2 つ以上のロックが存在するだろうし、またゆえに全体のロックの順序も達成は難しくなるかもしれない (そしてあるいは不要なのかもしれないのはさておき)。それゆえ、**部分配列**はデッドロックを回避するための、ロック獲得構造を使いやすい方法になりえるのである。1 つの素晴らしい部分ロック順の現実の例は Linux[T+94] にお

ける、メモリマッピングコードにて閲覧可能であり、ソースコード上部のコメントには、10 の異なるロック獲得順のグループを明らかにしており、”i_mmap_mutex の前に i_mmap”のようなシンプルな 1 つや”mapping->tree_lock の前に swap_lock の前に private_lock の前に i_mmap_mutex” のようなより複雑な順序を含んでいる。

あなたが想像できるように、双方の全体と一部の獲得順はロックを行う戦術における慎重なデザインであり、また念入りな注意によって構築されなければならない。更に言えば、順番はたかが慣習であり、また杜撰なプログラマーは簡単にプロトコルをロックすることを無視し、そしてデッドロックを引き起こす可能性もありえる。最終的に、ロック順はコードベースと、そしていかに多くのルーチンが呼び出されるか…における深い理解必要とし、たった 1 つのミスが”D”(Deadlock の象徴) の世界に入る結果に終わる。

1.2.2 保持&待ち

デッドロックにおける保持&待ちの条件は、すべてのロックを 1 度獲得することで回避することができる、原始的だが。実践では、これは以下のコードで達成することが可能である。

```
1  pthread_mutex_lock(prevention);    // begin lock acquistion
2  pthread_mutex_lock(L1);
3  pthread_mutex_lock(L2);
4  ...
5  pthread_mutex_unlock(prevention); // end
```

最初の prevention ロックを獲得することで、このコードは中央のロック獲得や、そしてそれによるデッドロックがが再発することが引き起こしかねない時期尚早なスレッド切替ががないことを確約し、防ぐ。もちろんのこと、これはいつでもどのスレッドがロックを獲得することを要求し、それが最初に得るグローバル防止ロックとなる。例えば、もし別のスレッドがロック L1 と L2 を、それぞれ別の順で獲得しようとした時、問題はない、何故ならばこれは防止ロックを実行中に保持しているためである。

ここで留意すべきは、この解決策は多数の理由において問題があることだ。従来では、作品を我々とは反対にカプセル化した…ルーチンが呼ばれた時、このアプローチは我々に、まさにどのロックが保持されそして獲得するかを前もって知ることを要求する。この技術はまた、彼らが本当に必要とされたときに代わって、すべての早い段階 (一度) で獲得されなければならないロックの同時並行性を落としかねないのである。

1.2.3 先取なし

我々は一般的にロックをアンロックがコールされるまで獲得するものとしてみるため、1 つのロックを待っている時に我々は別のものを保持していることにより、複数ロック獲得は通常我々をトラブルへ導く。多くのスレッドライブラリは、この状況を避けることを助けるための、さらなるフレキシブルなインターフェースのセットを提供している。

特に、pthread mutex trylock() ルーチンは、ロック (可能ならば) と、成功あるいはロックが保持されていることを示すエラーコードを返すとの両方を獲得し、後者においては、あなたがもしそのロックを獲得したければ、後でもう一度やることが可能である。

かようなインターフェースは次のようにデッドロックフリーを構築することに使われており、ロ

バストオーダリングは獲得プロトコルをロックする:

```
1  top:
2    pthread_mutex_lock(L1);
3    if (pthread_mutex_trylock(l2)) != 0) {
4      pthread_mutex_unlock(L1);
5      goto top;
6    }
```

ここで留意すべきは、別のスレッドは同じスレッドに続くことができるものの、ロック獲得は別の順序 (L2 から L1) であり、またプログラムは未だデッドロックフリーであろうということだ。1 つの新たな課題が浮上する、しかしながら……**ライブロック**である。これは 2 つのスレッドが再三このシーケンスを試し、また定期的に 2 つのロック獲得を失敗することも可能 (おそらくそうでないにもかかわらず) であるこのケースにおいては、双方のシステムはこのコードにより何度も何度も実行されるものの (そしてそれゆえこれはデッドロックではない)、進歩が作られることはなく、それゆえにライブロックと呼ばれる。ライブロック問題にもまたたくつかの解決法があり、例えば、1 つはランダムの遅延をループバックの前に入れ、全体をもう一度行い、それゆえにスレッド競合における干渉が繰り返される見込みが減少される。

1 つのこの解決法の最後のポイントは、これはトライロックのアプローチを使うにおいて難しい部分を回避することである。最初の課題はこれはカプセル化によってもう一度浮上しかねなくなっており、もし 1 つのロックが幾つかの呼び出されたルーチン内で時間がかかると、ジャンプバックが発生して更に実行において複雑になる。もしコードがいくつかのリソース (L1 とは別) をこの方法で獲得した場合、確実に注意深くリリースしなければならず、例えば、もし L1 より後に獲得した場合、コードはいくつかのメモリが割り当てられ、これはメモリが L2 に割り当てられるのを失敗されるようにリリースしなければならず、全体のシーペンスを行うために事前にトップにジャンプバックする。しかしながら、限定された状況 (例: Java ベクターメソッドが早く言及していた)、このアプローチのタイプがうまく動作する。

1.2.4 相互排除

最後の防止技術は相互排除の必要性を避けることにある。一般的には、我々はこれが難しいことを知っており、なぜならばコードに本当はクリティカルセクションをもったまま、走ることを我々は望むからである。では、我々は何ができるのだろうか？

ハーリヒイは 1 つの全体においてロックがない、データ構造のデザインアイデアを持っていた [H91, H93]。このアイデアの背後には**ロックフリー** (そして**ウェイトフリー**とも関連がある) が現れることは単純で、強力なハードウェア命令を使用し、あなたは明確なロックを要求しないマナーにおいてデータ構造を構築することができる。

シンプルな例として、我々がもつコンペアアンドスワップ命令を取り上げてもらうと、これは以下のコードから、あなたが核心的命令がハードウェアから提供されることを思い出すかもしれない。

```
1  int CompareAndSwap(int *address, int expected, int new){
2    if (*address == expected){
3      *address = new;
```

```
4       return 1;  // success
5     }
6     return 0;  // failure
7   }
```

我々が今自動的に値を現在の値からインクリメントしたいときを想像してみましょう。我々は以下のように行うことが可能である。

```
1   void AtomicIncrement(int *value, int amount) {
2     do {
3       int old = *value;
4     } while (CompareAndSwap(value, old, old + amount) == 0);
5   }
```

ロック獲得に代わって、アップデートを行うとして、そしてその後にリリースを行い、我々が代わりに、繰り返し値を新しい規模にアップデートすることをトライしたり、コンペアアンドスワップを使って実行したりするアプローチを構築した。マナーにおいては、何のロックも獲得されておらず、また何のデッドロックも浮上できていない (ライブロックは未だ可能性があるにもかかわらず)。

我々にわずかにもっと複雑な例を考慮させてもらう、リスト挿入だ。こちらがリストの頭に挿入するコードである。

```
1   void insert(int value){
2     node_t *n = malloc(sizeof(node_t));
3     assert(n != NULL);
4     n->value = value;
5     n->next  = head;
6     head =n;
7   }
```

このコードは単純な挿入動作をする、しかしながらもしもマルチスレッドが”同時に”呼び出されたら、レースコンディション (もし何故か図示できるなら見ること) を抱えることになる。もちろんのこと、我々はこれを以下のコードの周りのロック獲得・放出によって解決することができる。

```
1   void insert(int value){
2     node_t *n = malloc(sizeof(node_t));
3     assert(n != NULL);
4     n->value = value;
5     pthread_mutex_lock(listlock);    // begin critical section
6     n->next  = head;
7     head =n;
8     pthread_mutex_unlock(listlock); // end critical section
9   }
```

この解決法では、我々は古典的なマナーによってロックを使っている。[2] 代わりに、我々にこのロックフリーマナーの、単純なコンペアアンドスワップ命令実行させてほしい。これが 1 つの可能なアプローチである。

```
1  void insert(int value){
2    node_t *n = malloc(sizeof(node_t));
3    assert(n != NULL);
4    n->value = value;
5    do{
6      n->next = head;
7    } while (CompareAndSwap(&head, n->next, n) == 0);
8  }
```

こちらのコードは次のポインタを現在の先頭を指すようにアップデートしており、それから新たに作られたノードを新たな先頭として位置するように入れ替えよう挑む。しかしながら、これは幾つかの他のスレッドが問題なく新たな先頭を入れ替えているときに失敗し、このスレッドがもう一度新たな先頭とともに実行することを引き起こす。

もちろんのこと、使いやすいリストを構築するにはリスト挿入のみならず、そして驚きがないことにあなたが中に挿入できるリストを構築し、削除をし、そしてロックフリーマナーが自明であるように振る舞うことが求められる。ロックフリーとウェイトフリーシンクロナイゼーションをもっと学ぶには、多量な文章を読むこと [H01, H91, H93]。

---

[2] 鋭い読者は何故我々がロックをそんなに遅く獲得するか疑問に思うかもしれないが、`insert()` に入るときは真に代わる…あなたは理解できるだろうか、鋭い読者さん、これが何故ほぼ正しいのか説明できるだろうか？どんな仮説がこのコードを形成するのだろうか…例えば `malloc()` のようなものを呼び出すためには？