Lock-free 入門

熊崎宏樹

平成 31 年 12 月 17 日

1　初めに

スレッドやプロセス間でリソースを共有するためには、mutex や semaphore を利用してリソースを保護する手法 (ロック) が一般的です。しかし、並列に操作しようとするスレッドやプロセスの数が増える程に、ロックされた区間 (クリティカルセクション) の実行時間が全体の計算時間を占める割合が高くなってしまいます。例えば目的処理の 95%の理想的なマルチスレッド化に成功したとしても、20 コアのプロセッサを用いた場合でせいぜい 10 倍速、60 コアのプロセッサを持ち出しても 15 倍速にしかなりません。これはアムダールの法則 [1] と呼ばれるもので、並列処理の黎明期から問題視されてきました。

更に悪いことに、ロックを取ったクリティカルセクション内のスレッドにもキャッシュミスや TLB ミス、コンテキストスイッチや IO 待ちなどの要因で遅延が発生します。この遅延が全体に与える影響はスレッドやプロセスが増える程に悪化します [1]。

この問題に対処するには、できる限りクリティカルセクションを小さくすることが肝要です。

`Non-blocking` アルゴリズムは、クリティカルセクションを最小化する事で上述の問題を解決したアルゴリズムです。本稿では `Non-blocking` アルゴリズムの分類を解説し、それぞれに分類される Stack 構造の実装例を通じて理解を深め、ベンチマークを行って比較を行います。

対象読者　初めに を読んで、「`Lock-free` という言葉はよく聞くけれど、`Non-blocking` アルゴリズムって何？」と思った方は、本稿をお読みください。本稿のソースコードは、読者のみなさんが逐次ステップを追えるように C 言語の疑似コードで記述しています [2]。本文やコメントで意味や意図を説明しますので基本的な文法の知識があれば大丈夫です。それではまず本稿で使う用語と、いくつかの CPU 命令、そして達成される特性の分類について紹介します。

2　並行・並列とは

まず、マルチスレッドやマルチプロセスに関する記事や論文を読むと必ず出てくるこの二つの用語から説明します。日常会話で並行・並列の明確な違いを意識することは少ないと思います。実際にこれらの単語を多少混同しても日常生活で問題になることは少ないでしょう。しかし、コンピュータサイエンスの研究の分野でこれらが使われる場合、どこに焦点を当てているのかがはっきり異なります。

[1] 例えば 100 人集まる会議に 1 分遅刻して全員を待たせた場合、のべ 100 分が無駄になるのと似てますね
[2] メモリの解放周りはややこしくなるので後述する URCU を利用しています

並行コンピューティング (Concurrent Computing)　コンピュータサイエンスにおいて、”並行”とは複数の計算タスク間で実行順序が定められておらず、同時に実行されうる系の性質を意味します。研究分野としての並行コンピューティングは、計算タスク間の相互作用に着目し、いかにこれを調停するかがテーマです。すなわち、計算タスク間で共有されるリソースをいかに問題なく安全に利用できるようにするかが並行コンピューティングの主な課題です。

調停の難しい複数のデータ構造操作やトランザクション処理を特定の条件を満たして実現する事が研究対象です。また、一つの資源を時分割多重などで切り分ける等により、複数の目標の達成を目指す事も並行コンピューティングの研究対象の一つです。その実現に際しパフォーマンスが低下しても、調停に成功するならばそれは一つの研究成果となります。

スレッド間の調停・分業が可能である事が自明な問題[3]はこの分野の研究対象とされません。本稿で扱う `Non-blocking` アルゴリズムはこの研究分野に属します。

並列コンピューティング (Parallel Computing)　複数の計算資源 (CPU や GPU のコア) を用いて計算を高速化する事に焦点をあてた研究分野です。多くの資源を注ぎ込み最終的にパフォーマンスが出るかどうかが最大の焦点となっています。つまりパフォーマンスが向上しなければ成果となりません。

達成したい目標が複数あるかどうかは関係なく、複数ある計算資源をいかに上手く使うかが研究対象です。並行コンピューティングとは違って、資源が 1 つしかない場合にはそもそも実現できません。

高いスループットを出すタスクスケジューリングや GPU や SIMD 命令を使ってパフォーマンスを高める研究、他にも Hadoop のように分散・並列化して高速化する研究などはこちらに分類されます。本稿では扱いません。

3　Non-blocking アルゴリズムの用語説明

Non-blocking アルゴリズムには通常のプログラミングではあまり馴染みのない固有名詞がいくつか登場します。まずそれらについて解説します。

3.1　アトミック操作

他のスレッドから途中経過を観測・干渉されずに行える操作をアトミック操作[4]と言います。一般に出回っている多くの CPU の 1 ワード幅[5]のメモリアクセスはアトミック操作です。例えば代入操作を行っている最中の変数の値は他のスレッドから観測されず、代入前か代入後の値のみが観測されます。

この説明では「何を当たり前の事を言っているのか？」と感じる方もいらっしゃるでしょうから、アトミックとならない例を挙げます。一度に 32bit 幅までのデータしか扱えない CPU にて 64bit 幅のデータを扱う場合、上位 32bit と下位 32bit に分割してそれぞれに操作を行う事になります。1 スレッドで操作を行っている分には単一の 64bit の値として観測されるため何も問題は起きませんが、他のスレッドから途中経過を観測された場合には問題が起きます。その例を図 1 に示します。初期状態として `0x0000000000000000` が代入されていた変数に、例えば `0xffffffffffffffff` を

[3]処理したいデータ間に依存関係がないため並列化が自明なもの

[4]哲学での「これ以上分割できない最小の単位である atom」が由来。

[5]CPU が扱う標準的なデータサイズの事。ポインタサイズや整数型のサイズの事を指す。最近の PC に載っている CPU は 1 ワード幅が 64bit とされることが多い。

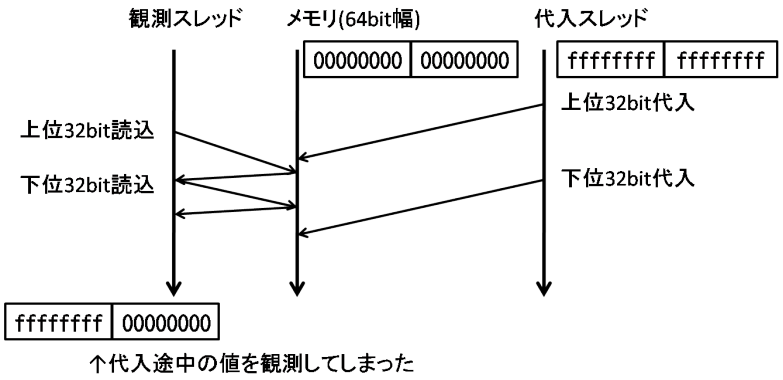

図 1: 変数の代入がアトミックとならない例

代入する場合を考えます。この時、代入される値は上下 32bit ずつに分けてそれぞれ操作されます。その途中で観測してしまったスレッドは `0xffffffff00000000` という代入前でも代入後でもない値を観測してしまいます。

この場合であっても 32bit 単位ではアトミックな操作であるため、例えば 16bit 地点で区切った `0xffff000000000000` のような値を観測することは絶対にありません。

3.1.1 Compare And Swap 命令

それでは、Non-blocking アルゴリズムを構築する根幹となるアトミック操作を二つ紹介します。まずひとつめは `Lock-free` の花形である Compare And Swap 命令 (以降では CAS 命令と略記) です。これは、「特定のメモリが指定した値と等しい場合に限り、別に指定した値へ書き換える」というアトミック操作です。疑似コードをソースコード 1 に示します。

ソースコード 1: Compare And Swap 命令の疑似コード

```
1 int CAS(void** target, void* expected, void* desired) /* ポインタを対象とした CAS の例
      */
2 {
3   if (*target == expected) { /* 指定したメモリが自分の希望する値だった場合に限り */
4     *target = desired; /* 別で指定した値を書き込む */
5     return 1; /* 成功 */
6   }
7   return 0; /* 失敗 */
8 }
```

x86 系の CPU では cmpxchg、SPARC 系の CPU では CAS というニーモニック [6] が用意されています。その倍の 2 ワード幅に対して CAS を行う DCAS 命令やそれを用いたアルゴリズムも研究されていましたが現在は主流ではありません。

Maurice Herlihy[2] の手によって、1bit 幅のアトミック操作を行う Test And Set 命令や、1 ワード幅で加算演算と値読み出しの組み合わせをアトミックに行う Fetch And Add 命令を駆使してもなお、3 つ以上のスレッド間で `Wait-free`(後述) に合意形成する事が不可能である事が証明され、それを実現可能な部品として CAS 命令が提唱されたという経緯があります。

[6] 機械語に対応するアセンブラコードの事

3.1.2 Load-Linked/Store-Conditional 命令

もうひとつが *Load-Linked/Store-Conditional* 命令 (以後は LL/SC と略記) です。これは疑似コードで書くとソースコード 2 のようになります。

ソースコード 2: Load-Linked/Store-Conditional 命令の疑似コード

```
1  void* load_linked(void** target)
2  {
3    marking(target, TID); /* 自分のスレッドIDでメモリをマーキング */
4    return *target;
5  }
6  int store_conditional(void** target, void* desired)
7  {
8    if(!is_marked(target, TID)){
9      /* 自分の付けたマークが消えていたら */
10     return 0; /* 失敗 */
11   }
12   *target = desired;
13   return 1; /* 成功 */
14 }
```

CAS とは異なり、*Load-Linked* と *Store-Conditional* の 2 つの命令をペアで用いる事でアトミックな書き換えを実現しています。いわゆる RISC(Reduced Instruction Set Computer) と呼ばれる CPU アーキテクチャでは原則として [7] 機械語の命令サイズを等しくする事で実装を簡略化しています。CAS 命令は「対象アドレス」「期待する値」「書き換えたい値」の 3 つの値を要求するため、命令サイズがどうしても大きくなってしまいます。一方で LL/SC 命令は LL 命令で「対象アドレス」の 1 つの値を、SC 命令は「対象アドレス」「書き込みたい値」の 2 つの値をそれぞれ要求するのみであるため、CPU アーキテクチャ上の実装が簡略になるという利点があります。

実装としては個別のメモリ領域に対してマーキングできる仕組みをハードウェア的に用意しており、そのメモリ領域に対し他スレッドが何らかの操作を行う事でマーキングが揮発してしまう仕組みになっています。このため、マーキングの消失を確認することで他スレッドによる割り込みの有無を判断できます。注意点として、マーキング可能なメモリ領域の粒度もそれぞれの CPU 実装ごとに異なることが挙げられます。例えば PowerPC アーキテクチャではキャッシュライン [8] をマーキング単位としています。通常 1 ワード幅以上の単位でマーキングが為されるため、False Sharing[9] も発生します。また、マーキングが揮発する割り込み操作の種類 [10] も CPU 実装ごとに異なります。

この LL/SC 命令を用いる事でいくつかのアルゴリズムを安全に ABA 問題 (後述) を招かず実装する事が可能となります。しかし LL 命令で付けたマーキングは本来消えるべきでないケースで消えてしまう事があります。それはマーキングした周辺の領域に対して他のスレッドが読み書きした場合や、マーキングしたスレッドがコンテキストスイッチした場合などです。こうした性質を偽陰性と呼びます。また、LL/SC 命令は CAS 命令を用いてエミュレーションすることが可能です (後述)。

[7] 例外として例えば ARM では半分の命令サイズの Thumb 命令セットがあります

[8] CPU が高速化のためにメモリをキャッシュとして切り出す際の最小単位。例えば PowerPC G4 では 32byte 単位。前述のワード幅とは無関係なので注意

[9] 共有していないメモリ間でも、同じキャッシュラインに載ってしまったがためにデータを共有している状況と同様のアクセス衝突に見舞われる状況

[10] メモリアドレスに対する store のみ揮発する場合や、load でも揮発する場合がある。

3.2 正確性

一般に、逐次実行されるプログラムの性質は正確に動いているかそうでないかの二元論で議論することが可能です。しかし並行プログラムは複数の操作の正確な順序が実行のたびに変動するので、並行プログラムの性質を議論する為に別の尺度が必要となります。ここではその際に広く使われる二つの尺度について説明します。

3.2.1 Safety(安全性)

一つめの尺度が Safety です。端的に言うと「危険な事が起きない」という意味であり、考えうるどのような実行パターンにおいても Segmentation Fault などの致命的な事故が起きない保証です。アルゴリズムの説明の文脈で「○○によって Safety が保証される」のように使われます。

3.2.2 Liveness(活性)

並行プログラムの正確性を示すもう一つの尺度が Liveness です。これは「操作が進行する」という意味であり、端的に言えばデッドロックしない保証です。アルゴリズムの説明の文脈で「○○によって Liveness が保証される」のように使われます。

Safety と Liveness は一見似ているように見えますが異なるものです。Safety はどんな事態でも状態が安全側に倒れる性質を指す一方で、Liveness はどんな事態からも正常状態へ復旧できる性質を指しています。

例えば銀行のデータベースにデッドロックするバグがある場合 (つまり Liveness が保証されていない) でも、取引の経過や預金の総額に矛盾が発生しないように設計されていれば Safety が保証されています [11]。

3.3 一貫性

並行でないプログラム、いわゆる逐次プログラムでプログラムが正しく動作しているかを証明するのは比較的単純です。なぜなら逐次プログラムでは実行順が毎回同じであるため、事前条件と挙動と事後条件がドキュメント通りであるかによって正しさを確認できるからです。簡単に言えば、テストをパスするかどうかでプログラムの正しさを確認できます。

しかし並行プログラムでは遥かに複雑になります。何故なら複数のメソッドが並行して行われる場合、並行の組み合わせパターンは膨大になるためです。そのためテストを一度パスしても、二度目はパスしないという状況が頻出します。それぞれの組み合わせパターンによって発生する結果が仕様通りであるかを人の手で網羅的に定義するのは現実的ではありません。

そのため、別の観点からプログラムの「正しさ」を検証/検定する必要があります。それが「一貫性」です。その定義は複数の階層に分けられます。

まずシングルスレッド (=逐次) で操作させた場合の挙動を、そのプログラムの振る舞いの基準として定義します。この定義が存在することによって「並行した操作は、何らかの逐次的な順序で実行されたものとみなせる」という比較が行えるようになります。

逐次的に実行されたかのようにみなすには、手続きの実行途中のオブジェクトが他のスレッドから観測されてはいけません。一見して至極順当な定義に見えますが、他の重要な定義が欠けている

[11] そしてデッドロックの度にマシンごと再起動する運用も不可能ではないでしょう

ためこれ単品で使われる事はありません。その重要な定義とは、どのタイミングで実行されるべきかです。例えばこの定義のみでは図 2 のようなケースを許容します。縦軸が時間軸を表しており、太線で書かれている部分は操作の開始と終了を表します。伸びている矢印は、関数呼び出しがどの瞬間に実行されたとみなすかという投影関係を表しています。この例ではオブジェクト x に対し、

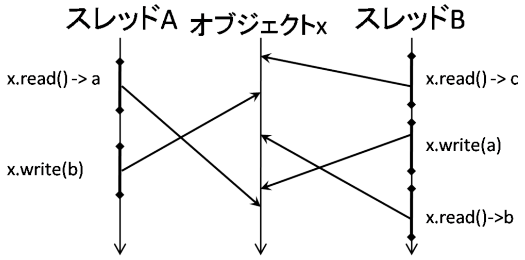

図 2: 何らかの順序で逐次実行されただけでは困る例

逐次順序にて「B による read」→「A による write」→「B による read」→「B による write」→「A による read」という順序での実行が為された状況を示しています。しかし実行順序の並び替えがこうも無秩序に行われてしまうと何も仮定できません。特に実時間での実行タイミングすら定義していないので、操作が 100 年後に実行される事すら許容してしまいます。

3.3.1 Quiescent Consistency(静止一貫性)

そこで定義された特性の一つが `Quiescent Consistency`(静止一貫性) です。定義は「静止状態 (Quiescent State) を跨ぐリオーダが起きない一貫性」というものです。定義のみでは意味を掴みにくいのでイメージを図 3 に示します。図中の水平な波線は「どのスレッドからの操作も行われていない瞬間 (=Quiescent State)」を表します。この波線を跨ぐリオーダが発生しない事が `Quiescent Consistency` の条件です。直感的には「波線と矢印が交差しない制約」という理解で正しいです。その制約さえ満たしていれば、図 4 のような順序反転も認めます。つまりプログラムが記述順に実行されるとは限りません。

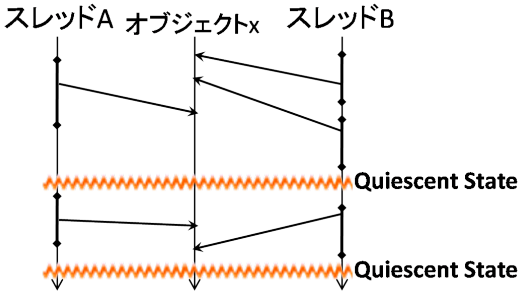

図 3: Quiescent Consistency の例

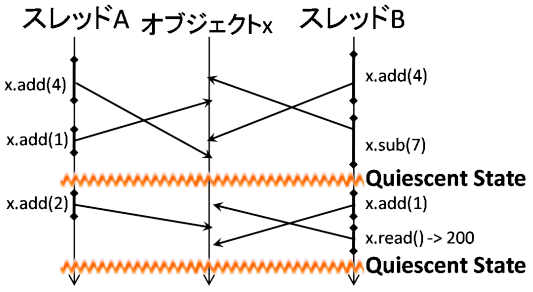

図 4: これでも Quiescent Consistency

終了時も含めて、どのスレッドも稼働していない Quiescent State においては、必然的にそれまでに開始された全ての操作が反映される事を意味します。そのように操作が反映された事を何らかのタイミングで保証できるため実時間性がある、とも言います。

この一貫性を採用したアルゴリズムの例としては、マルチプロセッサ環境用のスケーラブルなカウントアルゴリズムである Counting Network[3] や Diffracting Trees[4] などが挙げられます。これらは最終的なカウント値に矛盾が発生しなければ、個々のカウント操作の順が前後しても大丈夫という問題の性質を利用しています。

3.3.2 Sequential Consistency(逐次一貫性)

順序反転を許容できない状況のために定義された特性が Sequential Consistency(逐次一貫性) です。これは「観測される実行順序とプログラムの実行順序が等しい」というものですが、この定義のみでは意味を掴みにくいので図 5 に示します。

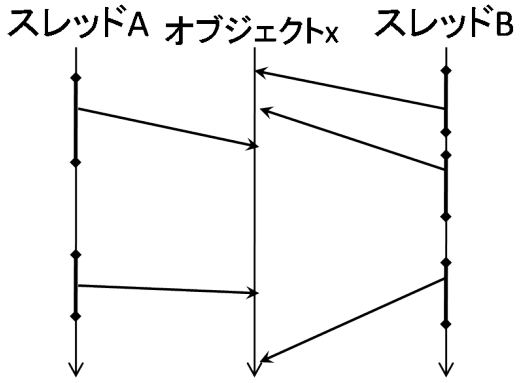

図 5: Sequential Consistency の例

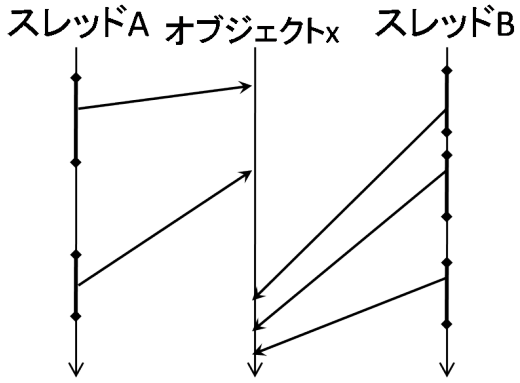

図 6: これでも Sequential Consistency

直感的には「矢印同士が交差しない」と言い換えることができます。これは、それぞれのスレッドに着目したとき、操作の実行順序が記述した順序と一致していることを表します。つまり図 6 のような状況であってもこの条件を満たしている事になります。この状況は例えば非同期実行を行った場合に発生します。

Sequential Consistency は Quiescent Consistency のように実時間性が無いため、仮にメソッド呼び出しが 100 年後に反映されようが実行順序の反転さえなければその特性を満たしている事になります。

Sequential Consistency と Quiescent Consistency は似た特性のように見えますが、実は直交しています。例えば、図 3 は Quiescent Consistency であり、たまたま Sequential Consistency も満たしています。しかし、図 4 は Quiescent Consistency ですが Sequential Consistency を満たしていません。また図 6 はたまたま Quiescent Consistency を満たしています。

順序の入れ替わりがないことが保証されるこの特性こそがパフォーマンスと扱いやすさのバランスを満たしているケースは現実の問題でも頻出します。例えばタスクのキューイングによって実行を非同期化し、並行制御の疎結合化を図る為に使われることがあります。

しかし、この特性には重大な欠点があります。それは、Composability[12] を満たさないという物です。ここでは「複数のオブジェクトに対する操作同士の順序反転は起きる」という事です。複数のオブジェクトに対する Read-Write は、非同期実行の環境下では古いデータに基づいた読み書きを行う事となり正しく動作しません。Sequential Consistency の欠点にまつわる議論に付いては

[12] ある特性を満たした物同士を組み合わせた時に、その合成物も同じ特性を満たすという性質。

書籍 The Art of Multiprocessor Programming[5] の 3 章や、Replication : Theory and Practice[6] の 1 章にて詳しく論じられています。興味のある方は覗いてみてください。

3.3.3 Linearizability(線形化可能性)

「あるオブジェクトに対する並行操作は、何らかの逐次的な順序で実行されたものとみなせる」という条件に加えて、「操作の開始から完了までのどこかの一瞬で実行されたと見なせる」ことが保証されている場合、Linearizability の特性を持っていると言えます。これは、同期実行であり、同一スレッド内での操作順の反転も発生しないという保証です。

図示すると図 7 のようになります。直感的には「それぞれの操作の開始と終了の時刻から横に

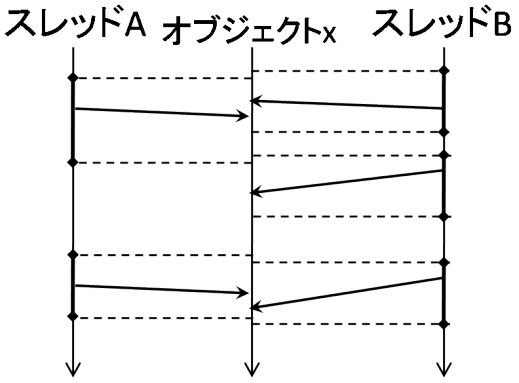

図 7: Linearizability の例

伸ばした点線と、その操作の矢印が交差しない」と言い換える事ができます。矢印が点線を跨がないという事は、どの関数も実行中でない瞬間 (Quiescent State) を跨ぐ矢印も存在しえません (=Quiescent Consistency の特性も満たしている)。この点線を跨がない限りはどのような実行順も許容されます。しかし実時間上で並行に実行されているメソッドのどちらが先に実行できるかが変化しうる (この場合はスレッド A と B の間の順序) のみであり、同一スレッド内での実行順序の並び替えは起きません (=Sequential Consistency の特性も満たしている)。

Linearizability は Quiescent Consistency と Sequential Consistency の両方を満たします。その包含関係を図 8 に表します。図 2 の居る領域は Sequential Consistency でも Quiescent Consistency でもありません。

Sequential Consistency を満たすのは図 5 と図 6 と図 3 と図 7 です。

Quiescent Consistency を満たすのは図 3 と図 4 と図 5 と図 7 です。

このように、一言に「正しい並行プログラム」と言った場合でも様々な一貫性レベルを保証する複数のアルゴリズムが考えられます。この魔導書で、他の著者の方々が紹介しているアルゴリズムそれぞれについて、どの一貫性保証を満たしているか考えてみる事は、アルゴリズムのより一層深い理解を導いてくれる事でしょう。

本稿ではこの中でも一番強い Linearizability を満たすアルゴリズムのみを扱います。

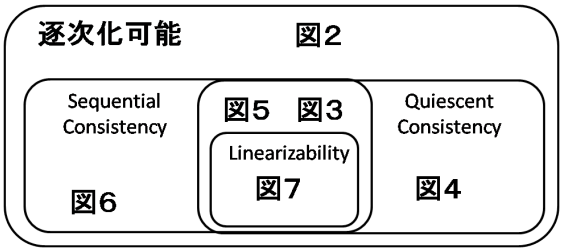

図 8: 一貫性の階層構造

4 Lock-freeって？

さて、必要な道具は揃ったので、いよいよ Lock-free の世界に飛び込んでいきましょう! 皆さんは Lock-free と聞いて何を想像するでしょうか。ロックのないプログラム？高速なプログラム？スケーラブルなプログラム？Lock-free という言葉はその響きがあまりに魅惑的な為にさまざまな誤解を招いています。そこでまずは Lock-free の世界がどうなっているのか俯瞰してみましょう。

Lock-free とは、Non-blocking アルゴリズムにおける、三段階の保証のひとつです。並行プログラムを研究する上で、アルゴリズムの挙動がスケジューラの思惑によって大きく変わる場合があることからそれらの分類が考案されました。残る二つの保証は、Wait-free と、Obstruction-free です。この 3 つは図 9 のような包含関係を成しており Wait-free であれば必ず Lock-free であり、Lock-free ならば必ず Obstruction-free です。逆は必ずしも成立しません。

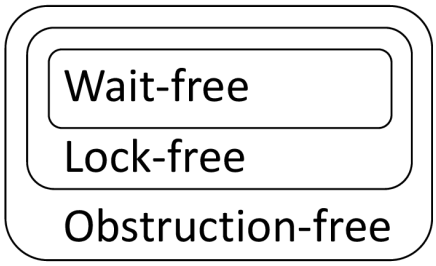

図 9: Non-Blocking アルゴリズムの位置づけ

さてスケジューリングとそれに対する挙動の変化無しには Lock-free は語れません。その点について説明します。

4.1 三つの保証

それでは、この三つの保証についてそれぞれ見ていきましょう。これらは複数のプロセスやスレッドが存在するシステムで、それぞれのタスクがどういった条件を与えられればどれほど完了するか、という保証を示すものです。性能については一切言及していません、つまりどんなに低速であっても Lock-free である場合もあります [13]。これらの保証の意味を理解する上では、**スケジューラが意思を持って進行を妨げようと工夫を凝らしてくる**という観点を持つことが理解の助けとなるでしょう。

[13] 一般には進行保証を強くするほどに速度は低下する事が多い

4.1.1 Obstruction-free

他の全てのスレッドが停止した状況で単独のスレッドに対してスケジューラから充分な時間が連続して与えられた場合に必ず操作が完了するという特性 (Non-blocking 特性とも言う) を持つとき、そのアルゴリズムを Obstruction-free と呼びます。例えば、会社で突然ある社員が仕事を途中で投げ出して退職するとしましょう、仮にどの社員が突然退職しても残った社員に**充分な時間が与えられる限り**各自の仕事を完了できるならば Obstruction-free です。これは自分の仕事が誰かの仕事の結果に依存している場合には満たすことができません。そのように Obstruction-free は「どのスレッドがどの瞬間でプリエンプションを食らっても全体が停止しない」という保証を意味します。ロックできなかったら他の手段を試みる前提で trylock を行う場合や、タイムアウトでアンロックする仕組みがあればロックを使ったアルゴリズムでも Obstruction-free の条件を満たす事ができます。

あるトイレの個室の鍵を内側から掛けたままの状態で、中にいた人が神隠しに遭った場合、そのトイレを外からこじ開ける手段がない限りそのトイレを使用するタスクはどれだけ CPU 時間を与えられても永久に実行不能であるため Obstruction-free ではありません。トイレの例と同じく、ロックを使用しているプログラムについて、どれかのスレッドがロック獲得 (　鍵を掛ける) に成功した時に、スケジューラがそのスレッドをプリエンプションし永久に CPU 時間を割り当てないようにスケジューリング (　神隠し) したとしましょう。このとき、それ以外のスレッドに何億クロック割り当てられようが全体で何一つ進行できなくなってしまいます。このようなプログラムは Obstruction-free ではありません。

図 10 に Obstruction-free の保証を図示しました。各矢印はスレッドを表し、縦方向が時間軸です。

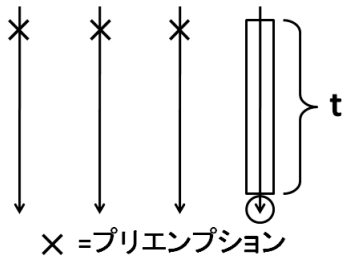

図 10: Obstrution-free の特性

×印がスケジューラによるスレッドのプリエンプションを表しています。◯印が何らかの手続きが成功した事を意味します。他のスレッドの進行を止めて、あるスレッドが単独で有限の時間 t の期間連続実行できれば必ず操作が完了する場合、Obstruction-free が保証されているといえます。考えうる全ての瞬間から「単独のスレッドで走るという前提さえあれば、有限のステップ数で自分の操作が完了できる」という事を証明することでこの保証を獲得できます。

Obstruction-free はロックを使っていない実装例が多いがために、「Lock-free だ」と誤解されやすいですが、Lock-free は後に説明する**進行保証**という大事な条件を満たす必要があります。Obstruction-free のみしか保証されていないアルゴリズムに対し、スケジューラはどのスレッドの連続実行も許さないようなスケジューリングを行なった場合、全体が停止 (Live-lock) してしまう恐れがあります。Live-lock というのはどのスレッドも自分の操作を進行できない状況の事を指しています。Dead-lock も同様ですが、Live-lock は「スケジューラの采配次第で無限に続

いたり即抜け出せたりする (= Liveness が保証されている)」のに対し、Dead-lock は「ひとたび陥ったらスケジューラが何をしようと再起不能 (= Liveness が保証されていない)」という違いがあります。

4.1.2 Lock-free

スケジューラからどのような実行順制御をされようと、CPU 時間が割り当てられる限り、どれかのスレッドが必ず有限ステップ内に操作を完了するという特性を持つ時、そのアルゴリズムは Lock-free です。みんな大好きですね。言い換えると Obstruction-free の保証に加えて、**進行保証 (*progress guarantee*)** のついたものが Lock-free という保証です。**進行保証**がどういうものかというと「どんなスケジューリングをされようが必ずどれかのスレッドの操作が成功する」という保証です。その「どんなスケジューリング」の中に「特定のスレッドだけに充分な時間を与えるスケジューリング」も含まれるので、Lock-free ならば必ず Obstruction-free の条件も満たします。前述の通り、Obstruction-free では Live-lock が発生していましたが、その Live-lock が発生しないことを証明したものが Lock-free です。Dead-lock と Live-lock の両方の Lock を避ける事から Lock-free と呼ばれます[14]。典型的なパターンとして、CAS スピンという技法を使ってコードを書いた場合に Lock-free の特性を満たします。その CAS スピンの例をソースコード 4 に示します。

ソースコード 3: 素直に実装した数え上げ

```
1  void increment(int* counter)
2  {
3    *counter += 1;
4  }
```

ソースコード 4: CAS スピンを使った数え上げ

```
1  void increment(int* counter)
2  {
3    int old_count, new_count;
4    for(;;) {
5      old_count = *counter;
6      new_count = old_count + 1;
7      if (CAS(counter, old_count,
           new_count)) {
8        /* カウンタの更新に成功 */
9        return; /* 操作完了 */
10     }
11     /* 初めからやり直し */
12   }
13 }
```

ソースコード 3 では与えられたポインタの指す値をインクリメントする操作を示しています。しかしこの実装では、複数のスレッドが同一の値をインクリメントする際に値がズレてしまいます。何故ならこの操作は CPU の内部では「ポインタの値を読み出す」「書き込みたい値を算出する」「書き込む」の 3 ステップに別れて操作しているため、複数のスレッドの操作が割り込みあった際に古い値を書き込んでしまう事があるからです。

それを CAS 命令を使って回避した例がソースコード 4 です。複数のスレッドが並行してこの関数を呼び出した場合でも、7 行目での CAS が成功した場合には「ポインタの値を読み出す」から「書き込む」の間で他のスレッドによる割り込みが発生していない事を保証できます。CAS が失敗した場合には割り込まれた事を検知し、初めからリトライします。

この操作は Lock-free に分類できます。Lock-free であるためにはまず Obstruction-free である必要があります。Obstruction-free である為には「他のスレッドをプリエンプションして自スレッドだけを充分な時間実行した場合、**必ず**終了できる」という特性を持つ必要があります。このアルゴリズムの場合、他のスレッドがどのような状態で停止していても、自スレッドが操作を続行できる事が明確であるため Obstruction-free です。

[14] Lock-free がこう定義されたのは 2003 年頃の話で、それ以前は Non-Blocking と混同されていた

次に、Lock-free であるためには進行保証が必要です。つまり、他のスレッドがどのようなタイミングで操作を中断したり割り込んできても、どれかのスレッドが目的を達成するという保証が必要です。CAS スピンの場合それは簡単に証明できます。

まず CAS に失敗するには、その「期待する値」が希望した値と異なる必要があります。期待した値と異なるということは、他のスレッドが割って入ってその値を書き換えたという事です。この操作では、カウンタの値を書き換える事がそのまま成功を意味するため、その割って入ったスレッドは値の書き換えに成功、つまり操作に成功しています。

逆に期待した値と同じであれば自分のスレッドの操作は成功します。つまり CAS の成功・失敗のどちらに転んでも、どれかのスレッドの操作が成功しているため、進行を保証できます。

繰り返しますが、Lock-free で大事なのはロックを使わない事ではなく**進行保証**の有無です。一方で「どれかのスレッドが成功する」までしか保証しないため、どれかのスレッドだけが永久に操作に成功しない飢餓状態 (*starvation*) が発生していようと Lock-free の条件は満たせます。

図 11 に Lock-free の特性を図示しました。プリエンプションがいかに激しく行われようと、全

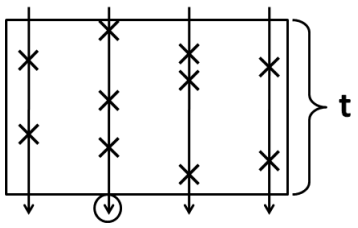

図 11: Lock-free の特性

体として CPU 時間 t が与えられている状況で一つでも手続きが成功するならば Lock-free の条件を満たします。スケジューラの立場になってみると、どのようなスケジューリングを行っても CPU 時間を与える限りは絶対にアルゴリズムの進行を止める事ができなくなります。

4.1.3 Wait-free

スケジューラからどのような実行順制御をされようと、CPU 時間が割り当てられる限り、全てのスレッドが必ず有限ステップ内に操作を完了するという特性を持つ時、そのアルゴリズムは Wait-free です。言い換えると Lock-free の保証に加えて、全てのスレッドの操作完了までの必要 CPU 時間を保証できるものが Wait-free という保証です。これは starvation という弱点を克服した Lock-free とも言えます。スケジューラの立場から見てみると「ありとあらゆる嫌がらせスケジューリングを行っても CPU 時間を割り振る以上は全てのスレッドがいつか成功してしまう」という状態です。

一見凄そうに見えますがこれをまともに実装するにはかなりの労力を必要とし、完成してもパフォーマンスが出るとも限らず、実際の所これほど強力な保証が必要な場面はあまりありません。音響システムやロボットやネットワーク機器などハードなリアルタイム性が必要なアプリケーションなどで使われる事があります。

他にも、ヘテロジニアス共有メモリ環境 (例えば CPU と GPU で同じメモリマップを共有する環境) では、Lock-free でアルゴリズムを実装すると、メモリの速度差のために一方が負けつづけてしまう (Starvation が発生する) ため、Wait-free 化することがあるそうです [7]。現実では、Starvation が深刻な問題となるケースは限られるため、この特性が要求される事は稀です。

図 12 に Wait-free の特性を図示しました。プリエンプションがいかに激しく行われようと、スレッドに CPU 時間 t が与えられている状況で、必ず全ての手続きが成功するならば、Wait-free の条件を満たします。starvation は発生しません。

5 それぞれの特性を満たす Stack の実装

ここまでの章では、Non-blocking データ構造の分類とそれぞれの保証について説明しました。この章では、汎用的なデータ構造の一つである Stack を例に、それぞれの保証を満たすように作り分けます。

5.1 普通の Stack

それでは並行 Stack の実装を通じて、Non-blocking アルゴリズムの詳細を追っていきましょう。アルゴリズムの説明を主とするため、実装するインターフェースは push() と pop() のみに絞ります。また、説明は (この Stack 実装について) アルゴリズム構築が容易な順、Lock-free、Obstraction-free、Wait-free の順で行います。それでは、まず基本になる Stack の実装をソースコード 5 に示します。

push(Item) によってアイテムが挿入され、pop() によってアイテムを取り出せます。何の変哲もない Stack 構造ですが、複数のスレッドからアクセスされるとデータ構造が破損する可能性があります。

ソースコード 5: 素直に実装した逐次 Stack 実装の例

```
1  typedef struct StackNode_ {
2    Item data; /* 何らかのデータ型 */
3    struct StackNode_* next;
4  } Node;
5
6  Node* head = NULL; /* スタックの先端 */
7
8  void push(Item new_item)
9  {
10   /* ノードを初期化 */
11   Node* const new_node = (Node*)malloc(sizeof(Node));
12   new_node->data = new_item;
13   new_node->next = head; /* head が指しているのが Stack の先端 */
14
15   /* Stack の先端を更新 */
16   head = new_node;
17 }
18
19 int pop(Item* popped)
```

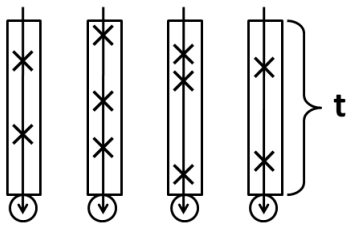

図 12: Wait-free の特性

```
20 {
21   /* スタックの先端のノードのItemを取得 */
22   Node* const old_head = head;
23   if (old_head == NULL) {
24     /* スタックが空なら失敗 */
25     return 0;
26   }
27
28   /* 取得したアイテムを代入 */
29   *popped = old_head->data;
30   free(old_head);
31
32   /* Stackの先端を更新 */
33   head = head->next;
34   return 1;
35 }
```

push(item) によってアイテムが挿入され、pop() によってアイテムを取り出せます。Stack が既に空だった場合には pop() は失敗し 0 を返します。pop() に成功した場合には 1 を返しポインタで指定したアドレスにデータを格納します。何の変哲もない Stack 構造ですが、複数のスレッドから同時に操作した場合に破損します。

これからこの実装をベースラインの逐次実装として、3 種の進行保証をそれぞれ満たした並行実装を紹介していきます。

5.2 Lock-free Stack

まずはこの Stack の Lock-free 実装を紹介します。

head ポインタの更新に CAS 命令を利用することで調停できます。これは前述した CAS スピンです。まずは push() の実装をソースコード 6 に示します。

ソースコード 6: Lock-free Stack 実装の push()

```
1  void push(Item new_item)
2  {
3    /* 新しいノードを作る */
4    Node *old_head, *new_node = (Node*)malloc(sizeof(Node));
5    new_node->data = new_item;
6
7    for (;;) {
8      old_head = head;
9
10     /* 線形リストを構成するためnextを更新 */
11     new_node->next = old_head;
12
13     /* headを更新 */
14     if (CAS(&head, old_head, new_node)) {
15       /* 成功したので終了 */
16       break;
17     }
18   }
19 }
```

push() 時の挙動について図 13 に示します。A B C…は Stack に積まれたノード、実線の矢印の示す先は現在 head ポインタが指しているノード、点線の矢印の示す先は head ポインタの更新成功時に head ポインタが指すノードです。

図 14 のように二つの push() が衝突した場合でも、CAS 命令がこれを調停するので、図 15 のように、必ずどちらかのスレッドが成功し、もう片方のスレッドが失敗します。失敗したスレッドは push() を再度行うので、最終的に図 16 のようにすべての操作を線形化できます。

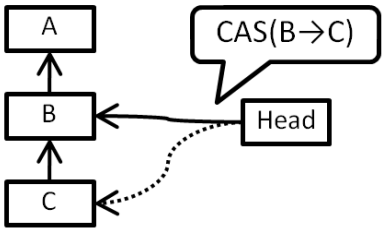

図 13: C を push()

pop() の実装をソースコード 7 に示します。push() と同じく、head ポインタの更新に CAS 命令を利用し、CAS スピンを形成します。これにより、競合が発生しても必ず一つのスレッドしか同時には成功しないため、こちらも最終的にすべての操作が線形化できます。

ソースコード 7: lock-free stack 実装の pop()

```
1  int pop(Item *popped)
2  {
3    Node *old_head, *next_head;
4    while (1) {
5      old_head = head;
6
7      /* スタックが既に空ならNULL を返す */
8      if (old_head == NULL) {
9        return 0;
10     }
11
12     /* 線形リストの先頭をnext へ */
13     next_head = old_head->next;
14
15     /* head を更新*/
16     if (CAS(&head, old_head, next_head)) {
17       /* 成功したので終了 */
18       break;
19     }
20   }
21   *popped = old_head->data;
22   free(old_head);
23
24   /* 成功 */
25   return 1;
26 }
```

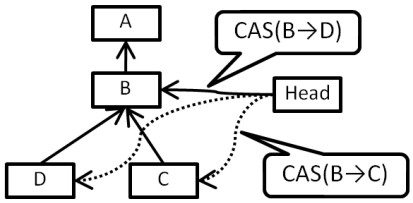

図 14: C と D の push が衝突

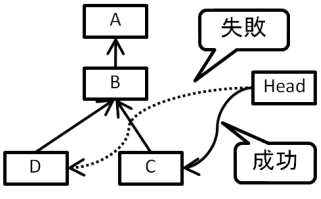

図 15: C が成功して D が失敗

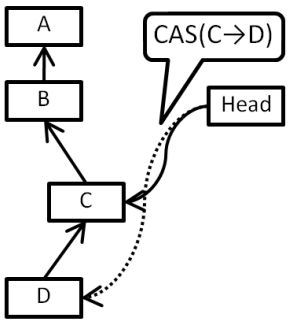

図 16: 再度 D の操作を行う

こちらも CAS スピンの形で実装され、図 17 のような挙動をします。

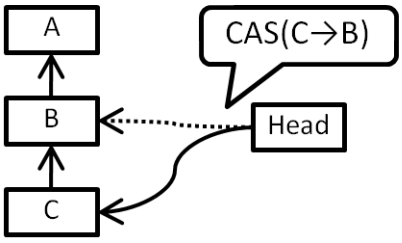

図 17: C を pop()

Lock-free 性　この Stack は push() もしくは pop() が成功する際必ず head ポインタの指す場所が変わります。この実装では CAS 命令にて head ポインタの書き換えに成功すれば push() 及び pop() の操作に成功します。つまり、操作に失敗する場合には必ず他のスレッドが操作に成功した事を意味するため、進行保証があります。しかし、CAS スピンによる実装であるため、前述した通り `starvation` は発生します。

ABA 問題　実はこの `Lock-free Stack` には重大なバグがあります。それは `ABA` 問題というバグです。具体的な発生順序を図 18 に示します。

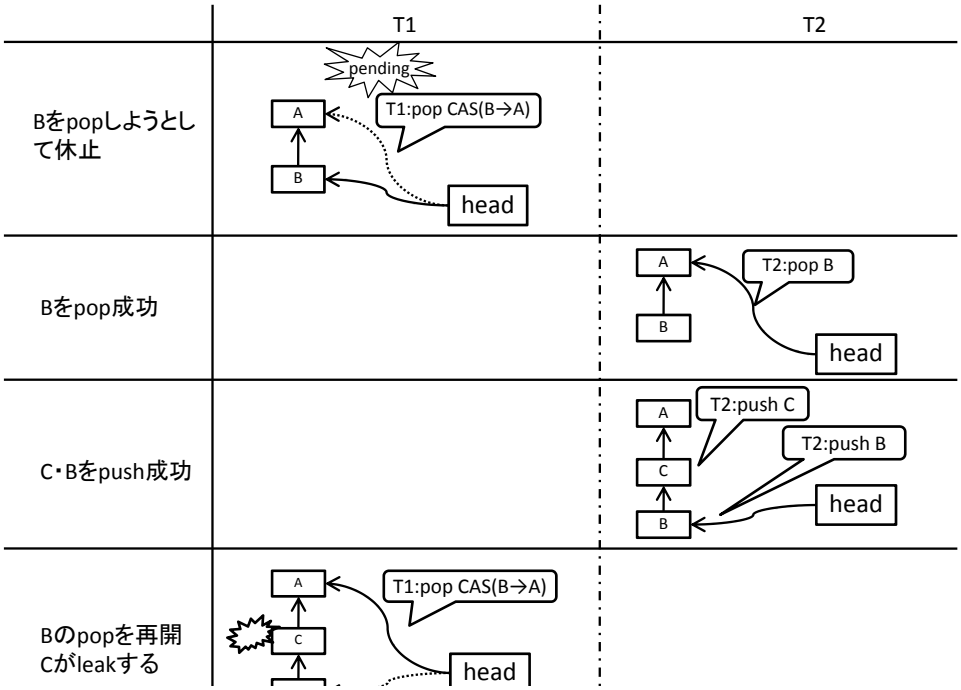

図 18: Lock-free Stack で ABA 問題が発生する例

スレッド T2 にて「B を pop()&メモリ解放」→「C をメモリ確保&push()」→「B をメモリ確保

&push()」という操作を行ってしまったがために、スレッド T1 が「A なら B へ CAS しよう」と準備していた所と重なってメモリリークという事故が発生します。この状況のように一旦別のもの (A → B) になったのに、偶然元に戻されてしまった (B → A) 場合、CAS 命令ではこれを検出できないためデータ構造に齟齬が生じる場合があります。これを ABA 問題と呼びます。この例では C を飛び越して A にアクセスしたため、メモリリークが発生しています。場合によっては解放済みのアドレスに触れてアクセス違反を引き起こす可能性もあります。この ABA 問題を回避する為の手段は複数提案されており、その一部を紹介します。

- そもそも同時に pop() が走る事が問題なので、pop() だけ同期プリミティブを用いて排他制御を行えば安全になります。2 つ以上のスレッドが同時に pop を行えなくなる一方で push() 同士は依然として排他制御をせず行えるため、目的を達成するためには充分な場合もあります。pop() ではなく push() を mutex で保護する戦略でも同様の目的は達成できます。しかし排他制御を用いる以上、その操作の Non-Blocking な特性は失われます。

- そもそも他のスレッドが参照中のノードを使いまわすのが悪いので参照中は使いまわされないように工夫するという戦略もあります。ガベージコレクタ (GC) のある言語なら何もせずとも参照中は再利用されないので心配は要りません。先ほどの例で言えばスレッド T1 が old_head としてノード B を参照中なので再利用されません。そのためスレッド T2 はその解放されていないメモリを新規ノードとして再利用することは起きず、T1 の CAS は正しく失敗します。GC のない言語でそれと同等の効果を得る手法は数多く研究されていますが、本筋から少々逸れるので章末のコラム「局所的な GC」にまとめます。本稿の最後ではベンチマークを行いますが、その際には局所的な GC の一種として使える UserspaceRCU を用いてメモリ解放タイミングの問題を解決しています。

- そもそも malloc がたまたま同じ領域を確保する可能性がある点が問題なので、変更を示すタグとともに管理すれば回避できます。その一例として、head ポインタの一部をカウンタとして利用する方法が挙げられます。ただしこの方法では、カウンタが一周してしまった場合に運悪く衝突する可能性があります。そのため、ABA 問題を完全には回避していませんが、実用上問題のないレベルまで改善することは可能です。

- CAS での事故的な一致が不味いのなら LL/SC 命令を使えば ABA 問題に陥る心配はありません。しかしその場合は、LL/SC 命令の偽陰性の影響で進行保証が得られません。そのため、Lock-free ではなくなってしまいます。

このように、いずれも一長一短で、汎用的な解決方法がないのが現状です。

5.3 Obstruction-free Stack

Obstruction-free の条件を満たし、Lock-free でない Stack の実装を考えてみます。言い換えればどんな状態であっても自スレッドに充分な時間が与えられた場合には必ず操作を完了できる必要があり、Live-Lock の可能性もある Stack です。LL/SC を用いる事でそのような状態になります。

LL/SC のエミュレーション　LL/SC 命令の説明の所で軽く触れましたが、CAS 命令で LL/SC 命令をエミュレーションする事ができます。その例をソースコード 8 とソースコード 9 に示します。

LL/SC 命令をエミュレートするために、スレッドごとにポインタの一時保管場所を用意し、他のスレッドから参照できるようにします。まずその下準備をしているコードをソースコード 8 に示します。

Lock-free の時とは別の箇所で ABA 問題が発生するため、カウンタを保持 (14 行目) して ABA 問題を回避しています。この詳細については後述します。

ソースコード 8: CAS 命令を用いた LL/SC 命令のエミュレートの準備

```
1  /* 各スレッドが個別に保持するID で 1 から始まる連番の整数 */
2  __thread intptr_t TID;
3
4  /* 数値も格納可能なポインタ */
5  /* union なので 1 ワード幅に収まっている */
6  typedef union markable_ptr {
7    Node* ptr; /* ポインタとして利用するとき */
8    intptr_t tid; /* スレッドID として利用するとき */
9  } MarkablePtr;
10
11 /* スレッドごとに保持するワーキングセット */
12 typedef struct {
13   Node* node; /* 退避したデータを置く場所 */
14   intptr_t counter; /* そのスレッドが持つカウンタ */
15 } WorkSet;
16
17 /* スレッド間で共有するObstruction-free Stack の本体 */
18 /* 初期値はhead == NULL であれば良い */
19 typedef struct {
20   MarkablePtr head; /* スタックの先頭 */
21   WorkSet workingset[THREAD_MAX]; /* データを退避する場所 */
22 } of_stack;
23
24 /* 渡されたmarkable_ptr が tid かどうかを判断するもの */
25 int is_tid(MarkablePtr address) {
26   /* 1bit 目が立っている場合に、MarkablePtr は tid を保持しているとする */
27   /* tid の値そのものは 2bit 目から使う */
28   return address.tid & 1;
29 }
30
31 /* MarkablePtr が tid を保持している際、10bit 目以上の部分でカウンタを持つ */
32 /* そのカウンタを除いてビットシフトした状態のものを獲得する */
33 int get_tid(MarkablePtr address) {
34   if (!is_tid(address)) {
35     /* tid ではなかったので番兵を返す */
36     return 0;
37   } else {
38     /* tid だったので獲得した tid を返す */
39     return (address.tid & ((1 << 10) - 1)) >> 1;
40   }
41 }
42
43 /* カウンタ付きtid を生成する */
44 intptr_t generate_tid(int tid, intptr_t count) {
45   return marking = 1 /* 最下位ビットでunion が TID を保持していることを示す */
46     | (TID << 1) /* 自分のスレッドID を埋め込む。最下位ビットを潰さないようシフト */
47     | (count << 10); /* カウンタを 10bit 以上の場所に埋め込む */
48 }
```

union を用いてポインタとスレッド ID のどちらも格納できる 1 ワードの領域を定義します (MarkablePtr)。その領域が保持しているものがポインタなのかスレッド ID なのかを判別する関数である is_tid() を用意します。

union がポインタとスレッド ID のどちらを格納しているかを区別するために特殊なテクニックを使っています (25 行目以降)。一般に malloc で確保されるメモリ番地は CPU や計算効率の都合上、4 の倍数や 8 の倍数といった偶数番地のみであるため、それを利用して偶数→ポインタ、奇数

→スレッド ID という区別をしています。

次は実際にそれらを使って Load-linked と Store-conditional を実装してみます。

ソースコード 9: CAS 命令を用いた LL/SC 命令のエミュレート

```
1  /* 他人の付けたマーキングを消しながら読むread */
2  Node* read(of_stack* stack) {
3    while (1) {
4      MarkablePtr old_head = stack->head;
5      if (is_tid(old_head)) {
6        const intptr_t tid = get_tid(old_head);
7        /* addressがスレッド IDを保持していた */
8
9        /* そのスレッドが退避していたデータを読みだして */
10       Node* next = stack->workingset[tid].node;
11
12       /* tidを指した状態からポインタを指した状態へ戻す */
13       if (CAS(&stack->head, old_head.tid, next)) {
14         /* 成功したら値を返す */
15         return next;
16       }
17     } else {
18       return old_head.ptr;
19     }
20   }
21 }
22
23 /* Load-Linked命令 */
24 Node* load_linked(of_stack* stack) {
25   /* CASループ */
26   while (1) {
27     /* 他のスレッドのマーキングを消しながら読む */
28     Node* target = read(stack);
29
30     /* 自スレッドの退避場所にコピー */
31     stack->workingset[TID].node = target;
32
33     /* カウンタを埋め込んだTIDを生成 */
34     intptr_t my_tid = generate_tid(TID, stack->workingset[TID].count);
35
36     /* 自分のスレッドIDをポインタの代わりに書き込む */
37     if (CAS(&stack->head, target, my_tid)) {
38       /* マーキング成功時はカウンタの値を増やしてABAを回避 */
39       ++stack->workingset[TID].count;
40       return target;
41     }
42   }
43 }
44
45 /* Store-Conditional命令 */
46 int store_conditional(of_stack* stack, Node* new_value) {
47   /* 自分の付けたマーキングが残っている場合に限り成功するように */
48   /* カウンタを埋め込んだTIDを生成する */
49   /* Load-linked成功時にカウンタを増やしてしまったのでここでは 1減らす */
50   const intptr_t expect_tid =
51     generate_tid(TID, stack->workingset[TID].count - 1);
52   return CAS(&stack->head, expect_tid, new_value);
53 }
```

MarkablePtr がスレッド ID を格納している時、MarkablePtr はそのスレッド ID のスレッドによってマーキングされている事を意味します。

指定された MarkablePtr にそれまでに入っていたポインタを自分の一時保管場所に移し、自分のスレッド ID を保持するよう書き換える (つまりマーキングする) のが load_linked() 関数です。

指定された MarkablePtr がポインタを保持するよう書き換え (つまりマーキングを消去し)、そのポインタ値を返すのが read() 関数です。

store_conditional() は自分のスレッド ID によるマーキングが消えてない状態に限り保存に成功する関数です。

CAS 命令を用いる事でどれも簡潔に実装できます。少しややこしいので図解を示します。図 19

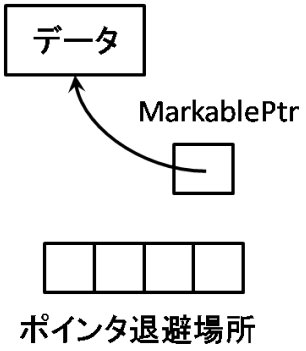

図 19: 初期状態

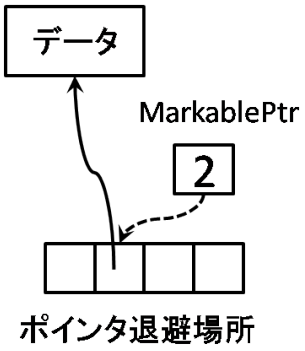

図 20: Load-Linked に成功

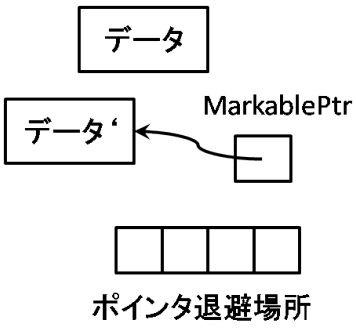

図 21: Store-Conditional に成功

は MarkablePtr とポインタ退避場所の位置関係を表しています。MarkablePtr は何らかのデータをポインタを用いて指しています。

図 20 は図 19 に対し load_linked() 関数を実行した後の状態を示しています。この図ではスレッド 2 番がマーキングに成功しました。具体的には CAS(MarkablePtr, データ, 2); という操作が行われました。MarkablePtr はマーキングしたスレッドの ID である 2 という整数を保持し、元々保持していたポインタはポインタ退避場所が保管しています。read() 関数は MarkablePtr を読み出し、スレッド ID を保管していた場合には再び図 19 の状態へと戻して、ポインタの値を返すものです。

図 21 は図 20 に対し store_conditional() 関数を実行した後の状態を示しています。具体的には CAS(MarkablePtr, 2, データ'); という操作が行われました。図 19 の状態からこの状態に至るまでの間に他のスレッドに read() 及び load_linked() が発行されていない場合に限り成功します。

実際にはこの説明の通りに実装すると ABA 問題に直面してしまいます。具体的に危険な ABA 問題の例を挙げます。

- スレッド 1 がデータ A を Load-Linked して読み出す。データ A をスレッド 1 の退避場所に移し、MarkablePtr には 1 としてマーキング
- スレッド 2 が MarkablePtr に書かれた 1 を読み出し、スレッド 1 の退避場所からデータ A を獲得。CAS(&head.tid, 1, A) を行う直前でプリエンプション
- スレッド 1 が Store-Conditional に無事成功しデータ A は消失。次の操作としてデータ B を Load-Linked して読み出し、データ B を退避場所に移し、MarkablePtr には再び 1 としてマーキング
- スレッド 2 がプリエンプションから再開し、CAS(&head.tid, 1, A) に成功してしまう。既に消失したデータ A を読みだしてしまいエラー

マーキング同士のバージョンが比較できないことが ABA 問題の根源となっています [15]。そのため、MarkablePtr に TID を保持する時は、TID と共にそのスレッドが過去にどれだけ load_linked() に

[15] デバッグするの大変でした

成功したかというカウンタを上位ビットに埋め込んで保持します。これによって、同一のスレッドのマーキングを古い値で上書きしてしまう ABA 問題を回避しています。

このアルゴリズムは、自スレッドに充分な CPU 時間さえ連続で与えられれば他のスレッドがどの状態で停止しても read() → load_linked() → store_conditional() がそれぞれ成功するため Obstruction-free の条件は満たしています。しかし、複数のスレッドが load_linked() を発行し続けてお互いのマーキングを潰しあう事が考えられます。その状態は、どの操作も完了しない Live-lock 状態であり、進行保証がありません。

LL/SC を用いた Obstruction-free な Stack の実装　ここで実装した LL/SC を利用して実際に Obstruction-free な Stack を作ってみます。まず push() 関数をソースコード 10 に示します。

ソースコード 10: Obstruction-free Stack の push()

```
1  void push(of_stack* stack, Item item) {
2    /* 挿入したいノードを確保 */
3    Node* new_node = (Node*)malloc(sizeof(Node));
4    new_node->item = item;
5
6    while (1) {
7      /* LL/SC を使ったループ */
8
9      /* head ポインタを load-linked で読み出し */
10     Node* old_head = load_linked(stack);
11     /* 線形リストを構成するため next を更新 */
12     new_node->next = old_head;
13
14     /* head を更新 */
15     if (store_conditional(stack, new_node)){
16       /* 成功したら終了 */
17       return;
18     }
19   }
20 }
```

Lock-free 版のソースコード 6 と比べてもほぼ同等の簡潔さに収まっています。load_linked() で書き換え対象となる head ポインタを読み出し、store_conditional() で head を書き換えています。内部での挙動は LL/SC を使っている事を除いて Lock-free Stack と同様なので詳細は省略します。次に pop() をソースコード 11 に示します。

ソースコード 11: Obstruction-free Stack 実装の pop()

```
1  int pop(of_stack* stack, Item* item) {
2    while (1) {
3      /* head ポインタを load_linked で読み出し */
4      Node* old_head = load_linked(stack);
5
6      if (old_head == NULL) {
7        /* スタックが空ならそのまま終了 */
8        return 0;
9      }
10
11     /* 線形リストの先頭を次の要素に繋げる */
12     Node* next_head = old_head->next;
13
14     /* head を更新*/
15     if (store_conditional(stack, next_head)) {
16       /* 成功したら終了 */
17       *item = old_head->item;
18       return 1;
19     }
20   }
21 }
```

こちらも Lock-free 版のソースコード 7 と比べても同等の簡潔さに収まっています。push と同様に load_linked() で書き換え対象となる head ポインタを読み出し、store_conditional() で head を書き換えています。

Obstruction-free 性　このアルゴリズムは他のスレッドがどの状態で停止していても自スレッドの操作を有限ステップしか妨げられないので Obstruction-free です。しかし前述のように複数のスレッドが交互に load_linked() を発行した際に Live-lock 状態に陥るため進行保証はありません。そのため Lock-free の保証は無く、Obstruction-free です。

5.4 Wait-free Stack

これまでは Lock-free や Obstruction-free という、比較的弱い進行保証による実装例の紹介をしてきましたが、これらより強い進行保証である Wait-free の条件を満たした Stack も実装可能です。そこでここでは満を持して Wait-free Stack の実装を紹介します。

Wait-free の条件を満たすためには Lock-free までの条件に加えて starvation の撲滅を果たす必要があります。言い換えれば「全てのスレッドの手続きの開始から終了までに必要なステップ数の上限が有限」である必要があります。それは「どんなに酷いスケジューリングされようがこれだけの CPU 時間を与えられれば必ず自分の操作を完了できる」という事を意味します。

このような条件を満たすのは困難そうに見えますが、実は Lock-free での CAS スピンと同様に、Wait-free での汎用パターンがあります。

StateArray　Lock-free の条件を満たしていて Starvation が発生する状態とは、後発のスレッドに CAS で負け続けている状態を意味します。つまり後発のスレッドに追い抜かれないようにする事で Wait-free の条件を満たすことが可能となります。これを実現する方法の一つが StateArray です。

大まかな挙動は FIFO に似ています。各スレッドは自分より先に操作を開始したスレッドが必ず先に終わるように、自分の操作の開始前に他スレッドが操作途中でないかを確認します。もし操作途中のスレッドがいた場合には先にそれを手伝います。

この仕組みにより、どのようなスケジューリングをされようと、先に操作を開始したスレッドの操作が先に完了するので、操作時間の上限を証明できます。

これから Wait-free Stack での実装例を示します。見本にしているのは Wait-free Queue の論文 [8] です。

ソースコード 12: Wait-free Stack 実装の例：データ定義

```
1  /* Stack のノード */
2  typedef struct node {
3    Item* item; /* 保存対象のデータ */
4    int winner; /* 操作を進行する権利を獲得したスレッドを指す */
5    struct node* next; /* 線形リストとしての next */
6    struct node* prev; /* pop 時に old_head を獲得するための prev */
7  } Node;
8
9  /* StateArray のノード */
10 typedef struct {
11   uint64_t count; /* 単調増加するカウンタ */
12   int pending; /* 操作が完了したかどうかを示すフラグ */
13   int is_push; /* この操作が push なのか pop なのかを示すフラグ */
14   Item* item; /* 操作対象のデータ */
```

```
15 } State;
16
17 /* Wait-free Stackの情報 */
18 typedef struct wf_stack {
19   Node* head; /* 線形リストの先頭 */
20   State* states[THREAD_MAX+1]; /* [0]は未使用を表す為に使うのでスレッド+1の数だけ用意する。*/
21 } wf_stack;
```

ソースコード 12 にデータ構造の定義を表します。

Node Lock-free Stack の時と比べて Stack のノードに winner と prev の要素が追加されています。winner を CAS によって書き換える事で次に操作を完了できるのがどのスレッドなのかを指定します。prev については後述します。

State State はスレッドの作業内容と進行状態を保持します。is_push() は操作が push() なら 1、pop() なら 0 に設定します。count は更新される度にインクリメントされる作業番号です。pending はそのスレッドの操作が完了したかどうかを表します。完了時に 0 を設定します。item は操作対象のデータを表します。push なら push したいデータ、pop なら pop によって獲得できたデータを保持します。

この State 構造体一つの中に、一つのスレッドの操作に必要な情報が揃っているので、他のスレッドからでも操作を手伝う事ができます。この State が配列になっているものこそが本題の StateArray です。Wait-free Stack はこの StateArray と、スタックの先頭を示す Node で構成します。

ソースコード 13: Wait-free Stack 実装の例：コンストラクタ

```
1  static Node* new_node(Item* const i, const size_t w, Node* const n) {
2    Node* const construct = (Node*)malloc(sizeof(Node));
3    construct->item = i;
4    construct->winner = w;
5    construct->next = n;
6    construct->prev = NULL;
7    /* ここにメモリバリア */
8    return construct;
9  }
10
11 /* Stateのコンストラクタ */
12 static State* new_state(const uint64_t p,
13                         const int pend,
14                         const int push,
15                         Item* const i) {
16   State* const construct = (State*)malloc(sizeof(State));
17   construct->count = p;
18   construct->pending = pend;
19   construct->is_push = push;
20   construct->item = i;
21   /* ここにメモリバリア */
22   return construct;
23 }
24
25 /* Wait-free Stackのコンストラクタ */
26 void stack_init(wf_stack* const stack, int threads) {
27   int i;
28   stack->head = new_node(NULL, 0, NULL); /* 番兵ノード */
29   stack->states[0] = NULL; /* stateの0番は使わない */
30   for (i = 1; i <= threads; ++i) {
31     /* 全てのStateに初期状態を設定 */
32     stack->states[i] = new_state(0, 0, 0, NULL);
33   }
34   /* ここにメモリバリア */
35 }
```

ソースコード 13 にコンストラクタを示します。Node と State のコンストラクタは与えられたものをそのまま代入するのみです。ただし、プロセッサやコンパイラ内でのリオーダ [16] によって、コンストラクタ内での代入操作の完了より先にデータ構造が他のスレッドから可視状態になってしまう危険があるため、メモリバリアで回避しています。

`Wait-free Stack` のコンストラクタでは、番兵として Stack の底にダミーのノードを挿入しています。Stack 操作の際、このノードに触れた場合にはスタックの底として認識するようにします。そして、各スレッドの状態を State 構造体に保存しています。この State の配列こそが `StateArray` であり、この配列の 1 エントリが 1 スレッドの状態を保持している事は前述の通りです。`StateArray` のエントリの pending メンバが 1 ならば該当スレッドは実行途中であることを意味します。しかし pending メンバの 1bit 分の情報だと周回遅れの場合に区別ができなくなるためカウンタによって世代を管理しています。こうした構造体をスレッド数だけ用意する必要があります。ただし、実装の簡潔さのための工夫として 0 番目を未使用とし 1 番目以降を各スレッドに割り当てています。そのため、都合スレッド数+1 の長さの `StateArray` を用意しています。

詳細は後述しますが、こうして世代を管理することによって `starvation` を抑止することが可能となります。なお、カウンタが桁溢れして 0 に戻る問題に付いては本稿では扱いません [17]。

これらを利用する関数宣言の一覧をソースコード 14 に示します。この中では push() と pop() のみが外部から呼び出せる関数です。

ソースコード 14: Wait-free Stack 実装内部で使う関数群

```
1  /* wait-free push */
2  void push(wf_stack* const stack, const Item item);
3
4  /* wait-free pop */
5  int pop(wf_stack* const stack, Item* const result);
6
7  /* ここから下は全てstaticな関数で、外から触れない */
8
9  /* 自分より先に操作を始めたスレッドを支援する */
10 static void help(wf_stack* const stack, const uint64_t count);
11
12 /* スタックの先頭ノードに印をつける操作 */
13 static void help_push(wf_stack* const stack, const size_t tid, const uint64_t
       count);
14 static void help_pop(wf_stack* const stack, const int tid, const uint64_t count);
15
16 /* 全スレッドの中での最大のカウンタを検出 */
17 static uint64_t max_count(const wf_stack* const stack);
18
19 /* 指定したスレッドが既に終わっている確かめる */
20 static int is_still_pending(const wf_stack* const stack, const int tid, const
       uint64_t count);
21
22 /* 先端のノードに既に印がついている場合そちらの操作を手伝う */
23 static void help_finish(wf_stack* const stack, const uint64_t count);
```

`Wait-free Stack` 全体の流れを図 22 に示します。

help() を全てのスレッドが共有している所がポイントです。その詳細は追々説明していきます。

まず push() を追ってみましょう。ソースコード 15 に push() の実装を示します。

ソースコード 15: push()

[16] リオーダとメモリバリアに付いては本稿では深追いしませんが、興味のある方は http://www.1024cores.net/home/lock-free-algorithms/so-what-is-a-memory-model-and-how-to-cook-it や文献 [9] を参照してください。

[17] 解決策としては有界タイムスタンプ [10][11][12] が提案されています

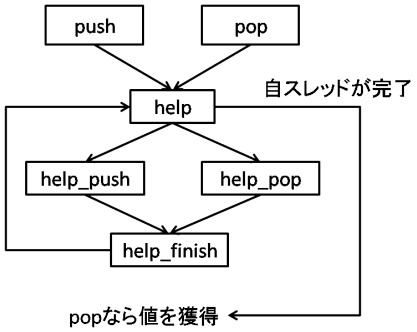

図 22: Wait-free Stack の概観

```
1  void push(wf_stack* const stack, const Item item) {
2    State* const old_state = stack->states[TID];
3
4    /* 挿入したいデータを複製 */
5    Item* const push_item = (Item*)malloc(sizeof(Item));
6    *push_item = item;
7
8    /* 全スレッドをスキャンして一番大きいカウンタを設定する。
9       これにより追い越しが発生しないためstarvationが発生しない */
10   const uint64_t my_count = max_count(stack) + 1;
11
12   /* 挿入したいデータとカウンタを使用して新しいpush操作を登録 */
13   State* push_state = new_state(my_count,
14                                 1 /*pending*/,
15                                 1 /*push*/,
16                                 push_item);
17   stack->states[TID] = push_state;
18
19   /* 登録されている操作を自分の分も含め完了させる */
20   help(stack, my_count);
21
22   /* 古いstate構造体を消去 */
23   free(old_state);
24 }
```

さて、この push() の中で実装を示していない関数が複数出てきました。これらの関数の内部を説明した後にソースコード 15 を解説します。

まず 11 行目の max_count() の実装をソースコード 16 で説明します。

ソースコード 16: 最大の count を探す手続き

```
1  static uint64_t max_count(const wf_stack* const stack) {
2    uint64_t result = 0;
3    int i;
4    /* 1からスタートしているのは 0番目がニュートラル状態を表すため */
5    for (i = 1; i <= stack->thread_max; ++i) {
6      result = result > stack->states[i]->count ? result : stack->states[i]->count;
7    }
8    return result;
9  }
```

max_count() は StateArray を走査して、最大のカウントを取得する関数です。これを用いて、自身のカウントを取得した最大カウント+1 に設定するのがソースコード 15 の 10 行目です。

これで「後から登録された作業はより大きな phase 番号が設定されている」ことになります。phase 番号が昇順になるように処理することで、登録された順とほぼ同じ順で処理することができます。このようなアルゴリズムを Lamport のパン屋のアルゴリズム [13] と呼びます。

さて、ここまででマルチスレッド脳が鍛えられた読者の方なら、「phase 番号が同じになる可能性」があることにお気づきでしょう。Lamport のパン屋のアルゴリズムでは、何らかの方法 (例えば StateArray の添字順) で優先順位を定義することで、対応順を一意に定めることができると示されています。一方、この StateArray では同一の phase 番号を持った複数のスレッドが現れても問題なく動くことが KP-Queue の論文 [8] で示されています。

この max_count() は有限の長さの配列を舐めるだけなので Wait-free です。[18]

このようにして全スレッドの操作にカウンタを振った上で、help 関数を用いて該当スレッドの操作を手伝います。その help() の実装をソースコード 17 に示します。

ソースコード 17: help()

```
1  static int scan(const wf_stack* const stack, uint64_t count) {
2    /* 操作が終わっていないスレッドを検出しcountが最小のスレッドの tidを返す */
3    /* 見つからなければ 0を返す。*/
4    int i;
5    uint64_t min_count = 0LLU;
6    int min_count_tid = 0;
7    for (i = 1; i <= stack->thread_max; ++i) {
8      if (stack->states[i]->count < min_count &&
9          is_still_pending(stack, i, count)) {
10       min_count = stack->states[i]->count;
11       min_count_tid = i;
12     }
13   }
14   return min_count_tid;
15 }
16
17 /* 自分の操作以前に登録されている操作を代行する */
18 static void help(wf_stack* const stack, const uint64_t count) {
19   size_t other_tid;
20   while ((other_tid = scan(stack, count))) {
21     /* 該当スレッドの目的がpushなら pushを、popなら popを代行する */
22     if (stack->states[other_tid]->is_push) {
23       help_push(stack, other_tid, stack->states[other_tid]->count);
24     } else {
25       help_pop(stack, other_tid, stack->states[other_tid]->count);
26     }
27   }
28 }
```

scan() はカウンタが最小でなおかつ pending 状態であるものを探して返却します。help() は scan() を用いて全てのスレッドの状態を確認し、完了していなければその進行を支援します。

これは前述した Lamport のパン屋のアルゴリズムの一部です。カウンタの古いスレッドから順に操作することで Starvation の発生を防いでいます。

scan() では、スレッドが pending 状態であるかを is_still_pending()(ソースコード 18 参照) を用いて確認します。

また、スレッドの ID から stack->states[ID] を見る事でそのスレッドの目的と状態を参照する事ができるため、操作を手伝う事 (ソースコード 17 の 23 行目、25 行目) ができます。この help() が操作全体の核を成している事は図 22 からも読み取れます。

help() は処理順を決定する重要な関数です。本稿の Wait-free アルゴリズムにおいて、”先に登録されている作業を追い抜かない”ことを担保しています。以降の help_push(),help_pop(),help_finish() は、その安全性を担保しているに過ぎません。

ソースコード 18: 実行中かどうかを判定する操作

[18]実際はこのカウントの仕方の他にアトミックな加算命令である Fetch And Add も選択肢として挙がっていますが必須ではないためこちらの数え方を用います。

```
1  static int is_still_pending(const wf_stack* const stack,
2                              const int tid,
3                              const uint64_t count) {
4    /* pending が再び真になる場合に配慮し */
5    /* pending の値とカウンタとで見比べる */
6    /* カウンタが自分より大きい場合はpending であると見做さない */
7    return stack->states[tid]->pending &&
8      stack->states[tid]->count <= count;
9  }
```

StateArray パターンでは i 番目のスレッドの作業終了時に, 対応する states[i]->pending のフラグを倒します。それにより複数のスレッドが同一の操作を多重に実行することを回避しています。

仮に処理が一回りして、再び pending が真になった場合は、count には必ず自身のカウンタよりも大きな値が設定されるため、一回りしてきたスレッドを先発のスレッドが助ける事はありません[19]。これらを元に、ソースコード 15 を見なおしてみましょう。

1. push() で push したいノードを作成 (5〜6 行目)
2. 新しい State 構造体に、これから実行したい旨 (14 行目)、操作内容が push である旨 (15 行目)、push したいもの (16 行目) を書き込む
3. 他のスレッドから観測できるように、StateArray の中の自スレッドの位置にその State 構造体を繋ぐ (17 行目)
4. 自分のカウンタ (my_count) 以下の全ての他のスレッドを手伝う (20 行目) 事で push() が完了

この部分の操作は他のスレッドによって邪魔されないため Wait-free です。

次に、自分を含む他のスレッドを手伝う help_push() をソースコード 19 に示します。

ソースコード 19: head ノードに push スレッドの印をつける

```
1  static void help_push(wf_stack* const stack,
2                        const size_t tid,
3                        const uint64_t count) {
4    /* tid 番目の State に保持されている item で、ローカルにノードを構成する */
5    /* winner に tid を指定したノードを作成する */
6    Node* new_head = new_node(stack->states[tid]->item, tid, NULL);
7
8    while (is_still_pending(stack, tid, count)) {
9      /* 差し替えるべきhead の情報を獲得 */
10     Node* const old_head = stack->head;
11     const int old_winner = old_head->winner;
12
13     if (old_head != stack->head) {
14       /* 読み出し時にずれたならやり直し */
15       continue;
16     }
17
18     /* old_winner は、進行権を獲得したスレッドの ID を保持している */
19     if (old_winner == 0) {
20       /* ニュートラル状態なら自分の操作を進行させる */
21       new_head->next = old_head;
22
23       /* ループの先頭直後でプリエンプションされていた場合のために */
24       /* ここでもpending のチェックが必要 */
25       if (is_still_pending(stack, tid, count)) {
26         /* winner が自スレッドを指した new_node で head を差し替える */
27         if (CAS(&stack->head, old_head, new_head)) {
```

[19]厳密には、自身と同じ count を獲得したスレッドが現れる可能性がありますが、そのようなスレッドの数は最大でも同時に走っているスレッド数と同じになり、有限です。

```
28            /* この瞬間に他のスレッドからnew_nodeが観測可能になる */
29            help_finish(stack, count);
30            return;
31          }
32        }
33
34      } else {
35        /* winnerが指していた他のスレッドを手伝う */
36        help_finish(stack, count);
37      }
38    }
39    /* 他のスレッドが目的操作を完了していた場合、ここに至る */
40    free(new_head);
41  }
```

help_push() は複数のスレッドから同時に呼ばれる可能性がある関数です。このため、同一の tid で呼ばれる可能性を考慮しながら、スレッド同士の動作が衝突しないよう調停する必要があります。その点で、単に別の目的を持った他のスレッドとの衝突にだけ気をつければ良い `Lock-free stack` よりも難度が上がっています。

衝突を避ける機構は三段構えになっています。まず、他のスレッドが進行権を獲得しているかどうかを確認します (19 行目)。

スタックの先頭のノードの winner がいずれかのスレッド ID を保持している際に、その ID のスレッドは操作を進行させる権利を持つため便宜上進行権と呼びます。この winner が 0 以外を指している状態の事を以後マーキングと呼びます。

誰も進行権を持っていなければ (つまり winner が 0 なら)、is_still_pending() でまだ進行していないことを再度確認した後 (25 行目)、stack → head ポインタを CAS で書き変える事で進行権を得ます (27 行目)。他のスレッドが進行権を持っていた場合は手伝いに行きます (36 行目)。

ここで気をつける必要があるのが 25 行目の is_still_pending() の再チェックです。この is_still_pending() は一見冗長ですが、もしこのチェックがなければ、

1. push(A) 操作を行うスレッド 1 が 8 行目の is_still_pending() チェックを通過
2. スレッド 1 が 8 行目に入る時点でプリエンプション
3. その間にスレッド 2 が push(A) 操作を代わりに全て完遂
4. スレッド 1 が起きて 10 行目の head 獲得から再開。この時点でスレッド 2 が既に操作を完了させた事に気づいていない
5. スレッド 1 は当初の目的通り push(A) 操作を誰にも邪魔されず完遂

となって、push(A) 操作を 2 回実行してしまいます。

それを回避するためには、old_head を獲得した後で必ず再度 is_still_pending() を行う必要があります。そのため、再チェックを行う場所は 11 行目や 16 行目でも構わないのですが、その場合はそもそも CAS を行わない場合に無駄になってしまうため CAS の直前にチェックしています。

help_push() の様子を図 23 と図 24 に示します。なお、push() では prev メンバは使用しないため図からは省略します。

このように StateArray を用いた `Wait-free` アルゴリズムは緻密なステートマシンになっており、同じ操作が複数のスレッドによって多重に行われることがないよう注意深く設計されています。複数のスレッドが同時に単一のスレッドの操作を代行しそうになった場合は図 25 のような状態になります。この場合であっても、各スレッドは Node 構造体をそれぞれ個別に持っており、`Lock-free`

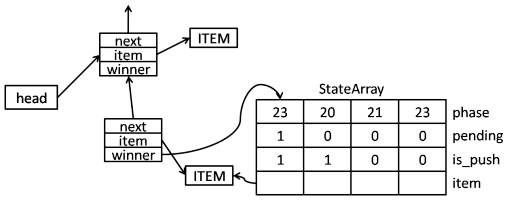

図 23: help_push 前

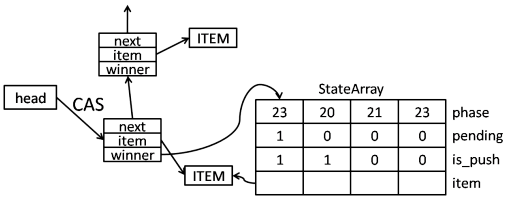

図 24: help_push 後

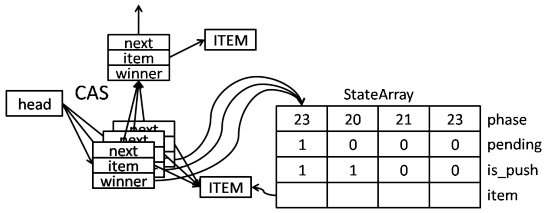

図 25: 3 つのスレッドが同時に tid=1 の push 操作を help した場合

Stack の時と全く同様に調停が行われます。そのため最大でも 1 つのスレッドしかマーキングに成功せず、安全です。

この図 24 の状態から次の状態に移る操作は図 22 に書かれているように pop() と共通のため、pop() の説明の際に併せて説明します。

さて、pop() をソースコード 20 に示します。

ソースコード 20: pop 操作

```
1  int pop(wf_stack* const stack, Item* const result) {
2    /* 消去するために古い state を保持 */
3    State* old_state = stack->states[TID];
4
5    /* pop するという旨を StateArray の自分の位置に書き込む */
6    const uint64_t my_count = max_count(stack) + 1;
7    State* pop_state = new_state(my_count, 1, 0, NULL);
8    stack->states[TID] = pop_state;
9
10   /* 自分含めた全スレッドを支援 */
11   help(stack, my_count);
12
13   /* pop に成功していれば State の item に結果が入っているので取得 */
14   Item* const got_item = stack->states[TID]->item;
15   if (got_item != NULL) {
16     /* pop 成功 */
17     *result = *got_item;
18
19     /* 使わなくなったデータを捨てる */
20     free(old_state);
21     free(got_item);
22     return 1;
23   } else {
24     /* stack が空なので pop 失敗。State だけは古い方を削除 */
25     free(old_state);
26     return 0;
27   }
28 }
```

pop() は push() と、Node の追加/取得という作業の違いに起因する部分を除いて良く似ています。help() を抜けた段階で、stack → states[TID] → item には pop した item がセットされているはず

なので、コピーして返します (11-14 行目)。

pop() は特に複雑ではありませんが、help_pop() は help_push() より若干複雑になります。ソースコード 21 に help_pop() の実装を示します。

ソースコード 21: head ノードに pop スレッドの印をつける

```
1  void help_pop(wf_stack* const stack,
2                const int tid,
3                const uint64_t count) {
4    /* head をすげ替える head を作る */
5    /* winner として自スレッドを設定している */
6    Node* new_head = new_node(NULL, tid, NULL);
7
8    while (is_still_pending(stack, tid, count)) {
9      Node* const old_head = stack->head;
10     Node* const next = old_head->next;
11     const int old_winner = old_head->winner;
12
13     if (old_head != stack->head) {
14       /* 読んでる間にズレたので読み直し */
15       continue;
16     }
17
18     /* ニュートラル状態なら自分の操作を進行させる */
19     if (old_winner == 0) {
20       /* ループの先頭直後でプリエンプションされていた場合のために */
21       /* ここでも pending のチェックが必要 */
22       if (is_still_pending(stack, tid, count)) {
23
24         /* 新しくスタックの先頭にするノードを構成 */
25         new_head->prev = old_head; /* この prev は後で使う */
26         if (next) {
27           /* 差し替え対象がスタックの底でないならその情報を獲得する */
28           new_head->item = next->item;
29           new_head->next = next->next;
30         }
31
32         if (CAS(&stack->head, old_head, new_head)) {
33           /* この瞬間に他のスレッドから new_node が観測可能になる */
34           help_finish(stack);
35
36           /* pop した Node と自分が複製した Node を解放する */
37           if (next) {
38             free(next);
39           }
40           free(old_head);
41           return;
42         }
43       }
44     } else {
45       /* winner が指していた他のスレッドを手伝う */
46       help_finish(stack);
47     }
48   }
49   /* 他のスレッドが目的操作を完了していた場合、ここに至る */
50   free(new_head);
51 }
```

help_pop() 操作も help_push() 操作と同様に head を差し替えます。しかし、Lock-free の場合とは異なり、新しくノードを作りそこに差し替える事で調停を行います。help_pop() は help_push() と同様に、ローカルに新しくノードを作って head の差し変えを行います。(Lock-free Stack のように) 既存のノードに差し変えようとすると、winner の更新と head の更新をアトミックに実行できないからです。

この処理を少し注意深く見ていきましょう。図 26 は help_pop() 直前の状態です。ここから、ノー

ドを新しく作って、必要な情報を書き込んだ状態が図 27(30 行目まで) です。もちろんこの時点ではこのノードは他のスレッドからは観測できません。

CAS で head を new_head に更新した状態が図 28(32 行目) です。CAS が成功した瞬間に 3 つの事象「head がこのノードであること」「winner は tid=1 のスレッド」「prev が指しているノードが直前のノード」が同時に外部から観測可能になります。

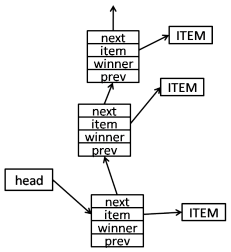

図 26: 初期状態

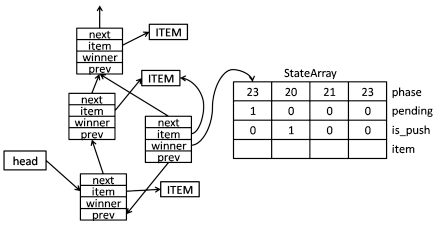

図 27: help_pop 前

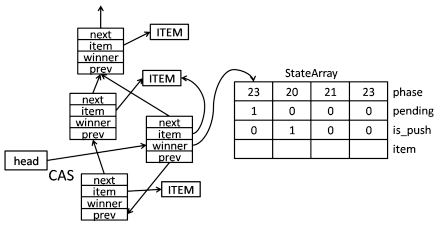

図 28: help_pop 後

さて、help_push() や help_pop() が終わった後には、head が指す先頭ノードの winner は 0 以外の値を示しています。それは進行権を勝ち取ったスレッドの tid を表しており、自身を含む全てのスレッドは、help_finish() を用いて、該当 tid に登録されている作業の完了操作を行います。

その完了操作をソースコード 22 に表します。

ソースコード 22: push 及び pop の完了操作

```
1  void help_finish(wf_stack* const stack) {
2    Node* const head = stack->head;
3    const int winner = head->winner;
4    State* const state = stack->states[winner];
5
6    /* 他のスレッドの手によって既にwinnerが初期化されたなら自分の操作に戻る */
7    if (winner == 0) {
8      return;
9    }
10
11   /* headが変わる、ということは stateは既に winner観測時より進んでしまっている */
12   if (head != stack->head) {
13     return;
14   }
15
16   /* winnerの目的が popだった場合に限り、Stateに itemを繋げる */
17   if (!state->is_push) {
18     /* StateArrayの該当部分の itemに値を代入 */
19     state->item = head->prev->item;
20   }
21
22   /* 下の代入が上の代入を追い抜かないようここにメモリバリア */
23   /* pendingフラグを折る。これで is_still_pending()が偽を返すようになるので
24      他のスレッドがこれ以降このtidの代行を開始しなくなる */
25   state->pending = 0;
26
27   /* 下の代入が上の代入を追い抜かないようここにメモリバリア */
28   /* Stackの先頭のマーキングを消去。これで他のスレッドがこのスレッドを手伝わなくなる */
29   head->winner = 0;
30 }
```

help_finish() が担う操作は、pop した item の処理、pending フラグを折る、winner をニュートラルにすることの 3 つです。特に、次の作業を開始する鍵である、winner をニュートラルにする操作の順番には細心の注意が必要です。

例えば pending フラグを折るより先に winner のマーキングを消去する (25 行目と 29 行目を逆にする) 場合を考えてみましょう。すると、winner のマーキングを消去した直後にプリエンプションされてしまうと、他のスレッドは pending フラグが立っている該当スレッドの push や pop を再び初めから実行してしまいます。それは同一のデータを多重に push, pop する事になり危険です。

また、pop した item の処理と pending フラグを折る操作を逆にする場合も考えてみましょう。pop() は、help() を抜けた段階で、自身の stete の item に pop した item が繋がれている事を期待しています。もし pending フラグを折る操作が先行した場合、その後 pop した item を State 構造体に繋いで結果を格納する前に、そもそもの pop() の呼び出しスレッドが is_still_pending() が偽になったことを観測して help() 関数を抜け、pop() の結果読み出しを行う (ソースコード 20 の 15 行目) 可能性があり、安全性が保証できません。

これらの条件から、help_finish() の内部で行われる操作の順序は「pop() の後始末」→「pending フラグ初期化」→「winner の初期化」でなくてはなりません。

help_finish() では変数の代入に CAS を使っていません。しかし、安全性は保証されています。順に見ていきましょう。

まず state 構造体への代入ですが、push/pop 操作の度に新しく作っているため、周回遅れのスレッドが現在の state に干渉することはできません。また、同じ周回ならば、item メンバの更新も pending フラグの更新も、一意な値への一方向の更新であるため、スレッド間で競合が起きても問題ありません。

次に head → winner への代入ですが、こちらも push/pop 操作で head が更新されるたびに新しい node が作成されているので、周回遅れのスレッドが現在の head に干渉することはできません。また、同じ周回ならば、winner をニュートラルにする (0 を代入する) という一意な値への一方向の更新であるため、こちらも競合が起きても問題ありません。

node 構造体の winner メンバも、初めに作成される時に作成したスレッドの tid を指した後は必ず 0 に書き換えられ、二度と 0 以外の値に戻る事はありません。つまり winner の操作も 0 に書き換えるという一方向で冪等な操作であるため、衝突しても矛盾無く動きます。

Wait-free 性　なぜこのアルゴリズムならば `Starvation` が発生しないかを説明します。そのためには環境とスケジューラが全て最悪に傾いた状態での最悪実行時間を見積もる事が必要です。

push(),pop(),help_finish() はロックもループもない操作で完結しているため、スケジューラ側から無限ループ化できる要素は一切無く、有限ステップで終了します。

次に help() における最悪のケースを考えてみましょう。自分以外の全てのスレッドが pending のまま停止しているケースが該当します。本稿では Lamport のパン屋のアルゴリズムを利用しているため、自身の作業を終了するためには最大で (総スレッド数) x (help_push(), help_pop() の実行ステップ数) の実行コストがかかります。よって、help_push() と help_pop() が有限ステップで終了するならば、help() は `Wait-free` になります。

それではその help_push() と help_pop() が `Wait-free` であるかどうかを検証していきましょう。help_push() と help_pop() が内部で行っている操作は CAS スピンそのものなので、その中で `Starvation` が発生するかどうかが焦点になります。

CAS スピンが無限ループとなるためには永久にそのスレッドが CAS に失敗し続ける実行パスが存在しなくてはなりません。

Lamport のパン屋のアルゴリズムを利用しているので、”phase 番号が小さい順に処理されていく”保証がここでも効いてきます。あるスレッドが CAS で負けつづけたとしても、最大で (総スレッド数 − 1) 回負けた段階で、自身が”最小の phase 番号を持つ”スレッドになります。こうなる

と、全てのスレッドがその作業を手伝うことになるので、必ずいずれかのスレッドで作業が完了します[20]。すると is_still_pending() が偽を返すようになるので、ループから抜けることになります。

つまり help_push() と help_pop() 内の CAS スピンが無限ループとなる実行パスは存在しえません。

以上から、help_push() も help_pop() も有限ステップで終了することが確認できました。

よって、このアルゴリズムは `Wait-free` です。

6 ベンチマーク比較

ここまでで `Lock-free`、`Obstruction-free`、`Wait-free` の Stack の実装を追ってきました。気になるのがそれぞれの速度差です。そこで、この 3 種類に加えて Mutex を用いて排他した実装とでベンチマーク比較をしてみました。実験環境は以下の通りです。

- Amazon EC2 の hi1.x4large インスタンス
- CPU は 2.4Ghz の Xeon E5620 16 コア (恐らく物理 4 コアの仮想 8 コアが 2 ソケット構成)
- コンパイラは GCC 4.7.3。最適化オプションは-O2。
- スレッドアフィニティを設定し、1 コア目から順番にスレッドを固定して割り付ける
- スレッド数がコア数を超えた場合にはまた 1 コア目から順に割り付ける
- 1 スレッドあたりそれぞれ 10000 回の push() 及び 10000 回の pop() を行う
- 全スレッドがその操作を終えるまでの時間を計測し、秒間 Operation 数を測定
- Stack の node の寿命管理には QSBR(コラム参照) を使用
- それぞれ 10 回測定を行い平均値を採用し標準偏差をエラーバーで表示
- 論理コア数の 2 倍である 32 スレッド数までを測定

結果を図 29 に表します。lockstack が `mutex` を用いた Stack で、wfstack が `Wait-free Stack`、lfstack が `Lock-free Stack`、ofstack が `Obstruction-free Stack` をそれぞれ表します。

1 つの head ポインタを取り合うアルゴリズムのため、マルチコアの性能を引き出して高速化できているものは lockstack 含めて一つもありません。その中では mutex を用いた `lockstack` が `Lock-free` 実装である `lfstack` に倍以上の差を付けて圧倒的に高速です。一般にスループットを目的とした場合に、`Non-blocking` アルゴリズムが mutex に勝利するケースは稀です[21]。それでも `Non-blocking` アルゴリズムを利用する理由とは何でしょうか。それは mutex の弱点であるレイテンシの不安定さを補う為にあります。今回のベンチマークでは `lockstack` は 5 コアと 16 コア地点で計測結果に大きなバラつきが出てしまっています。このブレが全体の品質に影響を与えてしまうのが 4.1.3 で説明したように、音響システムやリアルタイムシステム、ネットワーク機器などのハードなパフォーマンスを要求するシーンです。`Non-blocking` な実装では大幅にバラつきが減っており、安定したパフォーマンスが出ている事が分かります。

lockstack が圧倒的に速いため、それを除いた結果を図 30 に表します。lockstack を除くと一番高速なのは `Lock-free` な `lfstack`、次点で `Obstruction-free` な `ofstack`、一番遅いのは `Wait-free` な `wfstack` でした。しかし一番遅い `wfstack` の性能のブレなさは目を見張るものがあります。

[20] これは CAS スピンが持つ Lock-free 特性によるもの

[21] TAoMP 本 [5] でも、パフォーマンスが良いと明記されているのは Queue しかない。

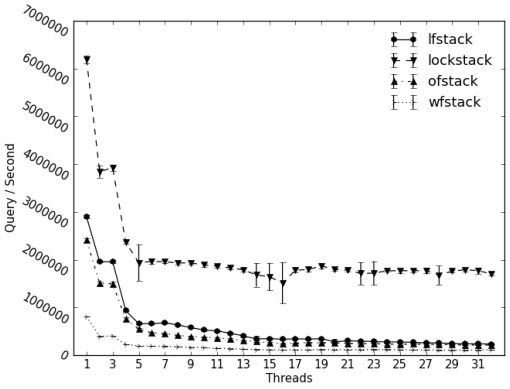

図 29: ベンチマーク結果

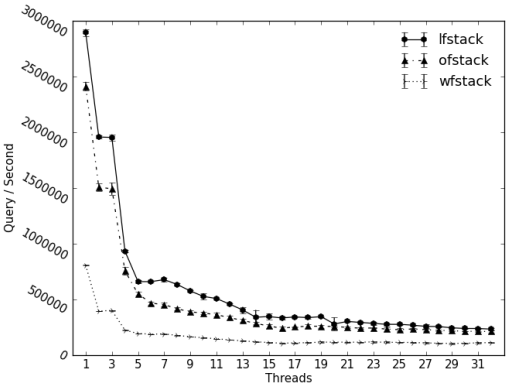

図 30: lockstack を除外したもの

高速化　push() と pop() が衝突し続けるタイプの問題は、一般に Producer-Consumer 問題と呼ばれています。これを高速化するための手法として `Elimination`[14] という技法が提案されています。それは push と pop の対称性に着目し、リトライ待ち状態に陥った push と pop がランダムに待ち合わせを行うことで head ポインタを介さずに値を交換する手法です。

`Elimination` ではダイナミックにサイズが変動する配列のランダムな位置で待ち合わせを行う事で確率的に性能を上げますが、配列の代わりに線形リストを円環状に繋げたリングバッファを待ち合わせに使う `Rendesvous`[15]、調停専用のスレッドが push と pop の割り付けを行う `Flat-combining`[16] なども提案されています。これらのアルゴリズムの詳細についてはまた機会があれば紹介します。

7　まとめ

本稿では `Obstruction-free`、`Lock-free`、`Wait-free` のそれぞれの特性を持った Stack 構造の実装を題材にその違いを追い、ベンチマークによる比較を行いました。分散・並行・並列の研究は違いに影響し合いながら発展しており、並行動作とその特性を深く追求することは分散・並列の研究者にとっても重要な土台となります。

並行プログラムについて `Lock-free` や `Linealizability` などの定性的な性質を探る事は、小手先の理解や把握に留まらず、デバッグやアーキテクチャ全体の改良にも大きく貢献すると確信しています。

この魔導書で他の著者の方々が紹介しているアルゴリズムについても、その特性を考えることはその性質のより深い理解へと導いてくれる事でしょう。

コラム：局所的な GC

ガベージコレクタとスケーラビリティの関係については多くの研究が行われてきました。基本的なマーク&スイープではマーク中にミューテータ (プログラム本体) がメモリ空間を書き換えないよう全体を停止させる事が基本なためスケーラビリティを大きく阻害します。並行処理で参照カウンタを利用する場合はカウンタのアップ・ダウンという至極普通の操作に対して、負荷の高いアトミック命令を用いる事になりオーバーヘッドの大きさが問題となります。これらの問題はオペレーティングシステムのカーネルの実装では古くから問題になっていました。プロセスの管理などの為に Read-Write ロックを用いた場合では、スケーラビリティに限界がありました。特にカーネル内部では Read がほとんどの割合を占めるため、Write の性能を犠牲にしてでも Read の性能を引き上げる事に注力されました。それが RCU[9] という技術です。Linux カーネルでは適用が進んでおり、2013 年現在 7000 箇所以上で使われています。カーネル内でそれほど適用実績があるなら、ユーザースペースでも使いたいというのは自然ですが、RCU は制約が多く直接ユーザースペースから利用することは現実的ではありません。そこでユーザースペースでも RCU と同等の効果を得られる UserspaceRCU[17] という技術が研究されてきました [18]。具体的には Quiescent State Based Reclamation(QSBR) や Epoch Based Reclaimation(EBR)[19]、また RCU という文脈ではないですが Hazard Pointer[20] や Pass the Buck[21] などが提案されています。
これらのアルゴリズムは一長一短で、QSBR は参照のオーバーヘッドが少ない一方で更新時のブロッキング期間は長くなる傾向があります。逆に Hazerd Pointer は参照のたびにメモリバリアを使うためオーバーヘッドが大きくなりがちでありながら、更新時のオーバーヘッドを抑えられる利点があります。これらのアルゴリズムの詳細はまたの機会に紹介します。

参考文献

[1] Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, spring joint computer conference*, AFIPS '67 (Spring), pp. 483–485, New York, NY, USA, 1967. ACM.

[2] Maurice Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, Vol. 13, pp. 124–149, 1991.

[3] James Aspnes, Maurice Herlihy, and Nir Shavit. Counting networks. *J. ACM*, Vol. 41, No. 5, pp. 1020–1048, September 1994.

[4] Nir Shavit and Asaph Zemach. Diffracting trees. *ACM Trans. Comput. Syst.*, Vol. 14, No. 4, pp. 385–428, November 1996.

[5] Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2008.

[6] Bernadette Charron-Bost; Fernando Pedone; Andre Schiper. *Replication: Theory and Practice (Lecture Notes in Computer Science / Theoretical Computer Science and General Issues).* Springer, 1st edition. edition, April 2010.

[7] Alex Kogan and Erez Petrank. A methodology for creating fast wait-free data structures. In *Proceedings of the 17th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*, PPoPP '12, pp. 141–150, New York, NY, USA, 2012. ACM.

[8] Alex Kogan and Erez Petrank. Wait-free queues with multiple enqueuers and dequeuers. In *Principles and Practice of Parallel Programming*, pp. 223–234, 2011.

[9] Paul E. McKenney. *Is Parallel Programming Hard, And, If So, What Can You Do About It?* kernel.org, Corvallis, OR, USA, 2011.

[10] Danny Dolev and Nir Shavit. Bounded concurrent time-stamping. *SIAM J. Comput.*, Vol. 26, No. 2, pp. 418–455, April 1997.

[11] Sibsankar Haldar and Paul Vitányi. Bounded concurrent timestamp systems using vector clocks. *J. ACM*, Vol. 49, No. 1, pp. 101–126, January 2002.

[12] Cynthia Dwork and Orli Waarts. Simple and efficient bounded concurrent timestamping and the traceable use abstraction. *J. ACM*, Vol. 46, No. 5, pp. 633–666, September 1999.

[13] Leslie Lamport. A new solution of dijkstra's concurrent programming problem. *Commun. ACM*, Vol. 17, No. 8, pp. 453–455, August 1974.

[14] Danny Hendler, Nir Shavit, and Lena Yerushalmi. A scalable lock-free stack algorithm. In *ACM Symposium on Parallel Algorithms and Architectures*, pp. 206–215, 2004.

[15] Yehuda Afek, Michael Hakimi, and Adam Morrison. Fast and scalable rendezvousing. In *Proceedings of the 25th international conference on Distributed computing*, DISC'11, pp. 16–31, Berlin, Heidelberg, 2011. Springer-Verlag.

[16] Danny Hendler, Itai Incze, Nir Shavit, and Moran Tzafrir. Flat combining and the synchronization-parallelism tradeoff. In *Proceedings of the 22nd ACM symposium on Parallelism in algorithms and architectures*, SPAA '10, pp. 355–364, New York, NY, USA, 2010. ACM.

[17] Thomas E. Hart, Paul E. McKenney, Angela Demke Brown, and Jonathan Walpole. Performance of memory reclamation for lockless synchronization. *Journal of Parallel and Distributed Computing*, Vol. 67, No. 12, pp. 1270–1285, May 2007.

[18] LTTng Project Userspace RCU. http://lttng.org/urcu.

[19] Thomas E. Hart, Paul E. Mckenney, and Angela Demke Brown. Making lockless synchronization fast: Performance implications of memory reclamation. In *In 2006 International Parallel and Distributed Processing Symposium (IPDPS 2006)*, 2006.

[20] Maged M. Michael. Hazard Pointers: Safe Memory Reclamation for Lock-Free Objects. *IEEE Transactions on Parallel and Distributed Systems*, Vol. 15, pp. 491–504, 2004.

[21] Maurice Herlihy, Victor Luchangco, and Mark Moir. *The Repeat Offender Problem: A Mechanism for Supporting Dynamic-Sized, Lock-Free Data Structures.* 2002.