そろそろvolatileについて一言いっておくか

自己紹介

- H/N: yamasa
- FreeBSD使い
- Java屋
- モバイル大好き

今日のキーワード

- アトミック変数
- メモリバリア
- happens before
- 正しく同期化されたコード
- volatile

マルチスレッドと排他制御

マルチスレッドプログラミングで苦労する点

- 同じ変数やオブジェクトが複数のスレッドから同時にアクセスされる可能性がある。

mutex などの排他制御(ロック)を使って対処

- あらゆる変数を排他制御しなければならないの?
  ロックによるパフォーマンスの劣化も気になる。

⇒ もっと軽量の仕組みはないの?

スレッド間通信のprimitive

- スレッド間でデータをやり取りするための最も基本的な仕組み(primitive)

それが…~~volatile~~

アトミック変数とメモリバリア

アトミック変数とは

「アトミック変数」とは

- 複数スレッドが同時に読み書きなどの操作を行っても、それらの操作が順々に行われているように見えることが保証されている変数

アトミック変数へのloadとstore (1)

```
// 初期値
atomic<int> a = -1
```

スレッド1:

```
// aに1を代入(store)
a.store(1)
```

スレッド2:

```
// aの値を読み出す(load)
r = a.load()
```

r の値は -1 または 1 のどちらかになることが**保証される**

アトミック変数へのloadとstore (2)

// 初期値
atomic<int> a = 0

スレッド1:
a.store(1)

スレッド2:
a.store(-1)

a の値は
　0 ⇒ 1 ⇒ -1
　0 ⇒ -1 ⇒ 1
のいずれかの順序で変化することが**保証される**

load, storeのアトミック性

- 当たり前のようだが非常に大切な性質
  - アトミック性が保証されない変数に対して同様にアクセスした場合、たとえば上位bitと下位bitが混ざった値になってしまうかも…

    (例: 32bitアーキテクチャでの int64_t (long long) 型変数への読み書き)

アトミック変数への複雑な操作

- 多くのアトミック変数の実装には、読み込みと書き込みを同時に行う操作も提供されている

r1 = a.exchange(r2)

⇒ { r1 = a; a = r2; } // 読み込みと同時に書き込み

r1 = a.fetch_add(r2)

⇒ { r1 = a; a = a + r2; } // 読み込みと同時に加算

など…

アトミック変数のもう一つの性質

- アトミック変数への書き込みは、いずれ必ず他スレッドからも見えるようになる。

```
atomic<int> a = 0;
```

スレッド1:

```
a.store(1);
```

スレッド2:

```
while (a.load() != 1) {
}
```

この場合、スレッド2のwhile文は無限ループにならないことが保証される。

アトミック変数と普通の変数の違い

- アトミック変数は、複数のスレッドからアクセスされても大丈夫なように設計された変数
  - ⇒ アトミックではない普通の変数は、マルチスレッドでは予期せぬ実行結果になることがある。
  - マルチスレッドプログラムでは全ての変数をアトミック変数にしなければならないの？
  - ⇒ No!
- 普通の変数についても、マルチスレッドでの動作を保証する仕組みが別に存在する！

- 以後の説明では変数を以下のように表記します

a, a1, a2 …

　　アトミック変数

x, y, z

　　スレッド間でアクセスされる可能性がある

　　アトミック変数以外の普通の変数

r, r1, r2 …

　　ローカル変数(レジスタ)

また、明示的に示さない場合の各変数の初期値は0とします

リオーダーとは？

「リオーダー」

- プログラム実行速度の向上などの目的で実行順序を入れ替える、最適化手法の一つ
  - コンパイラがプログラムコードを機械語に変換するとき
  - CPUが機械語のコードを実際に実行するとき

  キャッシュメモリの影響などを考慮して命令の実行順序を入れ替える

リオーダーが出来ない状況

- 当然ながら、実行結果が変わってしまうようなリオーダーは禁止

```
x = 1;        // …①
y = x;        // …②
```

⇒ ①と②を入れ替えるとyに代入される値が変わってしまうのでダメ！

⇒でも、②の右辺を定数1に置換する最適化はOK

リオーダーできる例

- では、以下の例はリオーダー可能？

```
// 例1
x = 1;        // …①
y = 1;        // …②´

// 例2
r1 = y;       // …③
r2 = x;       // …④
```

最終的な実行結果は変わらないので、
①と②´、③と④をそれぞれ入れ替えるのはOK

…ただし、シングルスレッドに限る

- 前述の例をマルチスレッド化すると…

```
// スレッド1                    // スレッド2
x = 1;        // …①            r1 = y;        // …③
y = 1;        // …②´           r2 = x;        // …④
```

- r1 == 1 && r2 ==0 となることはありえない、…はず。
- でも、リオーダーを考慮すると、ありえる。

マルチスレッドでの最適化の困難さ

- マルチスレッドでは、ちょっとした最適化でも実行結果に影響を及ぼす。

⇒ マルチスレッドでの実行結果にも影響しないようにリオーダーなどを制限すると、最適化する余地がほとんど無くなってしまう!!

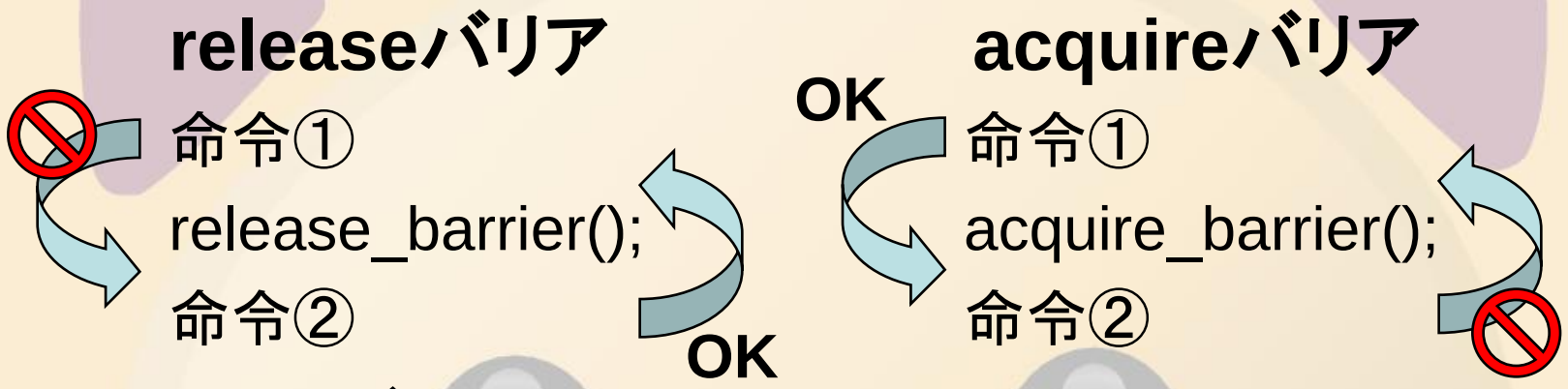
マルチスレッドでの最適化の原則

よって、最適化の規則を以下のように定める。

- シングルスレッドでの実行結果が変わらない限り、どんな最適化も基本的には許可する。
- 最適化がマルチスレッドでの動作に悪影響を及ぼさないよう、最適化を制限する手段を処理系は提供する。

  ⇒ それが「**メモリバリア(メモリフェンス)**」

メモリバリアの基本は2種類



- releaseバリア

  先行する命令が、バリアを超えて後ろにリオーダーされるのを禁止する。

- acquireバリア

  後続の命令が、バリアを超えて前にリオーダーされるのを禁止する。

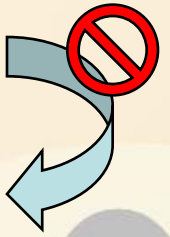
アトミック変数とメモリバリアの組み合わせ

- アトミック変数とメモリバリアは一緒に用いる

アトミック変数へのstore ＋ releaseバリア

処理①

a.store_release(r);



アトミック変数からのload ＋ acquireバリア

r = a.load_acquire();

処理②

- アトミック変数とメモリバリアを組み合わせることで、異なるスレッドの命令間に順序付けをすることができる。

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

スレッド1:

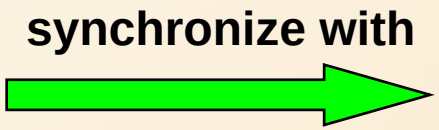
```
x = 1;      // …①
a.store_release(1);
```

スレッド2:

```
r1 = a.load_acquire();
if (r1 == 1) {
    r2 = x;     // …②
}
```

スレッド1:

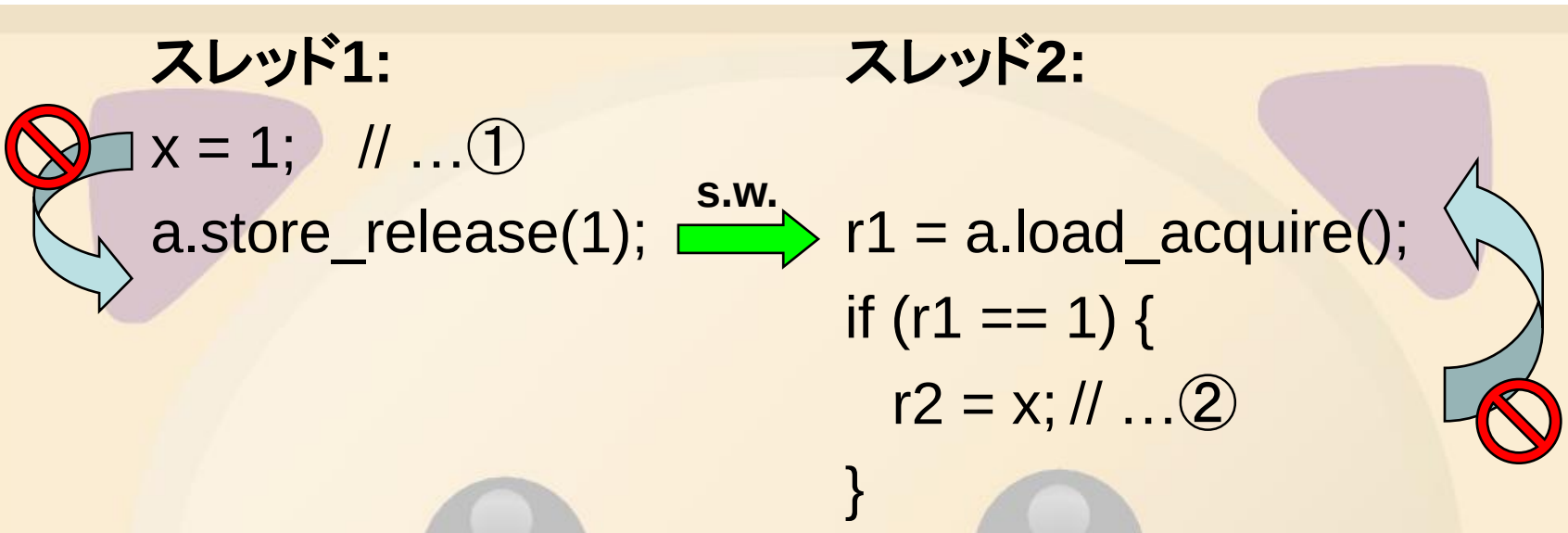
```
x = 1;  // …①
a.store_release(1);
```



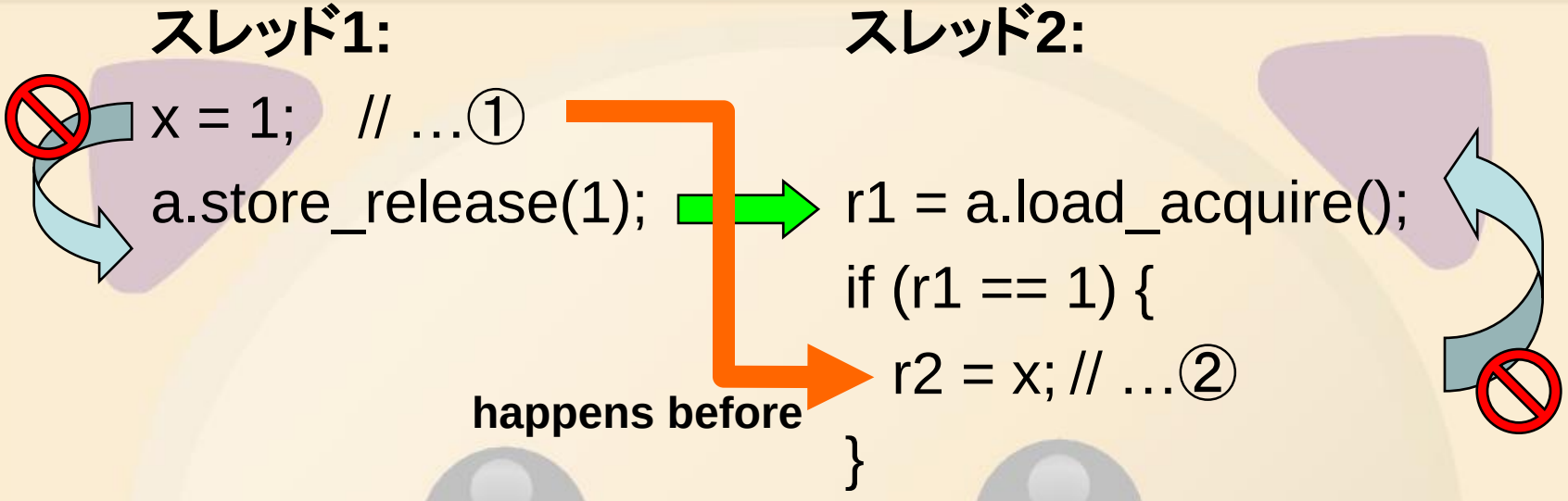
スレッド2:

```
r1 = a.load_acquire();
if (r1 == 1) {
        r2 = x; // …②
}
```

- store_release()で書き込んだ値を load_acquire()で読み込むことで、2つの間に前後関係が生まれる。
- この関係を **“synchronize with”** と呼ぶ。



- releaseバリアのため、①の書き込みは必ずstore_release() より前に行われる。
- acquireバリアのため、②の読み込みは必ずload_acquire() の後に行われる。
  （要するに、xの値の先読みは禁止！）



- synchronize with関係とメモリバリアの効果により、①と②の間に保証された順序関係が生じる。
- この関係(“**happens before**” と呼ぶ)が存在することにより r2 == 1 が保証される。

推移的なhappens before関係(1)

スレッド1:

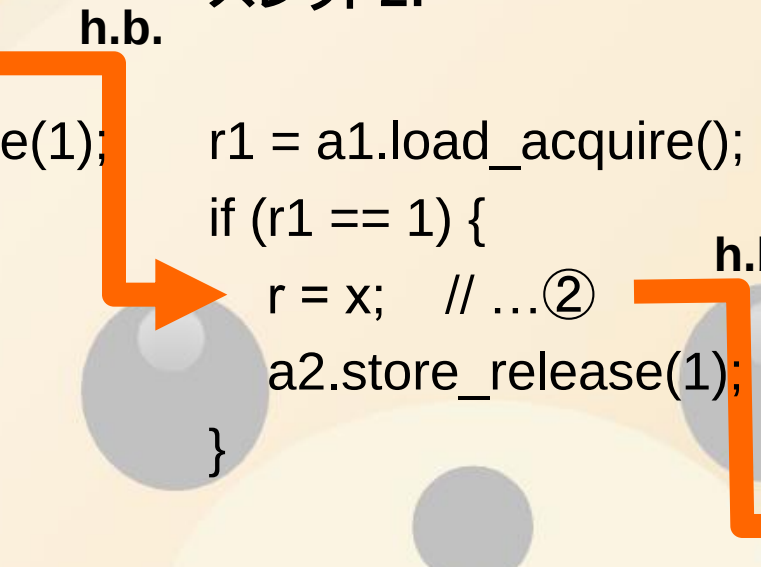
```
x = 1;    // …①
a1.store_release(1);
```

スレッド2:

```
r1 = a1.load_acquire();
if (r1 == 1) {
    r = x;    // …②
    a2.store_release(1);
}
```

スレッド3:

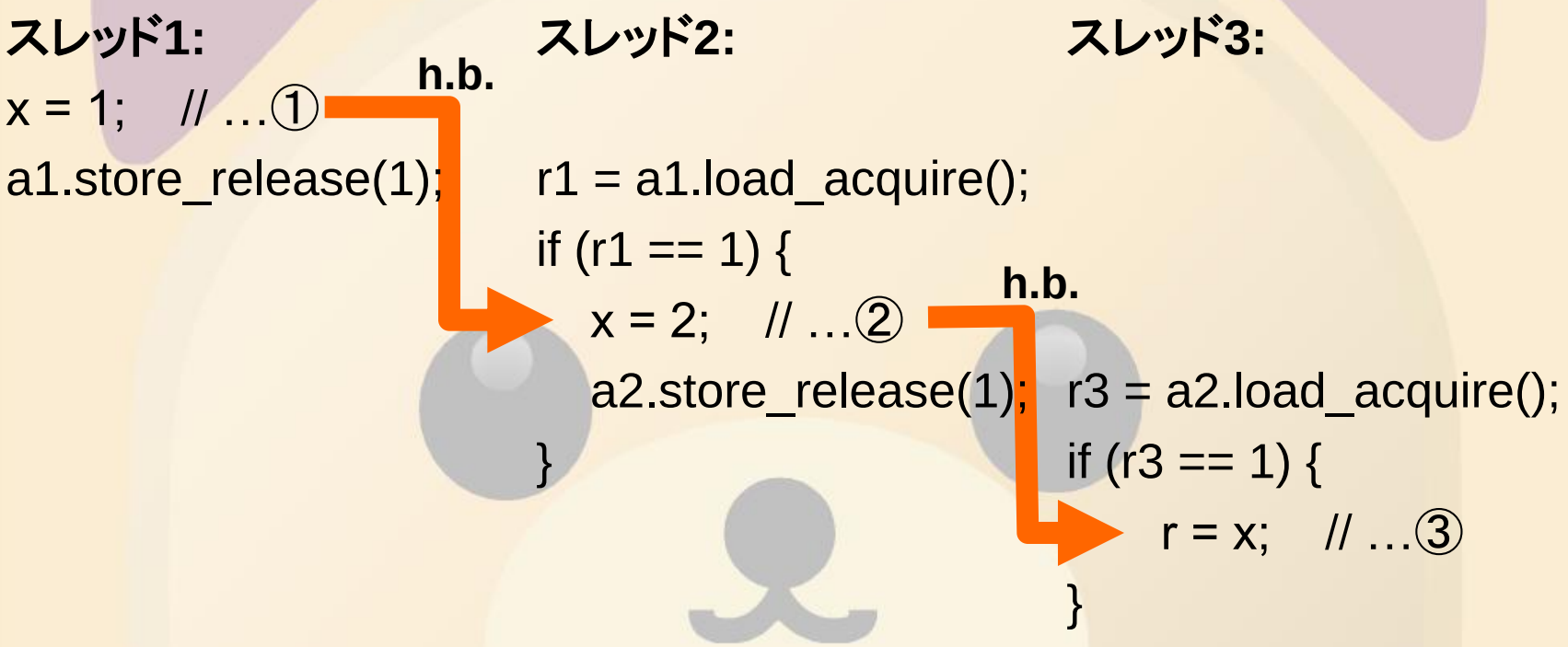
```
r2 = a2.load_acquire();
if (r2 == 1) {
    x = 2;    // …③
}
```



- ①の書き込みは②より前だが、③の書き込みは②より後。

  ⇒ よって **r == 1** となる。

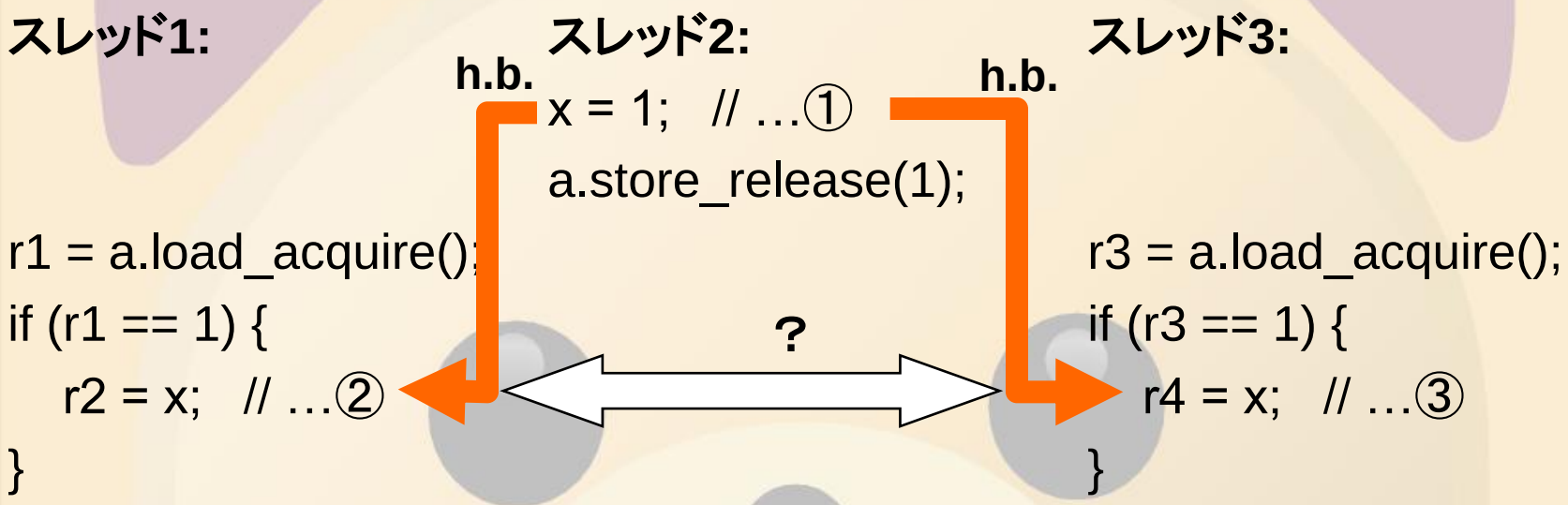
推移的なhappens before関係(2)



- ①で書き込まれた値は、その後の②で書き込まれた値によって上書きされる。

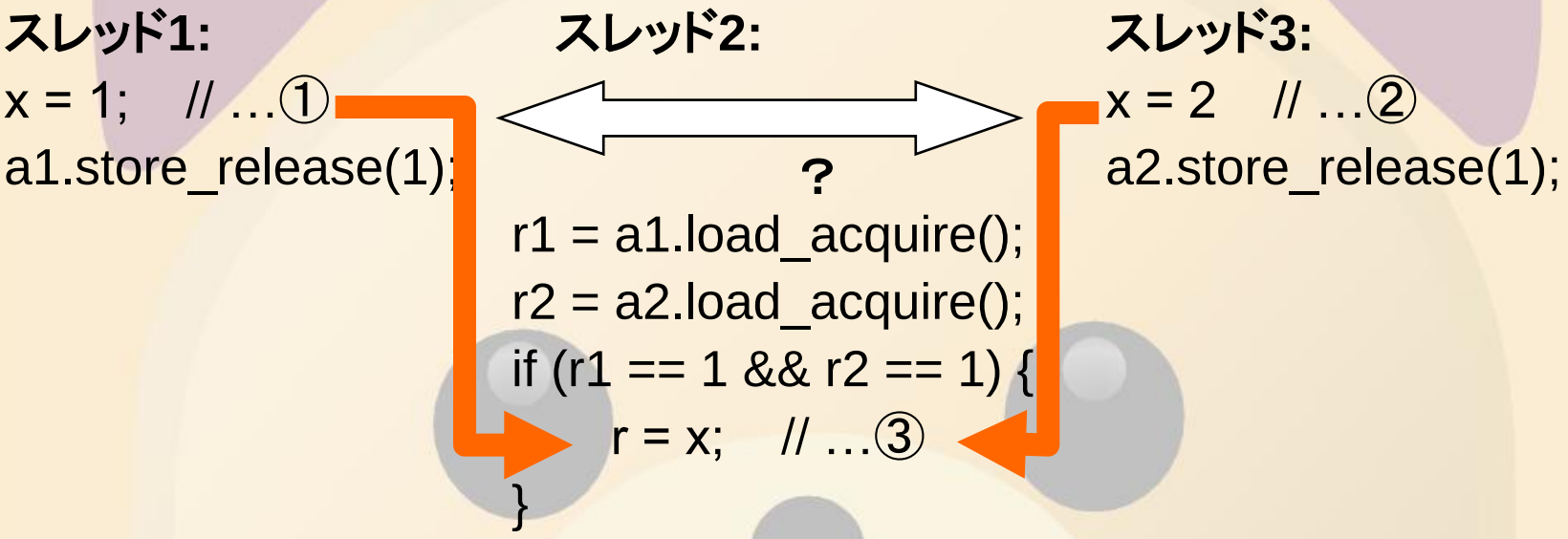
  ⇒ よって **r == 2** となる。

happens before関係がない場合(1)



- ②も③も、①より後なので、r2 == 1 かつ r4 == 1 となる。
- ②と③の間には happens before の関係が無いが、問題ない。

happens before関係がない場合(2)



- ①と②の間には happens before の関係がない。
  ⇒このような状態を “data race” と呼ぶ。

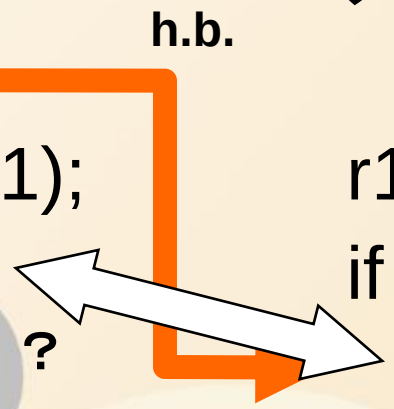
happens before関係がない場合(3)

スレッド1:

```
x = 1;      // …①
a.store_release(1);
x = 2;      // …③
```

スレッド2:

```
r1 = a.load_acquire();
if (r1 == 1) {
    r = x;      // …②
}
```



- ②と③の間には happens before の関係がない。
  ⇒これも “data race” である。

data raceとは

- アトミック変数ではない普通の変数に対して、異なるスレッド上からの
  - 書き込みと書き込み　　または
  - 書き込みと読み込み

  があり、2つの間に happens before の関係がない場合を **“data race”** と呼ぶ。

data race が起きると?

- data race の結果はどうなるのか?
  - Javaの場合

    「あらゆる最適化を考慮した上で起こり得る、全ての結果候補のうちのいずれか」
  - C++の場合

    undefined (未定義)。「何が起こるかわからない」
- 要するに、「data raceが起きた時点で負け」
  - data race が起きないプログラムコードのことを、**「正しく同期化されている」**と呼ぶ

正しく同期化されたコードを書くには

- 正しく同期化されたコードを書くための三か条
  - 変更したデータを、他スレッドからもアクセスできるようにするときは、releaseバリアを発行する。
  - 他スレッドが変更したデータへアクセスするときには、acquireバリアを発行する。
  - あるデータを他スレッドが読み書きしているタイミングでは、そのデータへの書き込みを行わないようにする。

正しく同期化されたコードの例(1)

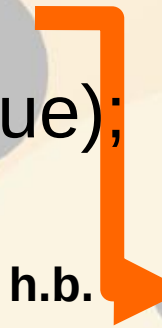
```
Hoge* x;
atomic<bool> a = false;
```

スレッド1:

```
x = new Hoge();
a.store_release(true);
```

スレッド2:

```
while (!a.load_acquire()) {
}
x->foo();
```



- 正しく同期化されているので、Hogeオブジェクトへはスレッド2から安全にアクセスできる。

正しく同期化されたコードの例(2)

```
atomic<Hoge*> a = NULL;
```

スレッド1:

```
Hoge* r1 = new Hoge();
a.store_release(r1);
```

スレッド2:

```
Hoge* r2;
do {
    r2 = a.load_acquire();
} while (r2 == NULL);
r2->foo();
```

- 正しく同期化されているので、Hogeオブジェクトへはスレッド2から安全にアクセスできる。

スピンロックへの応用

アトミック変数とメモリバリアを用いると、
いわゆる「スピンロック」も実現できる。

```
atomic<bool> a = false;

void spin_lock() {
  while(a.exchange_acquire(true)) { }
}

void spin_unlock() {
  a.store_release(false);
}
```

exchangeは、書き込むと同時にその直前の値を返す操作

それぞれ acquire, release のメモリバリアを持っていることに注目

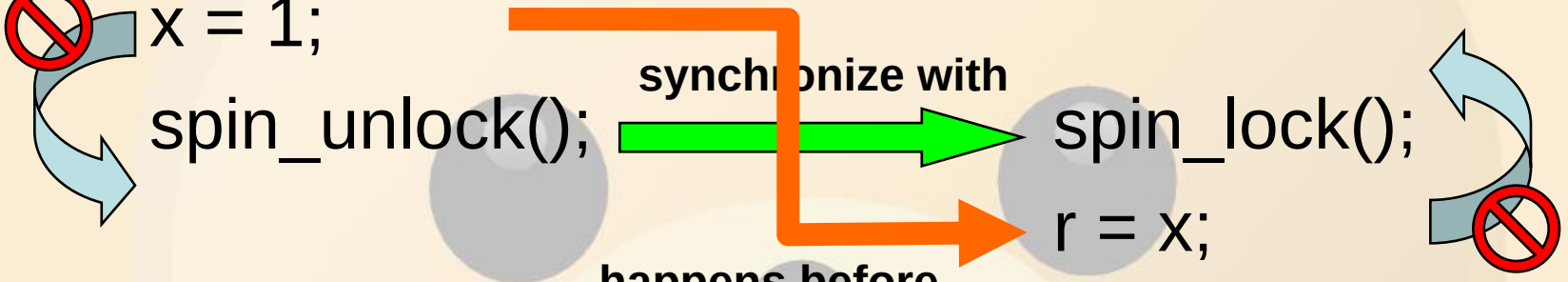
スピンロックによる同期化

スレッド1:

```
spin_lock();
x = 1;
spin_unlock();
```

スレッド2:

```
spin_lock();
r = x;
spin_unlock();
```



- スピンロックのもつメモリバリア効果により happens before関係が生まれるので、変数を安全に共有できる。

排他制御とメモリバリア

- スピンロックに限らず、スレッド間の排他制御や同期化を行う仕組み(mutexやセマフォ等)は、メモリバリア効果も持っている。

⇒ これらの仕組みを正しく使うことでも、安全なデータ共有を行うことが可能。

  - “acquireバリア”, “releaseバリア” という名称も、ロック取得と開放のもつメモリバリア効果に対応して名付けられている。

ここまでのまとめ

- 最適化の影響を考えずにマルチスレッドなプログラムを書くと、予期しない実行結果になることがある。
- アトミック変数とメモリバリアを使って正しく同期化されたコードを書けば、マルチスレッドでも安全にデータ共有を行うことができる。
- ロックなどの同期化機構も、アトミック変数とメモリバリアで構築されている。

volatileの出番は?

- ここまでアトミック変数とメモリバリアについて説明してきました。

  「ところで、volatile変数ってアトミック変数やメモリバリアの代わりにならないの?」

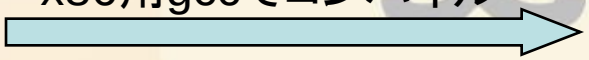
  ⇒ 答: C/C++においては**「ならない」**

volatileの問題点その1

- volatile変数はアトミック性が保証されない
  ⇒ ++ や += などの演算子はもちろん、単なるloadやstoreも型によってはアトミックとならない。

```
volatile long long  v;

v = 5;
```




```
movl  $5, v
movl  $0, v+4
```

64bit変数への代入が2命令に分かれてしまう。

volatileの問題点その2

- volatile変数はメモリバリア効果を持たない

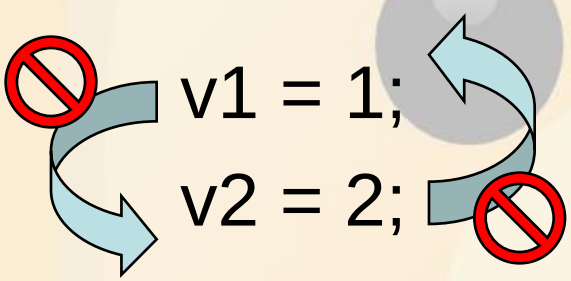
volatile int v;　　int x, y;

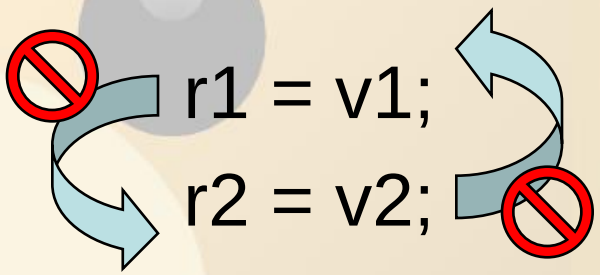
volatile変数をまたいだリオーダーが自由に起こる

volatileの問題点その2

一応、volatile変数同士では、コンパイラによるリオーダーはされないが…

```
volatile int v1, v2;
```





これでは、スレッド間で共有される全ての変数にvolatileを付けなければならなくなる!

volatileの問題点その3

- しかも、CPUによってもリオーダーされることがあるアーキテクチャでは、機械語レベルでも「メモリバリア命令」が必要となる。
  - ⇒ が、コンパイラがvolatile変数の読み書きに対してメモリバリア命令を発行してくれる保証も無い。

volatileに関する都市伝説

- 要するに、C/C++のvolatileはマルチスレッドのことを何も考えていない!

「volatileと宣言された変数はマルチスレッドで安全に使用できます。」

「複数のスレッドから読み書きされる変数にはvolatileを付けて最適化を抑制しましょう。」

⇒ NG!!

そしてatomicへ…

- 次期C++標準である C++0x (aka C++201x) では、アトミック変数やメモリバリアの仕様が登場する。
- volatileに関する定義はそのままで、アトミック型を別途定義している。

```
namespace std {
    template<class T> struct atomic;
}
```

C++0xでのメモリバリア

- C++0xのatomic型では、メモリバリアの有無を引数で指定する。

  void store(T, memory_order = memory_order_seq_cst);

  T load(memory_order = memory_order_seq_cst);

- デフォルト値は「メモリバリアあり」。
  - 「ゼロオーバーヘッド原則」のC++がコストのかかるメモリバリア付きをデフォルトとすることからも、アトミック変数とメモリバリアを一緒に扱うことの重要性がわかる。

Javaのvolatile

- 一方、Javaのvolatile変数は、load, storeのアトミック性やメモリバリア効果を備えている。
  - つまり、C++0xのatomic型に近い使い方ができる。
  - ただし、++ や += 演算子はアトミックではないので、これらのアトミック演算が必要であれば java.util.concurrent.atomic.Atomic* を使う。
- C/C++とJavaでは、volatileの意味が全く変わってしまっている。

まとめ

- スレッド間通信のprimitiveであるアトミック変数とメモリバリアについて。
- happens before関係の説明と、正しく同期化されたコードの作り方。
- C/C++のvolatileは、マルチスレッドでは役立たず。
- C++0xとJavaにおけるアトミック変数、メモリバリア。

ところで…

- 他の言語では、アトミック変数やメモリバリアに相当する機能はどう扱われているの?

⇒ 皆さんへの宿題としますので、わかったらLTなどで発表してみてください。(笑)