

nakameguro_feature.cpp

vol. 8



2018/9/13
yoh(@yohhoy)

未承諾広告※

事前にご一読頂けると、より楽しめると思います。

- スレッドセーフという幻想と現実

<https://yohhoy.hatenablog.jp/entry/2013/12/15/204116>

- メモリモデル？なにそれ？おいしいの？

<https://yohhoy.hatenablog.jp/entry/2014/12/21/171035>

おことわり

- 各種ヘッダ#includeは省略
- `using namespace std;`
- C++マルチスレッドプログラミングの基礎知識
 - Mutex, ScopedLock, atomic変数, データ競合(data race)
- CPU/プロセッサの基礎知識
 - キャッシュライン(cache line), キャッシュコヒーレンシ(cache coherence), アウトオブオーダー(OoO)実行

もくじ

- `scoped_lock<...>`
- `shared_mutex`
- Interference sizes
- `atomic::is_always_lock_free`
- Deprecate `shared_ptr::unique`
- Temporarily discourage `memory_order_consume`

もくじ

- `scoped_lock<...>`
- `shared_mutex`
- Interference sizes
- `atomic::is_always_lock_free`
- Deprecate `shared_ptr::unique`
- Temporarily discourage `memory_order_consume`

お役立ち情報



ヲタク向け

scoped_lock<...>

複数ミューテックス対応 ScopedLock

- lock_guard<Mutex>の上位互換
- ミューテックス型の自動推論
- デッドロック回避 (cf. lockフリー関数)

<https://wg21.link/p0156r2> Variadic lock_guard (Rev. 5)

<https://wg21.link/p0739r0> Some improvements to class template argument deduction integration into the standard library

scoped_lock<M1>

```
// C++14
```

```
mutex m;
```

```
{
```

```
    lock_guard<decltype(m)> lk{m};
```

```
    /* 共有変数へのアクセス */
```

```
} // mを自動unlock
```

```
// C++17
```

```
mutex m;
```

```
{
```

```
    scoped_lock lk{m};
```

```
    /* 共有変数へのアクセス */
```

```
} // mを自動unlock
```

「クラステンプレートのテンプレート引数推論（C++17）」により、ミューテックス型の明示指定が不要に（明示してもよい）。

scoped_lock<M1, M2>

```
mutex m1, m2;
```

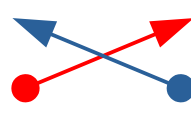
```
thread#1 {  
    lock_guard<mutex> lk1{m1};  
    lock_guard<mutex> lk2{m2};  
    /* 共有変数へのアクセス */  
} // m1,m2を自動unlock
```

```
thread#2 {  
    lock_guard<mutex> lk2{m2};  
    lock_guard<mutex> lk1{m1};  
    /* 共有変数へのアクセス */  
} // m2,m1を自動unlock
```


scoped_lock<M1, M2>

```
mutex m1, m2;
```

```
thread#1 {                                thread#2 {  
    lock_guard<mutex> lk1{m1};              lock_guard<mutex> lk2{m2};  
    lock_guard<mutex> lk2{m2};              lock_guard<mutex> lk1{m1};  
    /* 共有変数へのアクセス */              /* 共有変数へのアクセス */  
} // m1,m2を自動unlock                      } // m1,m2を自動unlock
```



デッドロック！

scoped_lock<M1, M2>

// C++14

```
mutex m1, m2;
```

```
{
```

```
    lock(m1, m2);
```

```
    lock_guard<decltype(m1)>
```

```
        lk1{m1, adopt_lock};
```

```
    lock_guard<decltype(m2)>
```

```
        lk2{m2, adopt_lock};
```

```
    /* 共有変数へのアクセス */
```

```
} // m1,m2を自動unlock
```

// C++17

```
mutex m1, m2;
```

```
{
```

```
    scoped_lock lk{m1, m2};
```

```
    /* 共有変数へのアクセス */
```

```
} // m1,m2を自動unlock
```

参考 <http://d.hatena.ne.jp/yohhoy/20120919/p1>

ScopedLockクラス比較

	lock_guard (C++11)	scoped_lock (C++17)	unique_lock (C++11)
adopt_lock	○	○	○
テンプレート 引数推論	○ (C++17以降)	○	○ (C++17以降)
遅延Lock TryLock			○
条件変数			○
複数Mutex		○	

shared_mutex

タイムアウトサポート無し共有ミューテックス

- Readers-Writerロック・セマンティクス
- C++14ライブラリ仕様の補完
(shared_timed_mutexはC++14で追加済み)
- タイムアウト対応版よりも最適な実装（かも）

<https://wg21.link/n4508> A proposal to add shared_mutex (untimed) (Revision 4)

排他/共有ロック

「書込(write)は一人だけ／読取(read)はみんな同時に」

- 排他ロック(Exclusive Lock)
 - 共有変数へ書き込めるスレッドは1つだけ
 - 他スレッドからの排他/共有ロック獲得要求をブロック
- 共有ロック(Shared Lock)
 - 共有変数から複数スレッドが同時に読み取り可能
 - 他スレッドからの共有ロック獲得要求は成功する
 - 他スレッドからの排他ロック獲得要求をブロック

排他/共有ロック

```
int resource; // 共有変数
shared_mutex sm;
```

```
write_thread#0 {
    // 排他ロックを獲得
    lock_guard wlk{sm};
    // 共有変数への書込み
    resource = 42;
} // 排他ロックを解放
```

```
read_thread#1..N {
    // 共有ロックを獲得
    shared_lock rlk{sm};
    // 共有変数からの読取り
    int local = resource;
} // 共有ロックを解放
```

排他/共有ロック

```
int resource; // 共有変数
shared_mutex sm;
```

```
write_thread#0 {
    // 排他ロックを獲得
    lock_guard wlk{sm};
    // 共有変数への書込み
    resource = 42;
} // 排他ロックを解放
```

```
read_thread#1..N {
    // 共有ロックを獲得
    shared_lock rlk{sm};
    // 排他ロックじゃないのに
    // 共有変数へ書き込んだら
    // どうなるのっと...
    resource += 1;
} // 共有ロックを解放
```

排他/共有ロック

```
int resource; // 共有変数
shared_mutex sm;
```

```
write_thread#0 {
    // 排他ロックを獲得
    lock_guard wlk{sm};
    // 共有変数への書き込み
    resource = 42;
} // 排他ロックを解放

read_thread#1..N {
    // 共有ロックを獲得
    shared_lock slk{sm};
    // 共有変数への読み込み
    // やないのに
    // 共有変数へ書き込んだら
    // どうなるのっと...
    resource += 1;
} // 共有ロックを解放
```

**データ競合！
未定義動作！**

標準Mutex一覧

	排他 ロック	共有 ロック	再帰 ロック	タイムアウト サポート	ヘッダ
<code>mutex</code>	○				<mutex>
<code>timed_mutex</code>	○			○	
<code>recursive_mutex</code>	○		○		
<code>recursive_timed_mutex</code>	○		○	○	
<code>shared_mutex</code>	○	○			<shared_mutex>
<code>shared_timed_mutex</code>	○	○		○	

C++標準を超えて...

`recursive_(timed_)mutex` = 「必要悪」

- “正しく・良い”並行設計では再帰ロックを必要としない。
 - 「何回ロックしても安全・便利だからとりあえず再帰ロック」は設計者の怠慢。並行動作を阻害。スパゲティ設計地獄の一丁目。
- 外部コードとの相互運用のため避けがたい状況もある。
 - 真に再帰ロックを必要とするならば、堂々と使えばよい。

※ 上記提言はスライド作者個人の主張に基づきます。

参考 <http://www.zaval.org/resources/library/butenhof1.html>

C++標準を超えて...

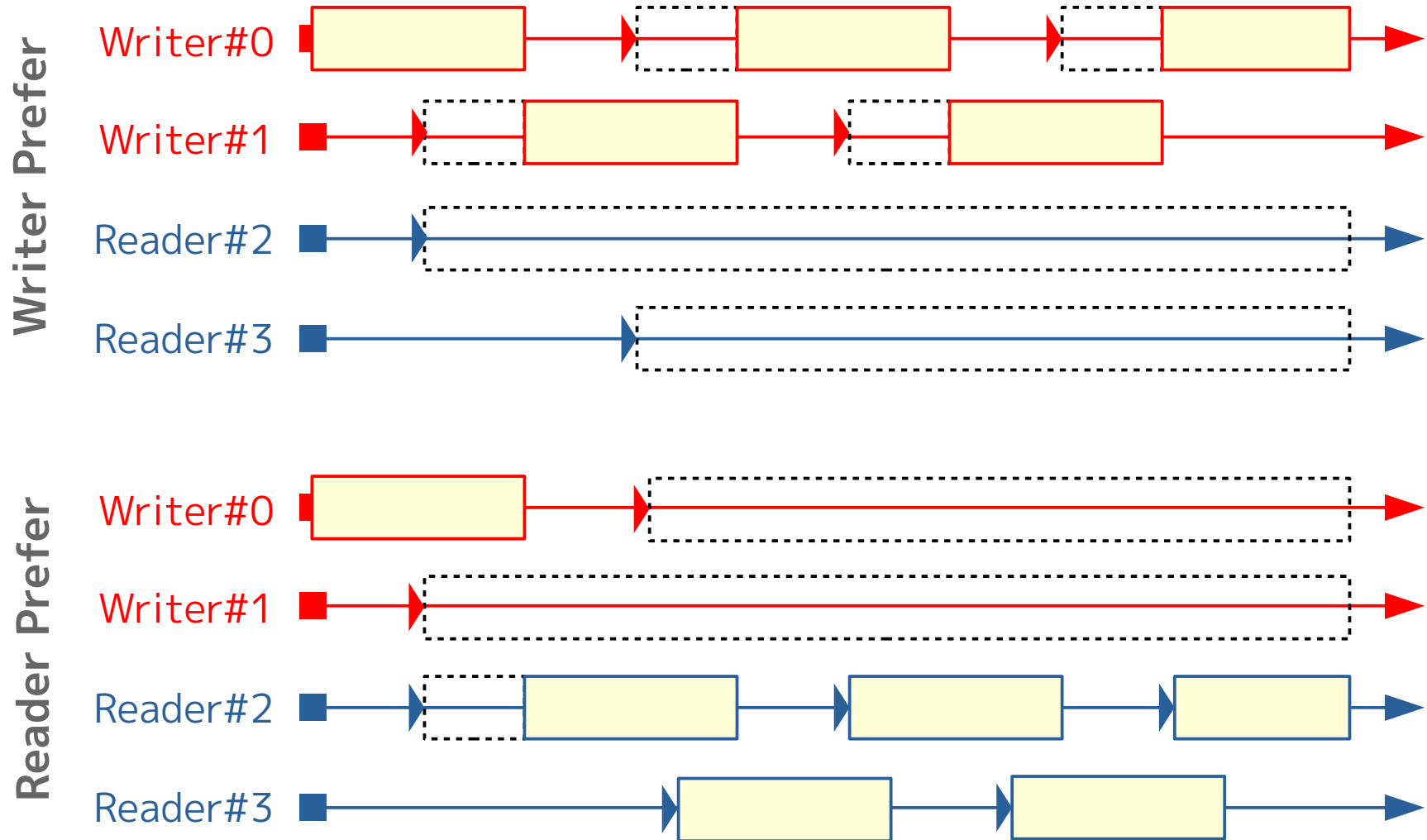
`shared_(timed_)mutex`では、排他/共有ロック競合時のスケジューリング戦略を規定しない。

- **Reader starvation** : 複数Writerスレッドが排他ロックを連続的に獲得しており、Readerスレッドがなかなか共有ロックを獲得できない。
- **Writer starvation** : 複数Readerスレッドが共有ロックを保持し続けており、Writerスレッドがなかなか排他ロックを獲得できない。

例 : Linux環境のPOSIXスレッド(Pthreads)実装では、スケジューリング戦略をオプション指定可能。 `PTHREAD_RWLOCK_PREFER_*`

※ Starvation=飢餓状態

Reader/Writer Starvation



C++標準を超えて...

「lock関数の呼出順 != ロック獲得順序」

- C++標準Mutex仕様は不公平(unfair)スケジューリング。
- 一般論として、公平(fair)スケジューリングの保証は実行時オーバーヘッドが大きくなる。
- 高いロック競合(lock contention)状態でない限り、実行効率の良い不公平スケジューリングがベター。
- どうしても公平性が必要な場合は...
 - Intel TBBの `tbb::queuing_mutex`
 - <https://github.com/yohhoy/yamc>

Interference sizes

CPUキャッシュラインサイズ・コンパイル時定数

- 標準ライブラリ中 識別子の長さNo.1！
- False-Sharing回避用の定数：
`hardware_destructive_interference_size`
- True-Sharing促進用の定数：
`hardware_constructive_interference_size`

<https://wg21.link/p0154r1> constexpr

`std::thread::hardware_{true,false}_sharing_size`



お名前長さコンテスト in C++

同率1位 [39文字]

- `atomic_compare_exchange_strong_explicit`
- **`hardware_constructive_interference_size`**

同率3位 [38文字]

- `propagate_on_container_copy_assignment`
- `propagate_on_container_move_assignment`
- **`hardware_destructive_interference_size`**

参考 <https://qiita.com/yohhoy/items/7d76e9f385876e16a93b>

定数名が難解問題

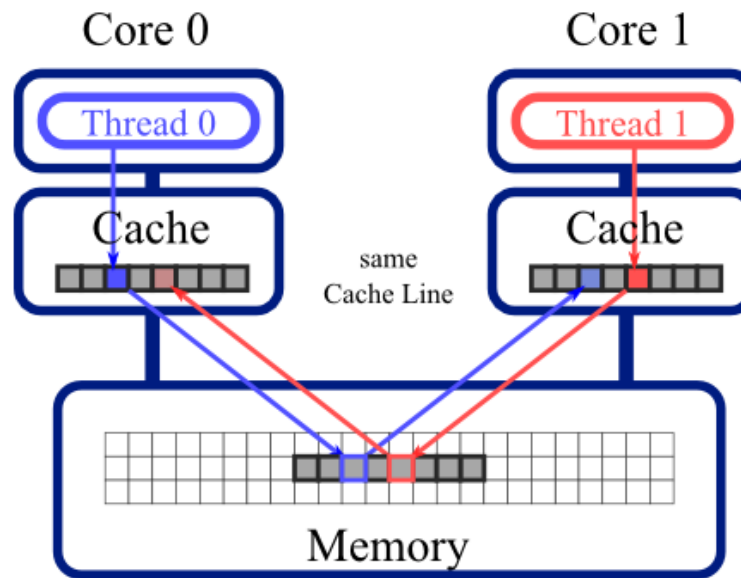
初版N4523では `hardware_{true,false}_sharing_size` と直接的な定数名だったが、抽象的な名前に変更された。

- `false_sharing` → `destructive_interference` (弱め合う干渉)
- `true_sharing` → `constructive_interference` (強め合う干渉)

こんな定数名を覚えておくほど人生は長くないので、必要になったときに大人しく検索しましょう...

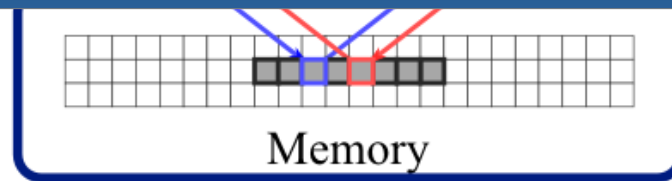
キャッシュライン競合

- メインメモリとキャッシュ内容は常に同期される。
- キャッシュ管理はキャッシュライン単位 [64Byteなど]
- 無関係な2変数が偶然に同一キャッシュラインに配置されると...



キャッシュライン競合

- メインメモリとキャッシュ内容は常に同期される。
- 書込を終えてメモリへ向かう複数の変数達。
- 疲れからか、不幸にも黒塗りの
キャッシュライン上で衝突してしまう。
- 後輩をかばいすべての責任を負った三浦に対し、
システムの主、黒の暴力団員谷岡
に言い渡された示談の条件とは . . .



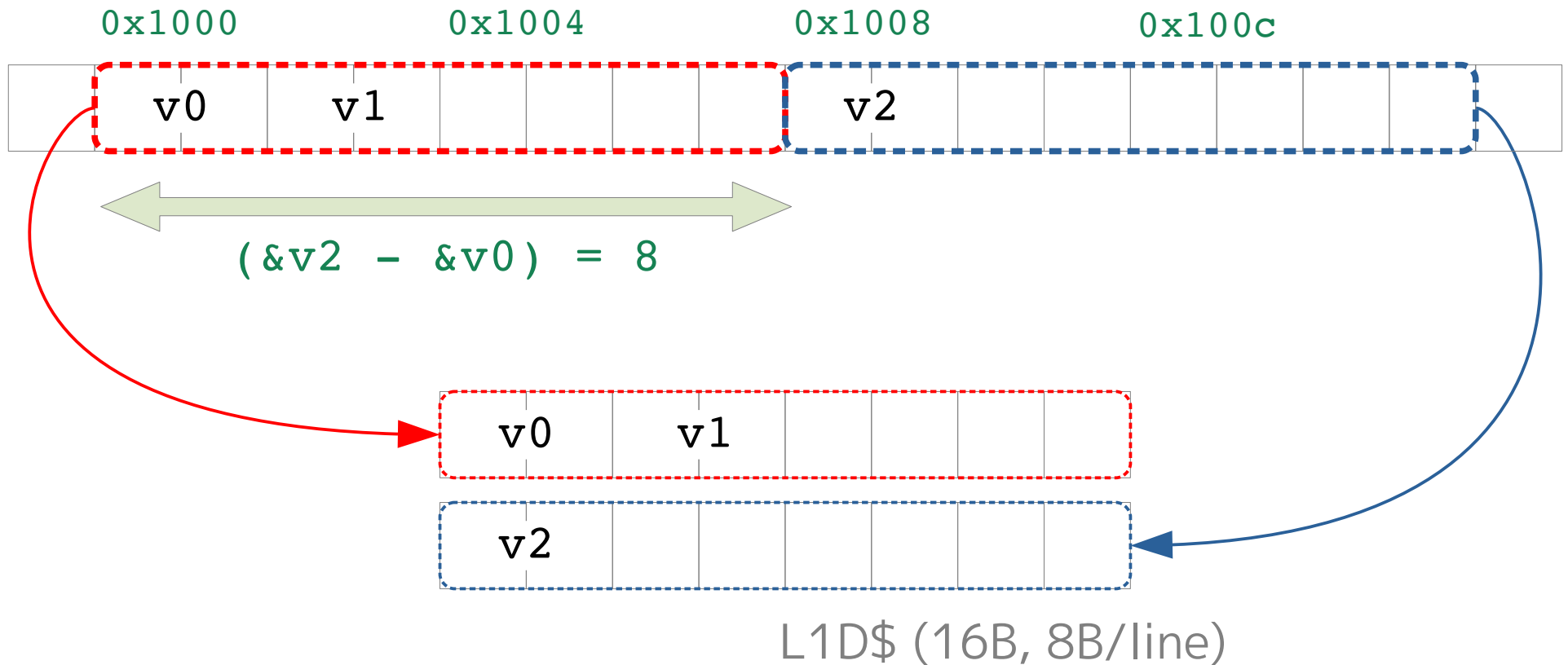
キャッシュライン競合

「偽りの共有 (False-Sharing)」

- 無関係な2変数A, Bが、偶然に同一キャッシュラインに配置されたと仮定：
 - プロセッサ#1による変数A更新の結果、該当キャッシュラインは無効化(invalidate)される。
 - プロセッサ#2が変数Bから読み取るとき、該当キャッシュラインは無効化されており、本来は不要であったメインメモリ→キャッシュ間のデータ転送が発生する。
 - 頻度によっては深刻なパフォーマンス低下を招く。

キャッシュライン競合

address=



※ メモリキャッシュ機構の模式図。L1D\$=Level1 Data Cache

なぜ2つの定数？

Question

“L1キャッシュラインサイズ”を表す定数1個じゃだめ？

Answer

2つの利用目的「キャッシュラインを分離したい」

「キャッシュラインを共有したい」で分けた。

コンパイル時には実行環境のキャッシュラインサイズが未確定なケースを考慮し、目的別に保守的な値を採用できる仕組み。たとえば WebAssembly など。

P0154作者のコメント <https://stackoverflow.com/questions/39680206/>

hardware_(ry 利用例

```
// C++17仕様[hardware.interference] よりExample抜粋
struct keep_apart {
    alignas(hardware_destructive_interference_size)
        atomic<int> cat;
    alignas(hardware_destructive_interference_size)
        atomic<int> dog;
};

struct together {
    atomic<int> dog;
    int puppy;
};

struct kennel {
    // Other data members...
    alignas(sizeof(together)) together pack;
    // Other data members...
};

static_assert(
    sizeof(together) <= hardware_constructive_interference_size);
```

(注：puppy=子犬、kennel=犬小屋)

hardware_(ry 対応状況

各C++コンパイラの hardware_(ry 対応状況：

- MSVC：VisualStudio 2017 15.3以降 対応済み
- GCC, Clang：2018年9月現在 未対応

議論はされているものの、対応アーキテクチャが幅広いぶん
難しい問題がいろいろと... (MSVCはx86と一部ARMだけ)

<http://clang-developers.42468.n3.nabble.com/RFC-C-17-hardware-constructive-destructive-interference-size-td4060786.html>

`atomic::is_always_lock_free`

atomic変数のlock-free性コンパイル時確認

- C++テンプレート・フレンドリー
- (プリプロセッサマクロ`ATOMIC_*_LOCK_FREE`, `atomic::is_lock_free`メンバ関数は従来通り)

<https://wg21.link/p0512r1> `constexpr atomic<T>::is_always_lock_free`

atomic変数 != lock-free

atomic<T>型に対する操作はlock-freeとは限らない。

- atomic変数がロックベース、つまり通常変数+Mutexで内部実装されていても良い。
- ロックベース実装の場合、シグナルハンドラ中からのatomic変数アクセスはデッドロックを引き起こす。
- lock-free保証はatomic_flag型のみ。ただし提供操作は最低限で(2つ)、atomic<bool>よりも使いづらい。
 - アトミックな “Test-and-Set(TAS)命令” or “メモリオペランドをとるSwap命令” があれば実装可能

lock-free性の確認

プリプロセス時の確認

- `ATOMIC_*_LOCK_FREE`マクロ定数（整数値）
- 型レベルで `Never(0)` / `Sometimes(1)` / `Always(2)` lock-free

コンパイル時の確認

- `is_always_lock_free`コンパイル時定数（bool値）
- `atomic<T>`型レベルでのlock-free性を確認

実行時の確認

- `is_lock_free`メンバ関数（bool値）
- オブジェクト(atomic変数)単位でのlock-free性を確認

Sometimes lock-free?

Question

“ときどき” lock-freeとは。

Answer

動的リンクライブラリのバージョンアップなどで、将来的にはlock-freeに振る舞う可能性を考慮する。

やむを得ない“不適切なアライメントをもつアトミック変数”の存在を許容する。型レベルでのlock-free保証を諦めることで、適正アライメントなアトミック変数のみを対象とした効率的実装を選択できる。

“lock-free” is 何

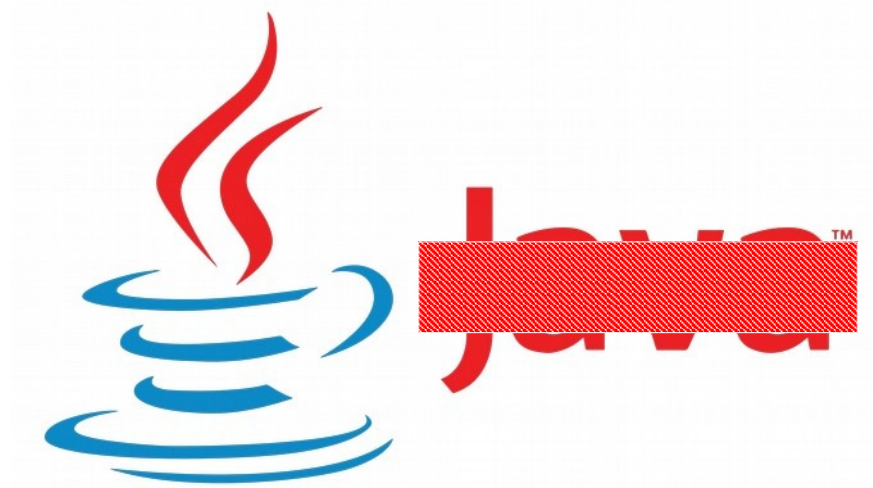
C++17仕様 [intro.progress]/p2 より超訳：

1. ブロックされていないスレッドが1つだけならば、該当スレッドによる操作は必ず完了する。
2. 複数スレッドが並行実行中ならば、少なくとも誰かの操作は必ず完了する。

C++仕様定義は、通常のNon-Blocking区分とは少し異なる。CAS命令ではなくLL/SC命令を提供するCPUを考慮し、一般的なlock-freeよりも弱い要件を採用している。

※ CAS = Compare-And-Swap, LL/SC = Load-Link/Store-Conditional

COFFEE BREAK



Here Be Dragons (ドラゴンの住まう地)



(恐怖のランダムメモリ破損
破滅のドラゴン)

“The C++17 Lands”, 部分

<http://fearlesscoder.blogspot.com/2017/02/the-c17-lands.html>

Deprecate `shared_ptr::unique`

`shared_ptr::unique` メンバ関数の非推奨化

- 誤使用によるデータ競合・UBリスクの回避
- 並行アクセス操作時は `shared_ptr::use_count` メンバ関数戻り値を近似値とみなす旨を明記
- (`unique` は C++20 で削除予定)

<https://wg21.link/p0521r0> Proposed Resolution for CA 14 (`shared_ptr::use_count/unique`)

<https://wg21.link/n4619> Editors' Report -- Working Draft, Standard for Programming Language C++, LWG motion 26

shared_ptr::use_count

```
int resource = 0; // 共有変数
shared_ptr<T> sp;
```

```
thread#0 {
```

```
    // 参照カウント +1
```

```
    auto p0 = sp;
```

```
    // 共有変数への書込み
```

```
    resource = 42;
```

```
    // 参照カウント -1
```

```
    p0 = nullptr;
```

```
}
```

```
thread#1 {
```

```
    // 参照カウントの値に基づいて...
```

```
    if (sp.unique()) // or
```

```
    if (sp.use_count() == 1)
```

```
{
```

```
    // 共有変数からの読取り
```

```
    int local = resource;
```

```
}
```

```
}
```


shared_ptr::use_count

```
int resource = 0; // 共有変数
shared_ptr<T> sp;
```

```
thread#0 {
```

```
// 参照カウント +1
```

```
auto p0 = sp;
```

```
// 共有変数への書き込み
```

```
resource = 42;
```

```
// 参照カウント -1
```

```
p0 = nullptr;
```

```
}
```

```
thread#1 {
```

```
// 参照カウントの値に基づいて...
```

```
if (sp.unique()) // or
```

```
p.use_count() == 1)
```

```
// 共有変数からの読取り
```

```
int local = resource;
```

```
}
```

```
}
```

スレッド間の
“同期”効果なし

shared_ptr::use_count

```
int resource = 0; // 共有変数
shared_ptr<T> sp;
```

```
thread#0 {
```

```
// 参照カウント +1
```

```
auto p0 = sp;
```

```
// 共有変数への書き込み
```

```
resource = 42;
```

```
// 参照カウント -1
```

```
p0 = nullptr;
```

```
}
```

```
thread#1 {
```

```
// 参照カウントに基づいて...
```

```
// or
```

```
sp.use_count() == 1)
```

```
// 共有変数からの読取り
```

```
int local = resource;
```

```
}
```

```
}
```

**データ競合！
未定義動作！**

同期が未よし

参考 <http://d.hatena.ne.jp/yohhoy/20161217/p1>

shared_ptr::use_count

```
int resource = 0; // 共有変数
shared_ptr<T> sp;
```

```
thread#0 {
```

```
    // 参照カウント +1
```

```
    auto p0 = sp;
```

```
    // 共有変数への書込み
```

```
    resource = 42;
```

```
    // 参照カウント -1
```

```
    p0 = nullptr;
```

```
}
```

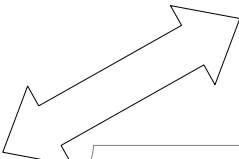
```
thread#1 {
```

```
    // 参照カウントの値に基づいて...
```

```
    if (sp.unique()) // or
```

```
    if (sp.use_count() == 1)
```

```
{
```



**happens-before関係
が無いことを明確化**

<https://timsong-cpp.github.io/cppwp/n4659/util.smartptr.shared.obs#13>

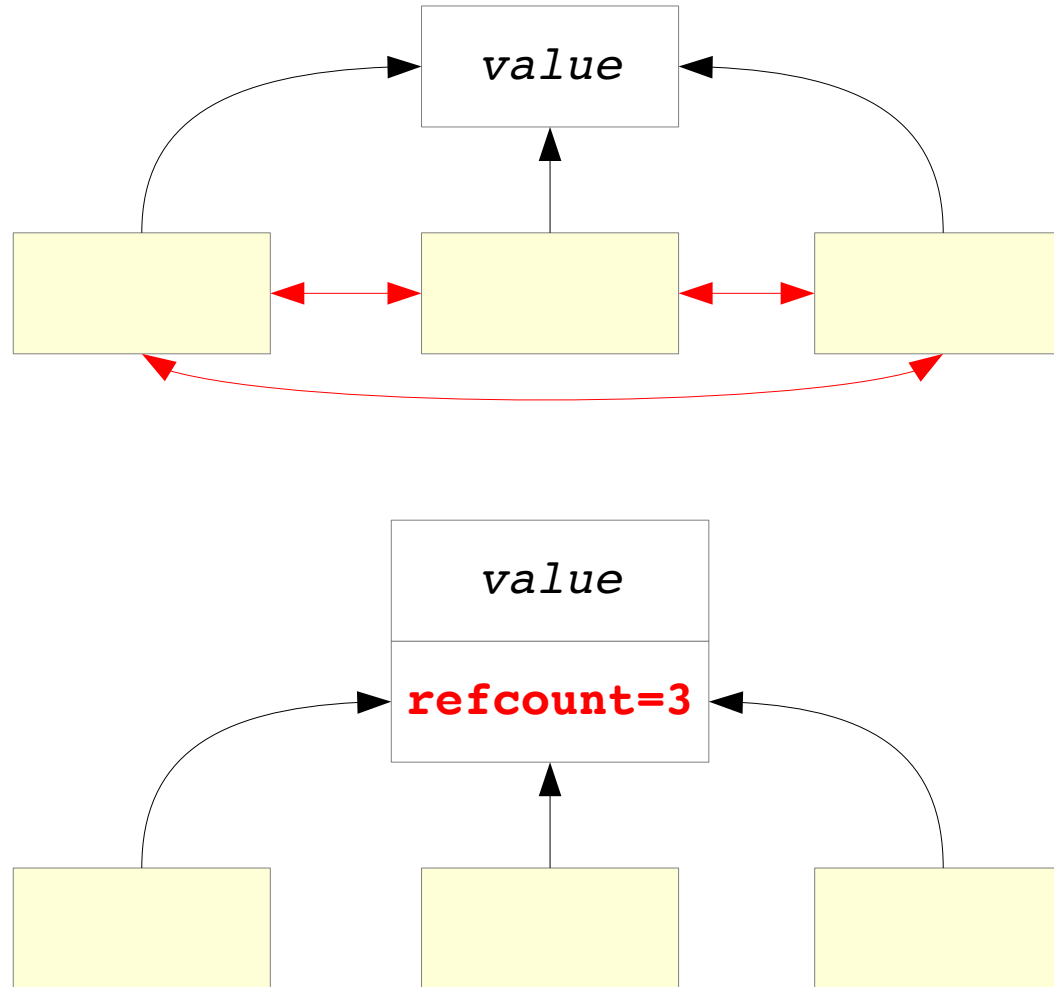
uniqueはuse_countの最適化版 (でした)

C++11策定当時、“リンクリスト方式”によるshared_ptr実装を考慮し、use_count()==1判定用に最適化実装を提供できるようuniqueメンバ関数を用意した。

- 実際には、そのようなC++処理系は登場しなかった。
(Lokiライブラリはリンクリスト方式だった)
- “参照カウント方式”のC++処理系では、use_countメンバ関数はメモリバリア効果をもたない軽量なrelaxedロード命令で実装される。

実装に即した仕様へと修正し、その振る舞いを明確化した。

リンクリスト方式 vs. 参照カウント方式



※ 実際には弱参照カウンタも存在する。図中では省略。

Deprecate `shared_ptr::unique`

`shared_ptr<T>::unique` メンバ関数

- C++17から非推奨、C++20で削除(予定※)

`shared_ptr<T>::use_count` メンバ関数

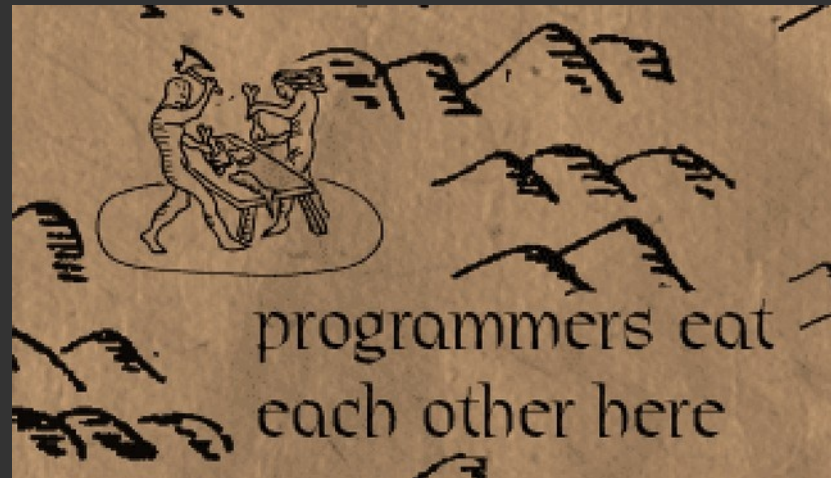
- スレッド間の“同期”効果を持たない
- シングルスレッドでは期待通り動作する

※ <https://wg21.link/p0619> Reviewing Deprecated Facilities of C++17 for C++20

COFFEE BREAK (2)



Terra Incognita (知られざる未開の地)



(この地ではプログラマ達は
互いを喰いあうのだ)

“The C++17 Lands”, 部分

<http://fearlesscoder.blogspot.com/2017/02/the-c17-lands.html>

Temporarily discourage `memory_order_consume`

`memory_order_consume`の“一時的な”非推奨化

- `memory_order_acquire`で代替可
- `consume`実装済みのC++コンパイラは存在せず
- 実装不可能な仕様のためC++20に向け絶賛議論中
- (C++メモリモデル定義からは削除せず残置)

<http://wg21.link/p0371r1> Temporarily discourage `memory_order_consume`

memory_order_xxx

atomic変数操作(Load/Store/CAS)やfence関数へと指定する、“順序性保証の強さ”を指定するオプション。

- デフォルト動作は、最も強い *mo_seq_cst* により逐次一貫性 (Sequential consistency)を保証。一般人向け。
- より弱いRelease一貫性のために *mo_release*, *mo_acquire*, *mo_acq_rel* が提供される。プロ向け。
 - データ依存性に基づく限定的な一貫性セマンティックスのために *mo_consume* が提供される。人外向け。
- 順序性保証のない *mo_relaxed* も提供され、fence関数と組み合わせて利用する。ガチプロ向け。

私立C++女学園 マルチスレッド科

(© 2011 yamasaさん)



登場人物	人物設定
<code>mo_seq_cst</code>	優等生な学級委員長。完璧主義者で、何事も全て順番どおり(total order)でないと気が済まないタイプ。
<code>mo_release</code>	シンクロナイズド(synchronized)スイミングが得意な双子の姉妹。みんなの人気者。
<code>mo_acquire</code>	
<code>mo_acq_rel</code>	双子acq/relのお姉さん。存在感が薄い不遇の子。
<code>mo_relaxed</code>	一見すると平凡な生徒だが、魔法の杖(fence)で変身する魔法少女。実際には魔法に失敗するドジっ子。
<code>mo_consume</code>	スピード狂の不良生徒。彼女のために校則さえ書換えられた。死のレース(race)に参加しているという噂...

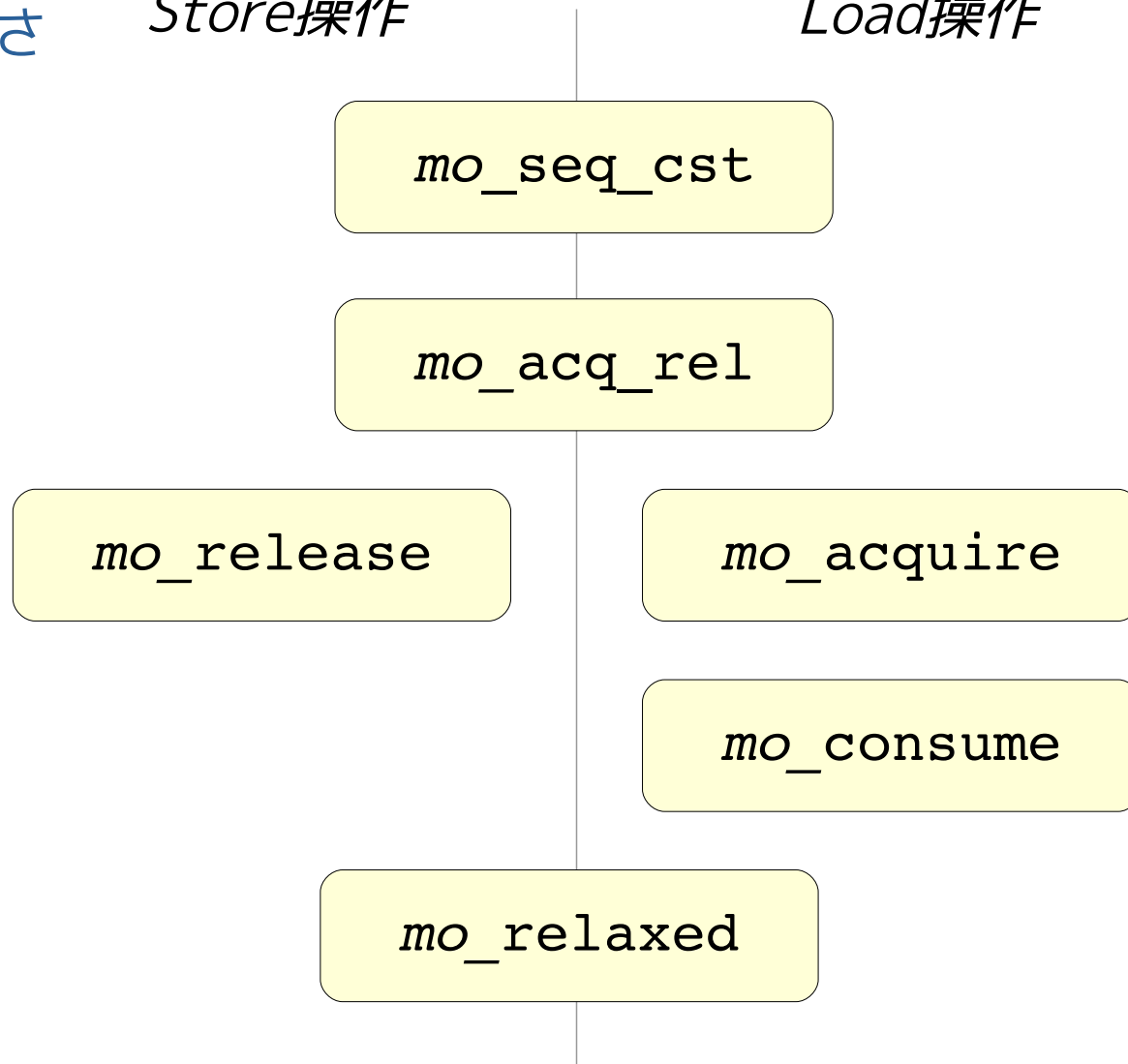
出展 <https://yamasa.hatenablog.jp/entry/20110401/1301583600>

memory_order_xxx

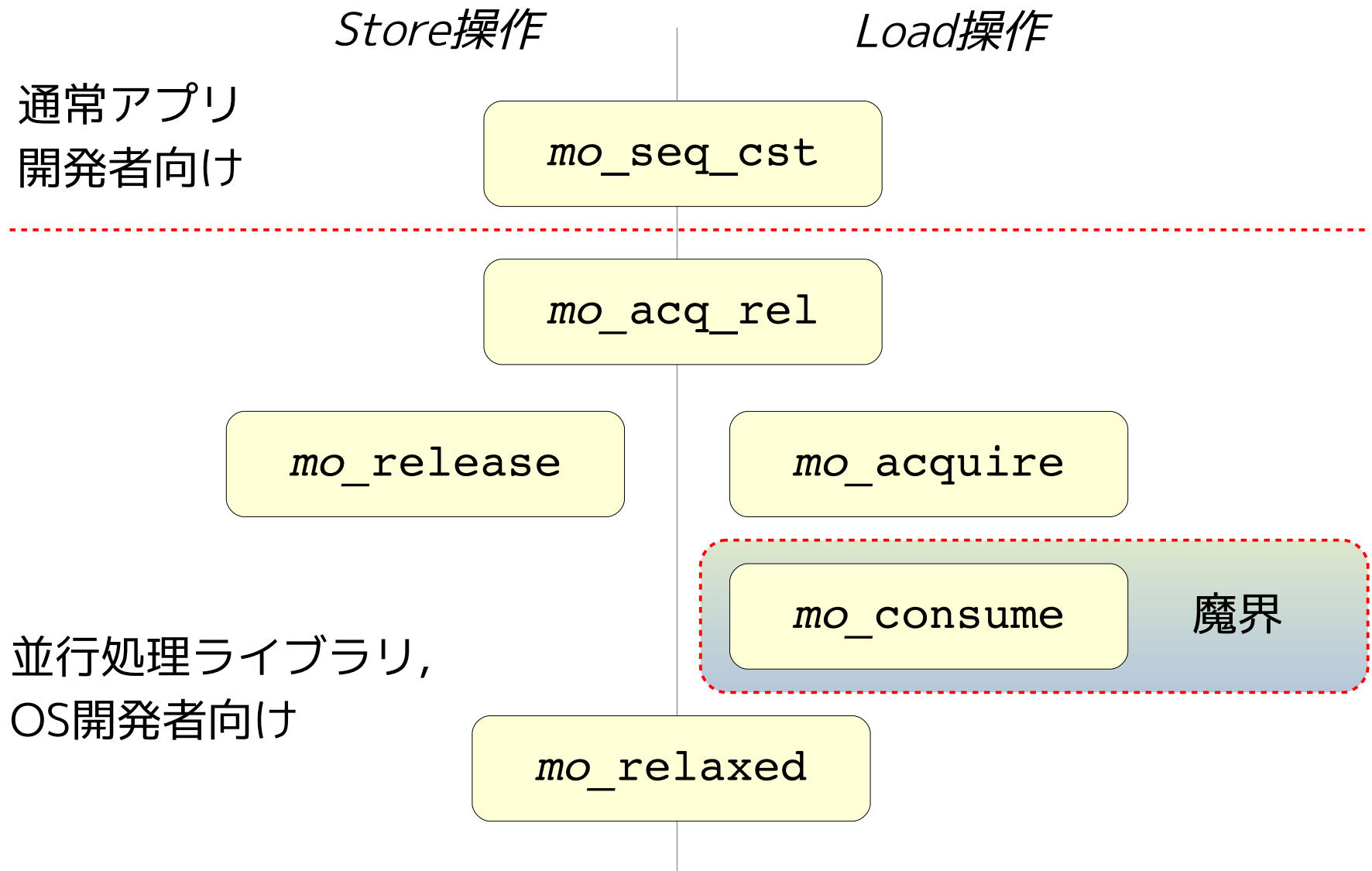
順序性
保証の強さ

Store操作

Load操作



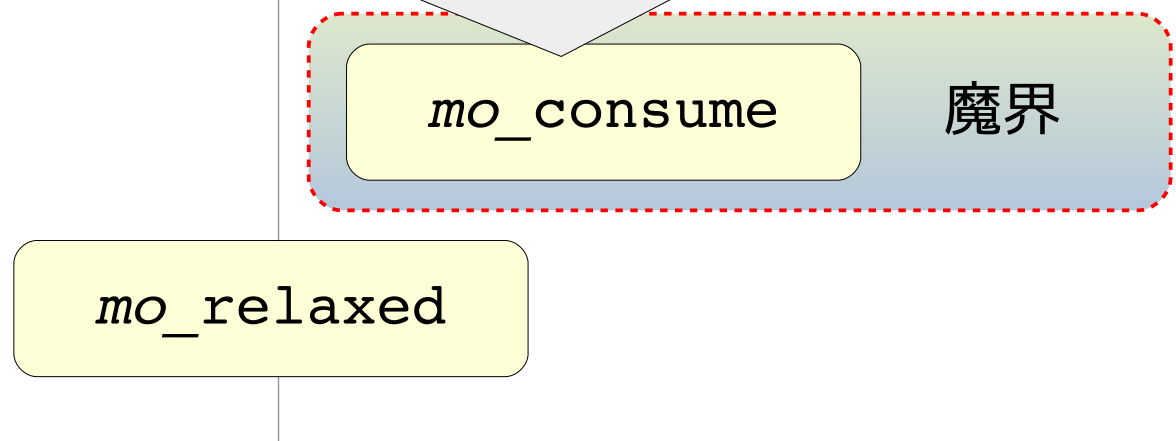
memory_order_xxx



memory_order_xxx

弱いメモリモデル(ARM, Powerなど)を対象に
ロックフリー・データ構造を実装しており
数十クロック〜の実行時オーバーヘッドを
最適化する意義があるケースを想定している
例) Linuxカーネル内部のRCUデータ構造

並行処理ライブラリ,
OS開発者向け



consumeとは何か

先人による素晴らしい解説記事を参照ください。

- C++0xのメモリバリアをより深く解説してみる

<https://yamasa.hatenablog.jp/entry/20090929/1254237835>

- The Purpose of memory_order_consume in C++11

http://preshing.com/20140709/the-purpose-of-memory_order_consume-in-cpp11/

consume実装への挑戦

かつてGCCは`mo_consume`実装に挑戦し、失敗している...

- Fixing GCC's Implementation of `memory_order_consume`

http://preshing.com/20141124/fixing-gccs-implementation-of-memory_order_consume/

- Bug 59448 - Code generation doesn't respect C11 address-dependency

https://gcc.gnu.org/bugzilla/show_bug.cgi?id=59448

2015-01-14 Andrew MacLeod <xxx@yyy.zzz>

PR middle-end/59448

* `builtins.c` (`get_memmodel`): **Promote consume to acquire always.**

* [...]

C++20に向けた関連提案

<https://wg21.link/p0098> Towards Implementation and Use of memory_order_consume

→ 旧版N4215の一部訳出 <http://d.hatena.ne.jp/yohhoy/20141115/p1>

<https://wg21.link/p0124> Linux-Kernel Memory Model

<https://wg21.link/p0190> Proposal for New memory_order_consume Definition

<https://wg21.link/p0462> Marking memory_order_consume Dependency Chains

<https://wg21.link/p0668> Revising the C++ memory model

<https://wg21.link/p0735> Interaction of memory_order_consume with release sequences

<https://wg21.link/p0750> Consume

Temporarily discourage `memory_order_consume`

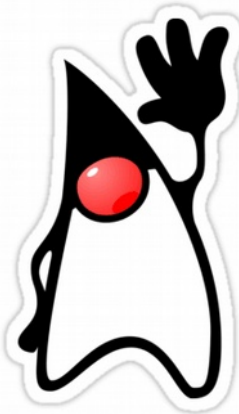
`s/memory_order_consume
/memory_order_acquire/g`

<https://timsong-cpp.github.io/cppwp/n4659/atomics.order#1.3>

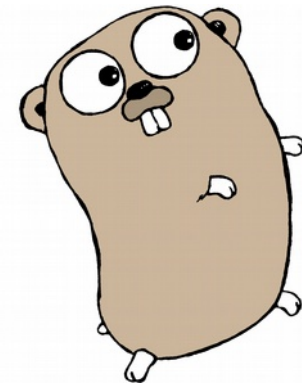
COFFEE BREAK (3)



Ferris@Rust



Duke@Java



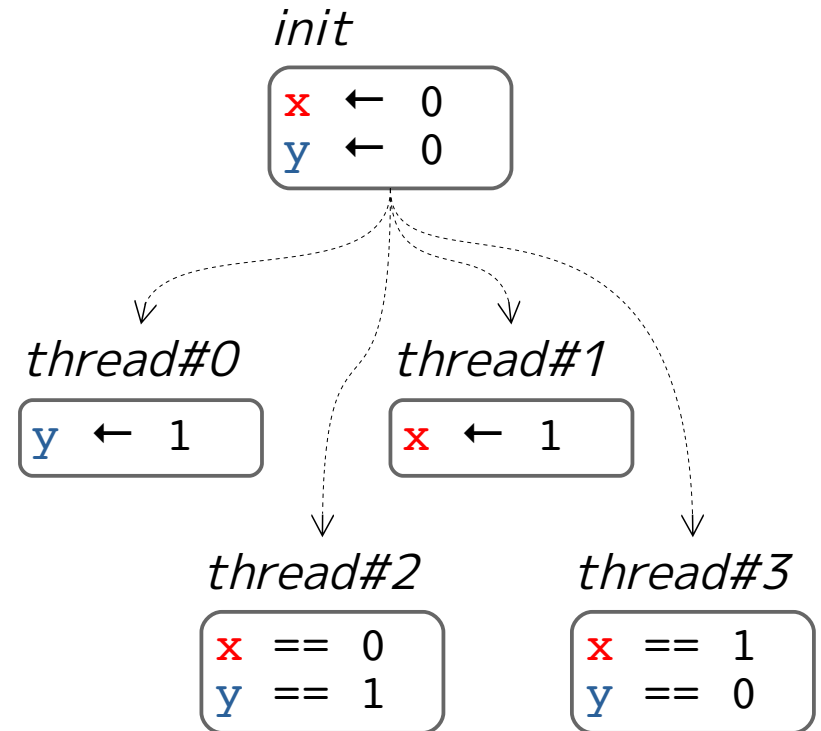
Gopher@Go

(C++メモリモデルに影響を与えた/相当仕様を持つ
プログラミング言語のマスコットの皆さん)

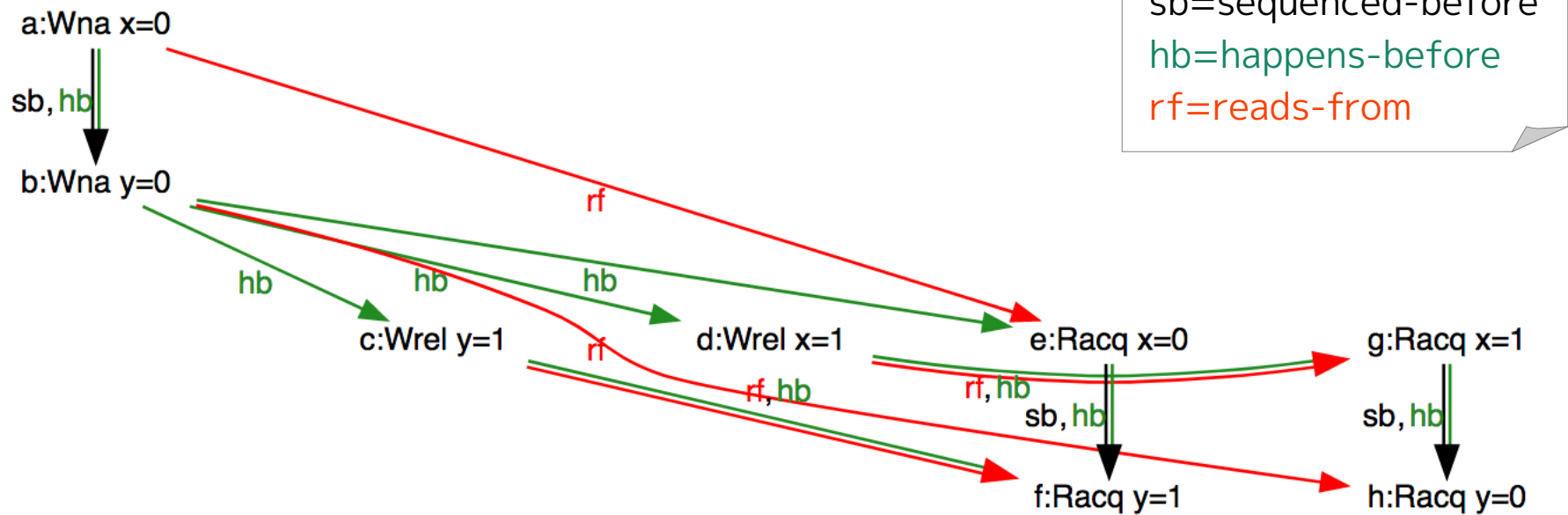
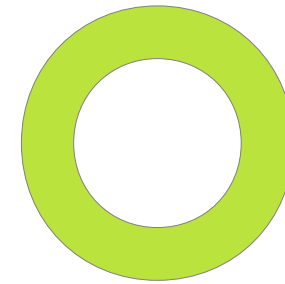
おまけQuiz#1

```
atomic<int> x = 0;
atomic<int> y = 0;

{{{ { y.store(1, mo_release); }
||| { x.store(1, mo_release); }
||| { r0 = x.load(mo_acquire); // 0
    r1 = y.load(mo_acquire); // 1
    }
||| { r2 = x.load(mo_acquire); // 1
    r3 = y.load(mo_acquire); // 0
    }
}}}
```



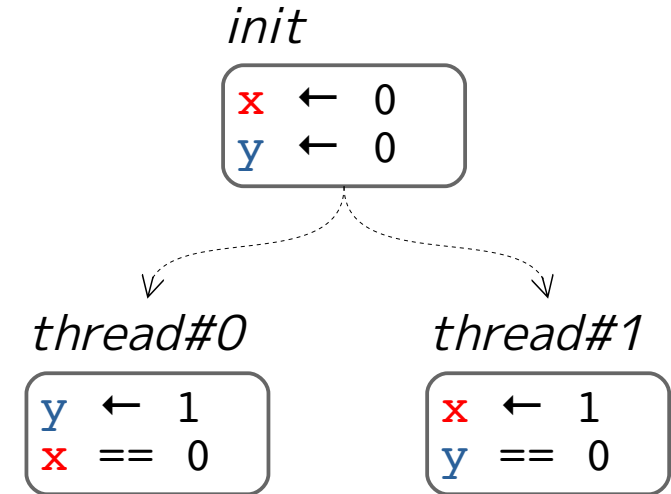
おまけQuiz#1



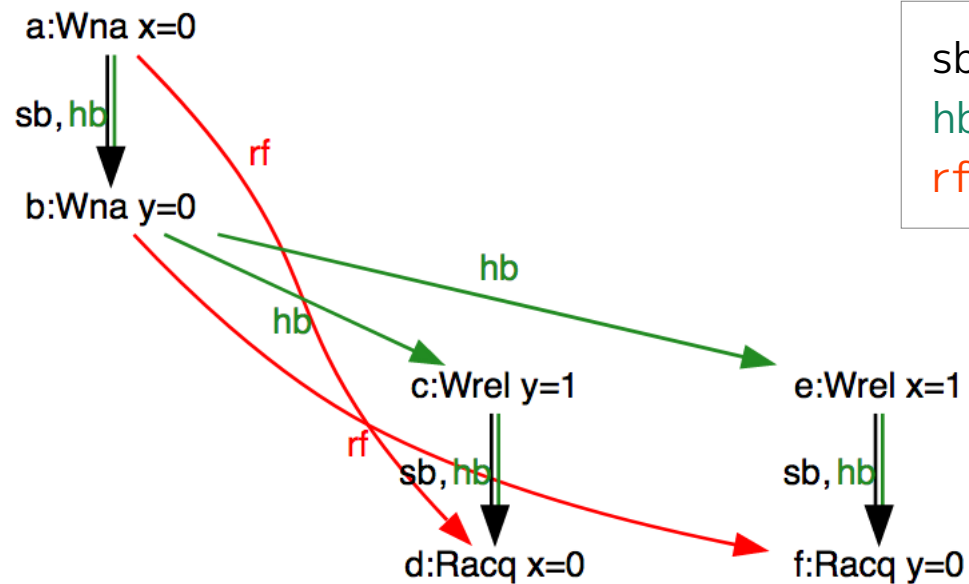
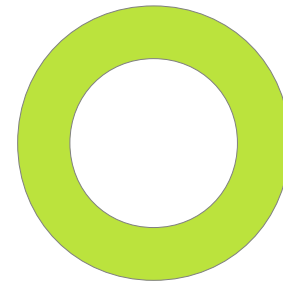
Powered by <http://svr-pes20-cppmem.cl.cam.ac.uk/cppmem/>

おまけQuiz#2-1

```
atomic<int> x = 0;
atomic<int> y = 0;
{{{ { y.store(1, mo_release);
      r1 = x.load(mo_acquire); // 0
    }
||| { x.store(1, mo_release);
      r2 = y.load(mo_acquire); // 0
    }
}}}
```



おまけQuiz#2-1

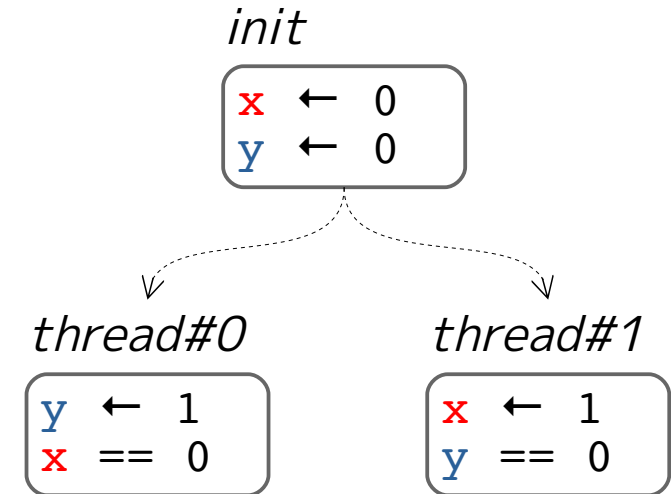


sb=sequenced-before
hb=happens-before
rf=reads-from

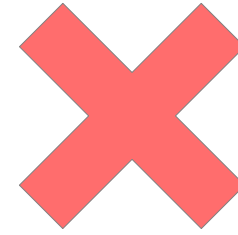
Powered by <http://svr-pes20-cppmem.cl.cam.ac.uk/cppmem/>

おまけQuiz#2-2

```
atomic<int> x = 0;
atomic<int> y = 0;
{{{ { y.store(1, mo_seq_cst);
      r1 = x.load(mo_seq_cst); // 0
    }
||| { x.store(1, mo_seq_cst);
      r2 = y.load(mo_seq_cst); // 0
    }
}}}
```



おまけQuiz#2-2



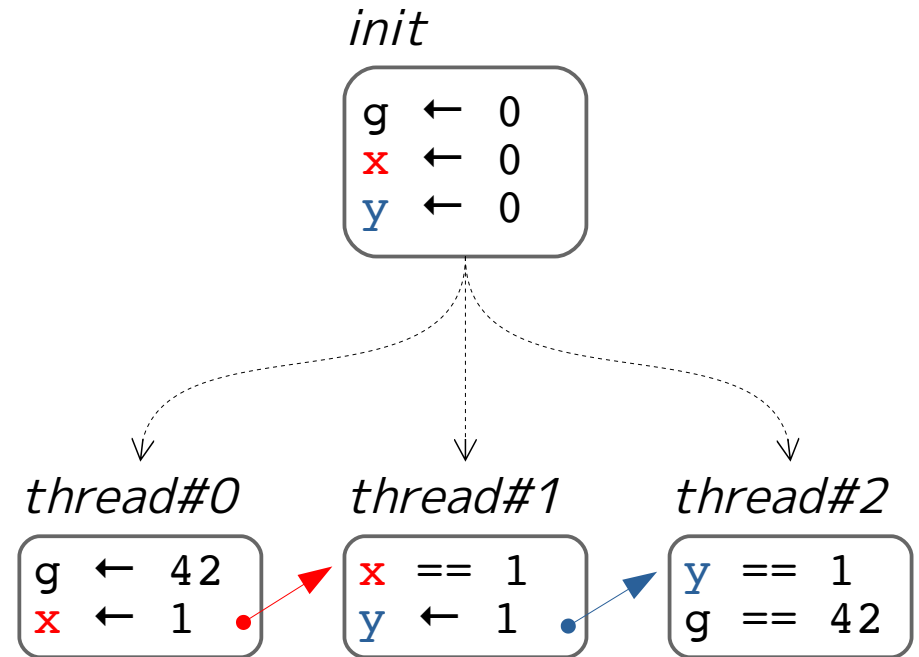
```
atomic<int> x = 0;
atomic<int> y = 0;
{{{ { y.store(1, mo_seq_cst);
      r1 = x.load(mo_seq_cst);
    }
||| { x.store(1, mo_seq_cst);
      r2 = y.load(mo_seq_cst);
    }
}}}
```

*mo_seq_cst*による逐次一貫
実行では $\{r1, r2\} = \{0, 0\}$ とい
う結果には決してならない。

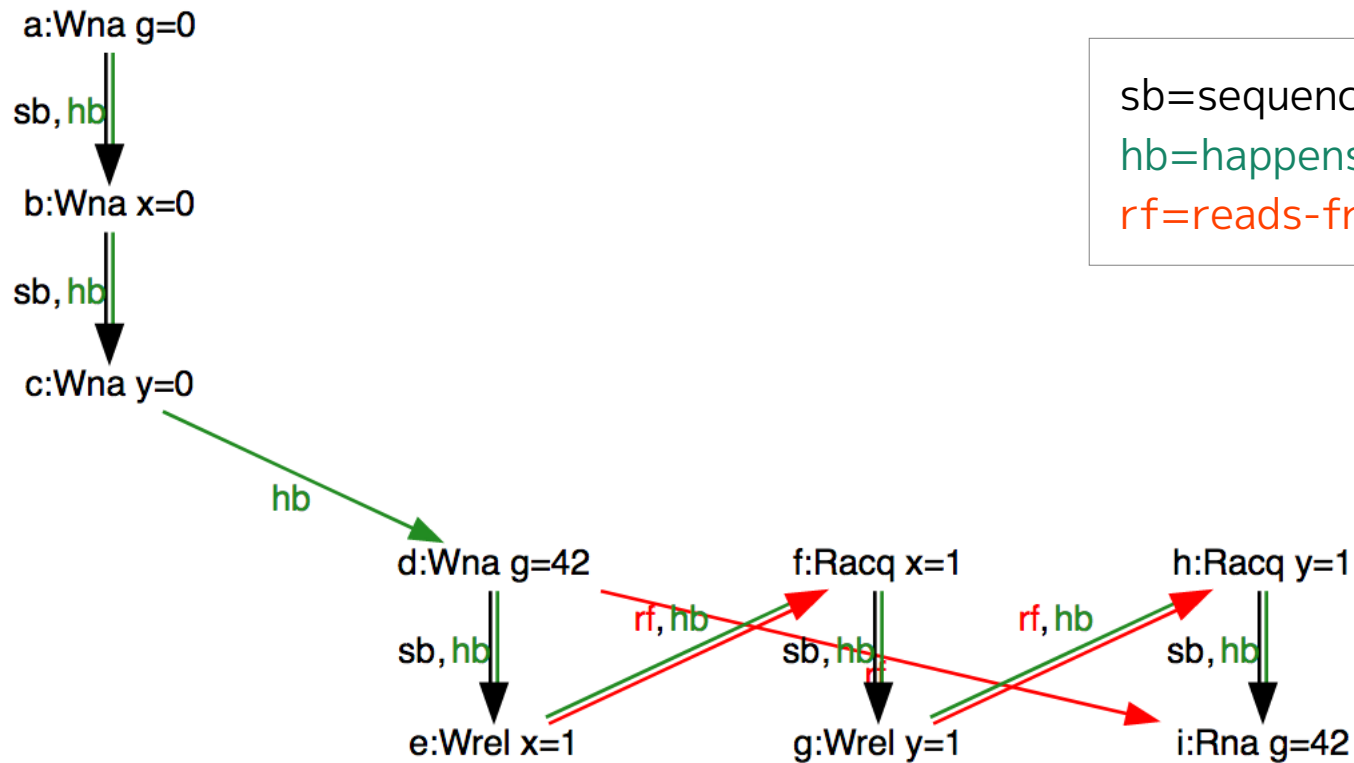
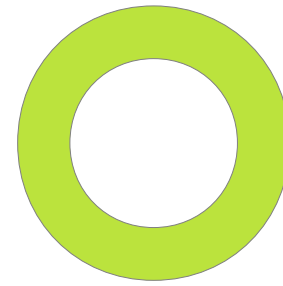
$\{r1, r2\}$ は $\{0, 1\}$, $\{1, 0\}$, $\{1, 1\}$
のいずれかの結果になる。

おまけQuiz#3-1

```
int g = 0;
atomic<int> x = 0;
atomic<int> y = 0;
{{{ { g = 42;
      x.store(1, mo_release); }
||| { r0 = x.load(mo_acquire); //==1
      y.store(mo_release);
    }
||| { r1 = y.load(mo_acquire); //==1
      r2 = g; // 42 ?
    }
}}}
```



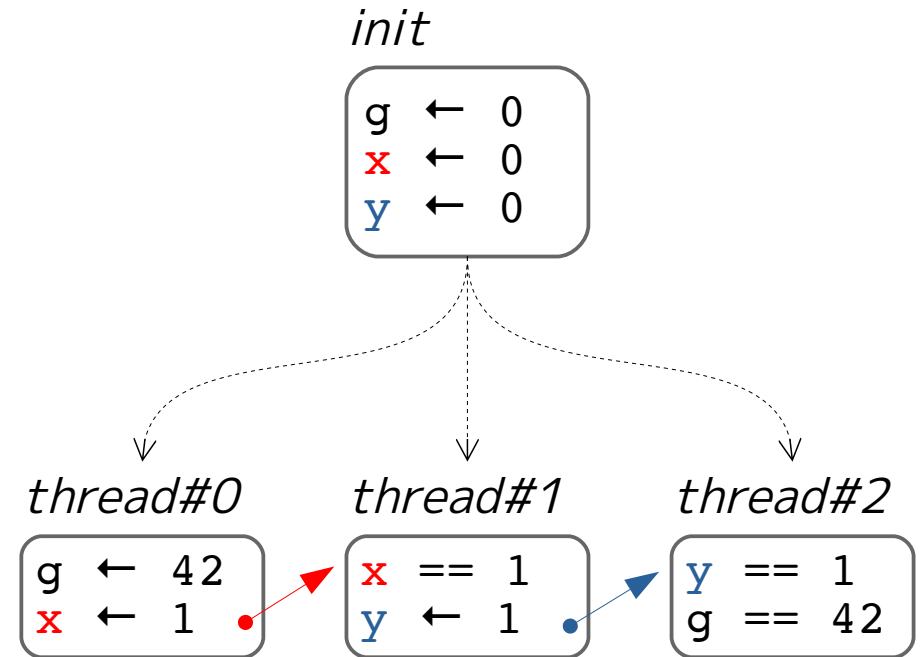
おまけQuiz#3-1



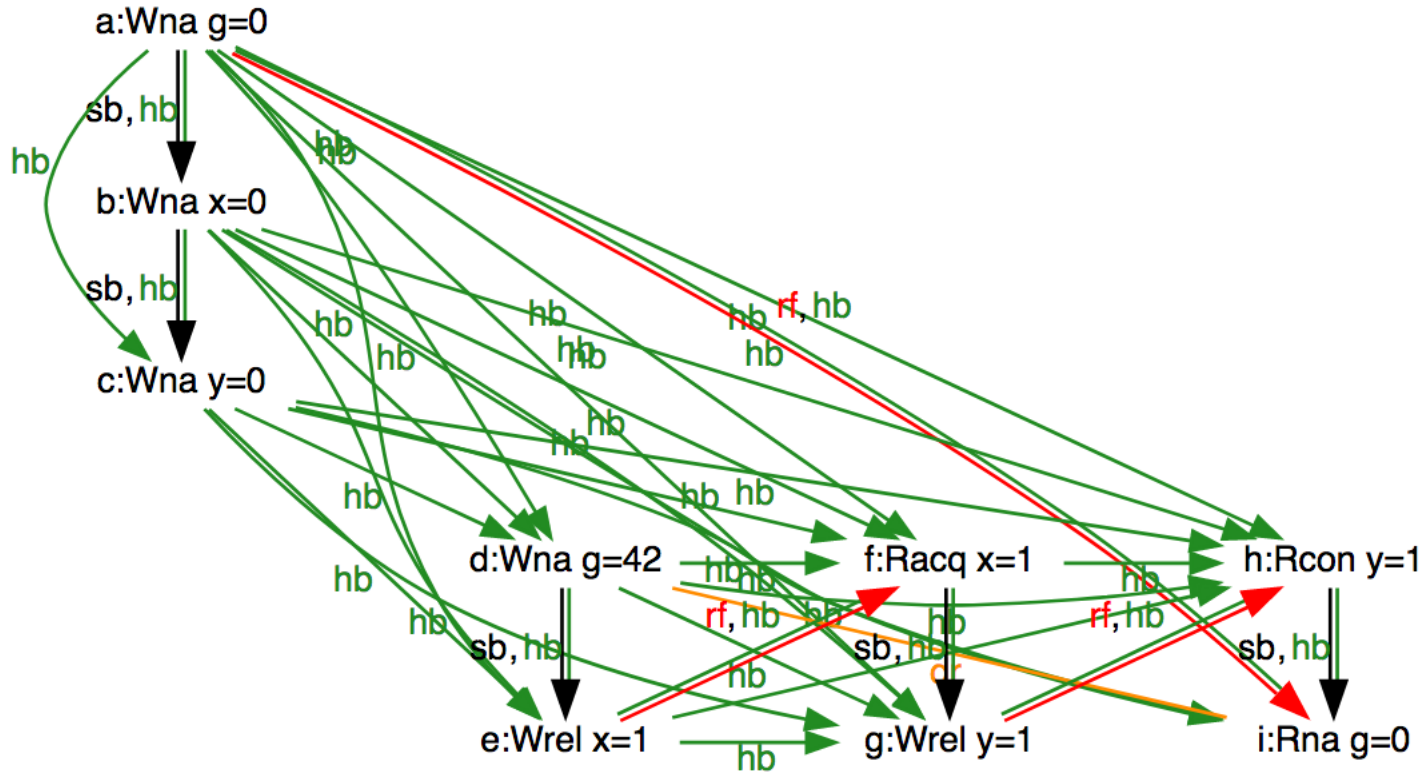
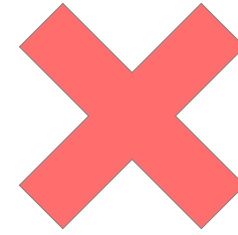
Powered by <http://svr-pes20-cppmem.cl.cam.ac.uk/cppmem/>

おまけQuiz#3-2

```
int g = 0;
atomic<int> x = 0;
atomic<int> y = 0;
{{{ { g = 42;
      x.store(1, mo_release); }
||| { r0 = x.load(mo_acquire); //==1
      y.store(mo_release);
    }
||| { r1 = y.load(mo_consume); //==1
      r2 = g; // 42 ?
    }
}}}
```

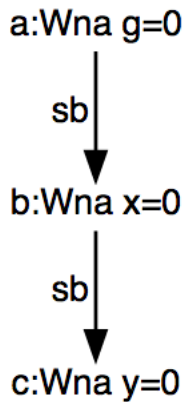
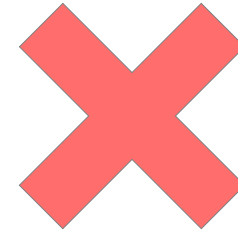


おまけQuiz#3-2

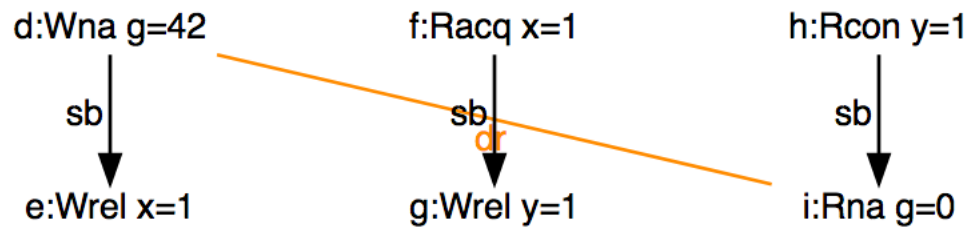


Powered by <http://svr-pes20-cppmem.cl.cam.ac.uk/cppmem/>

おまけQuiz#3-2



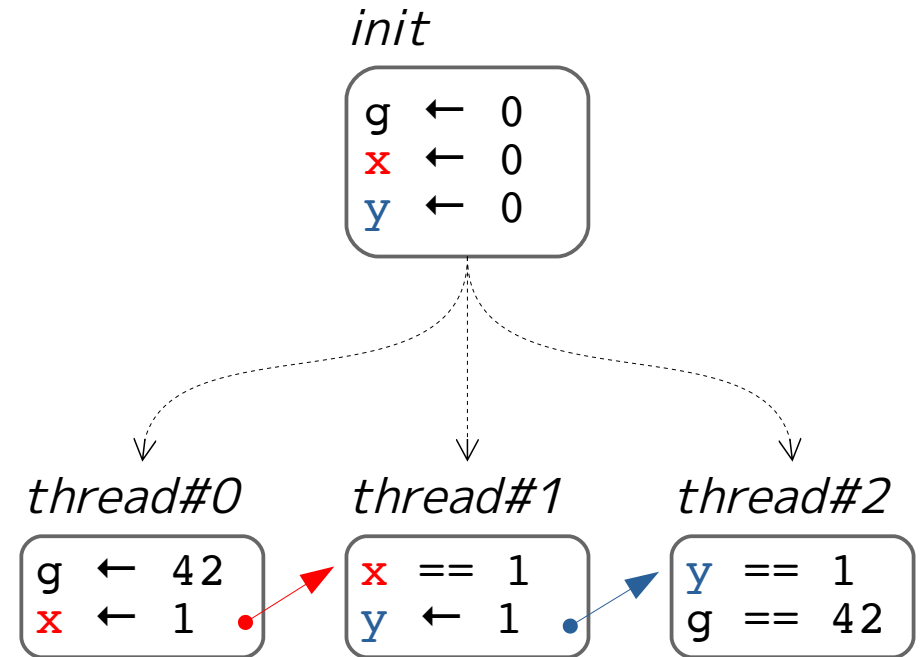
操作d, i間にhb関係がなく、
通常変数gに対する書込/読取
でデータ競合が発生する。



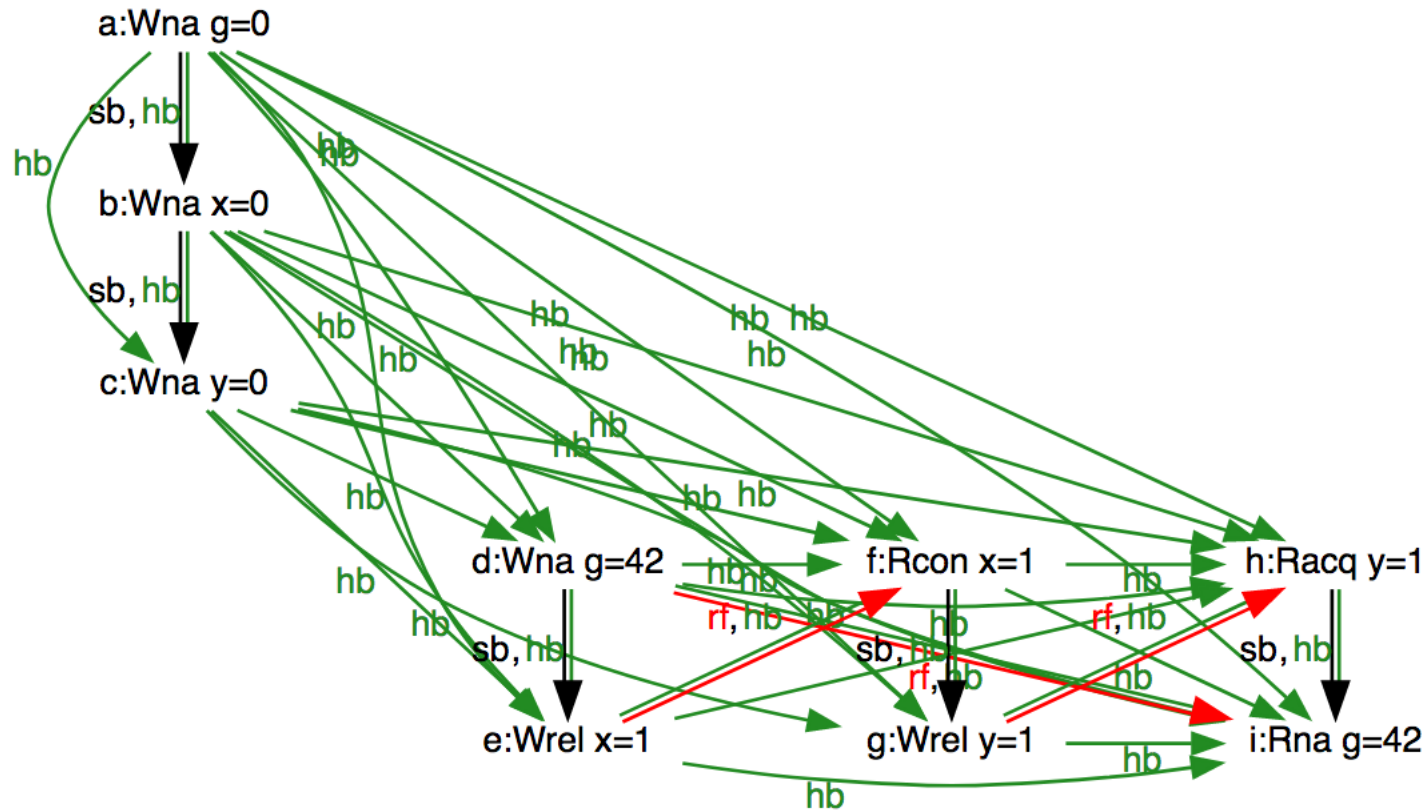
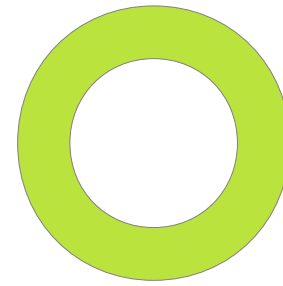
Powered by <http://svr-pes20-cppmem.cl.cam.ac.uk/cppmem/>

おまけQuiz#3-3

```
int g = 0;
atomic<int> x = 0;
atomic<int> y = 0;
{{{ { g = 42;
      x.store(1, mo_release); }
||| { r0 = x.load(mo_consume); //==1
      y.store(mo_release);
    }
||| { r1 = y.load(mo_acquire); //==1
      r2 = g; // 42 ?
    }
}}}
```



おまけQuiz#3-3



Powered by <http://svr-pes20-cppmem.cl.cam.ac.uk/cppmem/>