

Java Pathfinder

2009.06.29

田辺

資料

- 本日の講義資料 (スライド, ソース) は,
`http://cent.xii.jp/
tanabe.yoshinori/09/06/jpflect/`
から, ダウンロードできます.

Java PathFinder (JPF)

- モデル検査器
 - 対象: Java のバイトコード
 - マルチスレッドプログラムの検証
 - 独自のJava Virtual Machine
 - 状態を明示的に生成
 - 拡張可能
- オープンソース
 - sourceforge.net で公開
 - 当初は NASA Ames Research Center で W.Visserらが開発

本日の内容

- Java 同期機構の復習
- モデル検査
- Java PathFinder概要
- デッドロックの検出
- アサーションの検証

Java同期機構の復習

Threadクラス

- スレッドを作りたい時, Threadクラスのサブクラスを作成する.
- run メソッドを再定義
- start メソッドによって, runメソッドの内容を実行するスレッドが開始される.
- サンプルコード: MyThread1.java
(フォルダ javaRev)
 - 2つのスレッドth1とth2が並行して実行される.
 - 実行順序は, 毎回異なる.

synchronized文

- obj を任意のオブジェクトとして,
`synchronized(obj) { ... }`
という文が作れる. 同一のobjに対する複数のsynchronized文が, 複数のスレッドによって同時に実行されることはない.
 - サンプルコード MyThread2.java
- 異なるobjに対するsynchronized文は同時に実行されうる.
 - サンプルコード MyThread3.java
- `synchronized(obj)`ブロック
→ objに関する排他的ロック.

synchronizedメソッド

- メソッドにsynchronized修飾子をつけることができる.

```
synchronized method() { ... }
```

は,

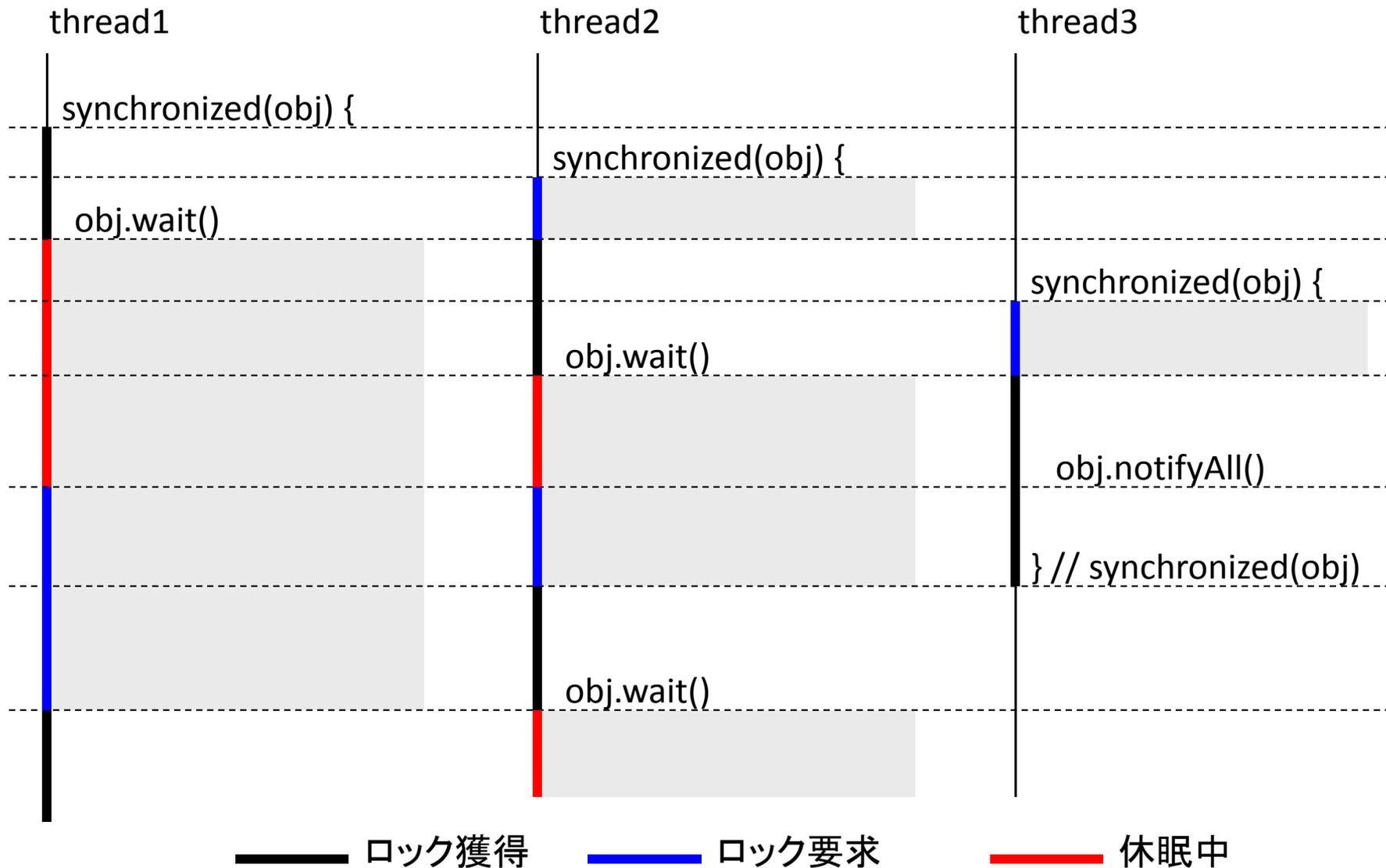
```
method() { synchronized(this) { ... } }
```

と同じ.

(参考) wait() と notifyAll()

- synchronized(obj) {...} ブロック内では, obj.wait() と obj.notifyAll() を呼ぶことができる.
- obj.wait() が呼ばれると, objに関するロックを放棄して, 休眠をはじめめる.
- obj.notifyAll()が呼ばれると, そのobjで待っているスレッドすべてが休眠から復帰する.
(しかし, ロックを取得するまでは実際には動作できない.)
- 注意: wait() は, InterruptedExceptionを投げる.

(参考) wait()とnotifyAll()



練習

- 配布ファイル中の, prodConsフォルダ
- Producer-Consumerモデル
- ソースファイル:
Main.java, Consumer.java, Producer.java,
Queue.java
- overflow, underflow エラーが発生する.

練習

1. Queue を次のように改良して, Queue2を作れ.
 - put: booleanを返すようにする. うまくputできたときにはtrueを, そうでないときには(例外を発生させずに>falseを返す.
 - get: getできなかつたときには, 例外を発生させずにnullを返す.
2. ConsumerX, ProducerX は, 改良としてあまり良くない. 理由を述べよ.
3. wait/notifyAll を用いて, Consumer, Producer を改良せよ.

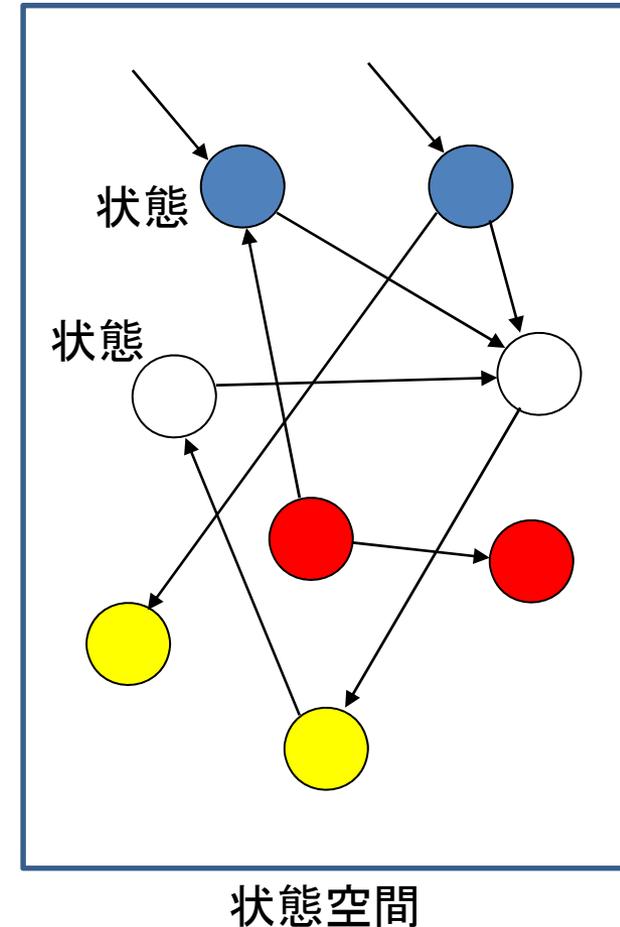
モデル検査

モデル検査

- 状態機械の状態空間を網羅的に探索することにより、与えられた性質が正しいかどうかを判定する検証方法
 - モデル検査
 - 完全な自動化が可能
 - 専門的な知識が必要ない
 - 定理証明の手法を応用した検証方法
 - 数理論理学の知識が必要
- 取り扱える状態数
 - 10^4 から 10^5 (初期)
 - 10^{120} (現在)
- 問題点: 探索空間のサイズ (state explosion problem)

状態空間の性質の検証

- 安全性: 「常に〜〜が成立する」
 - 例: 常に赤以外の色である.
- 活性: 「(〜〜がおこると) いつかは〜〜が起こる」
 - 例: 白色になったら, いつかは黄色になる.



現代的モデル検査

- 古典的モデル検査

- 対象: 仕様書, プロトコルなど
- 設計段階など, 「上流工程」でモデル検査を適用.
- 仕様書などから検査器にかけられるモデルを (ほとんどの場合) 手動で構築.

- 現代的モデル検査

- 対象: プログラムソースコードなど
- テスト段階など, 「下流工程」でモデル検査を適用. ほとんどデバッグの一種.
- ソースコードからモデルを自動的に抽出.

Javaプログラムのモデル検査

- 状態は, (おおざっぱにいて) 以下の要素からなる.
 - 各スレッドの実行位置
 - 変数の値
 - ヒープに存在するオブジェクトの状況

例題

- 例外が投げられることはあるか?

```
public class A extends Thread{

    private static int x = 0;
    private int id;

    public static void main(String[] args) {
        A a = new A(1);
        A b = new A(2);
        a.start(); b.start();
    }

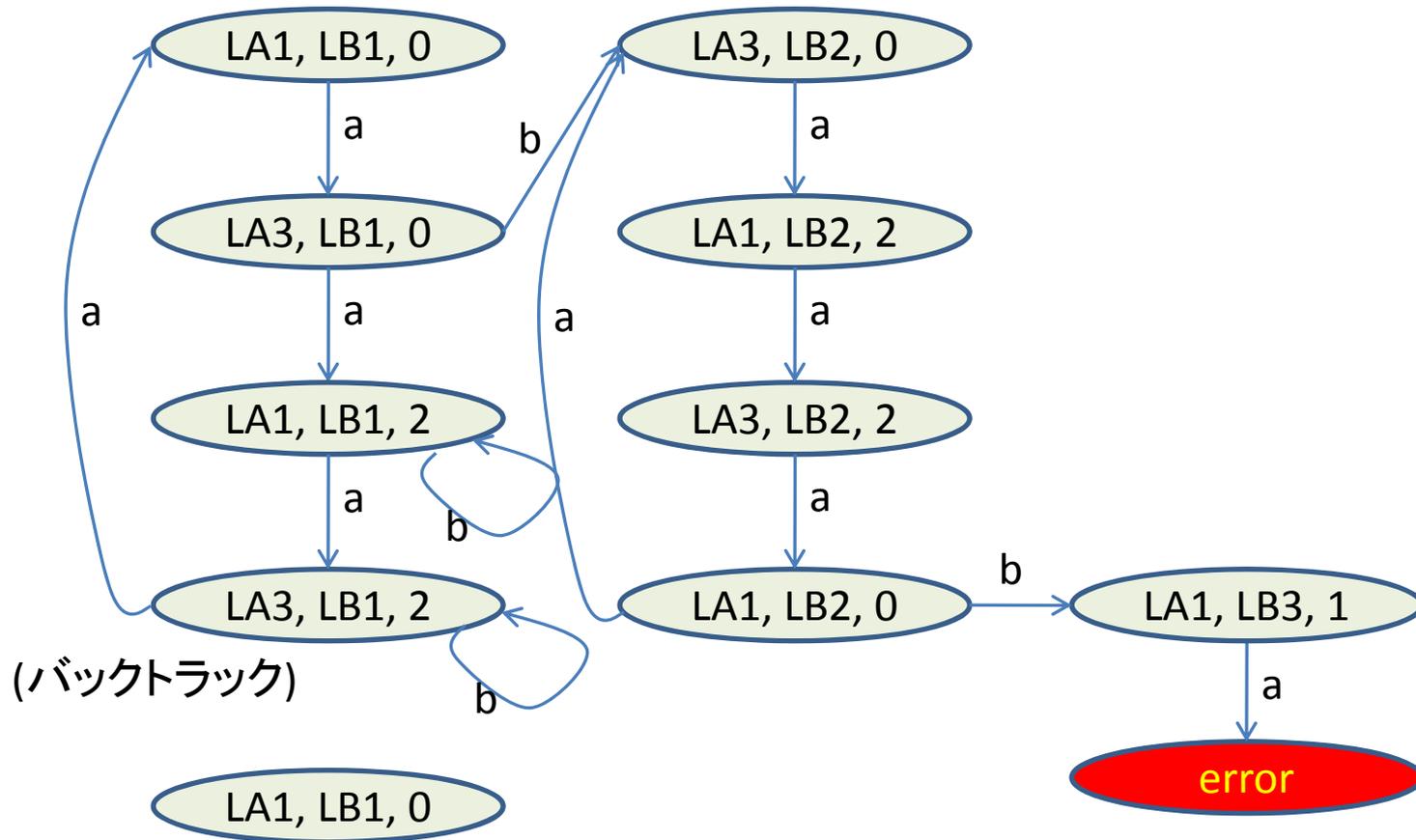
    public A(int id) { this.id = id; }

    public void run() {
        if (id == 1) { ma(); }
        else          { mb(); }
    }
}
```

```
public void ma() {
    while (true) {
        LA1: if (x == 1) {
            LA2: throw new
                RuntimeException();
        }else {
            LA3: x = 2 - x;
        }
    }
}

public void mb() {
    while (true) {
        LB1: if (x == 0) {
            LB2: x++;
            LB3: x++;
        }
    }
}
}
```

- 状態 ... (aの実行位置, bの実行位置, xの値)
(ほんとうは, もっと考慮すべきものあり)



深さ優先方式による状態空間の探索

例題 (改變)

```
public class A extends Thread{

    private static lock = new A(0);
    private static int x = 0;
    private int id;

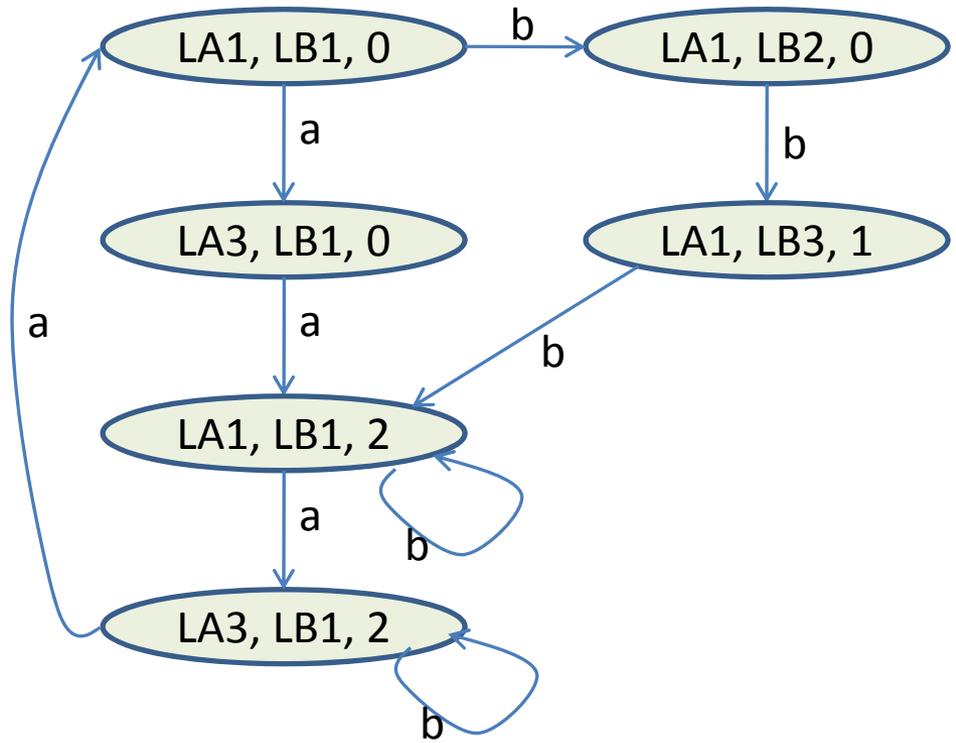
    public static void main(String[] args) {
        A a = new A(1);
        A b = new A(2);
        a.start(); b.start();
    }

    public A(int id) { this.id = id; }

    public void run() {
        if (id == 1) { ma(); }
        else          { mb(); }
    }
}
```

```
public void ma() {
    while (true) {
        LA1: synchronized(lock) {
            if (x == 1) {
                LA2: throw new
                    RuntimeException();
            }else {
                LA3: x = 2 - x;
            }
        }
    }
}

public void mb() {
    while (true) {
        LB1: synchronized(lock) {
            if (x == 0) {
                LB2: x++;
                LB3: x++;
            }
        }
    }
}
}
```



実際には、
 ロック取得状況も状態の中に入っている。

実際には...

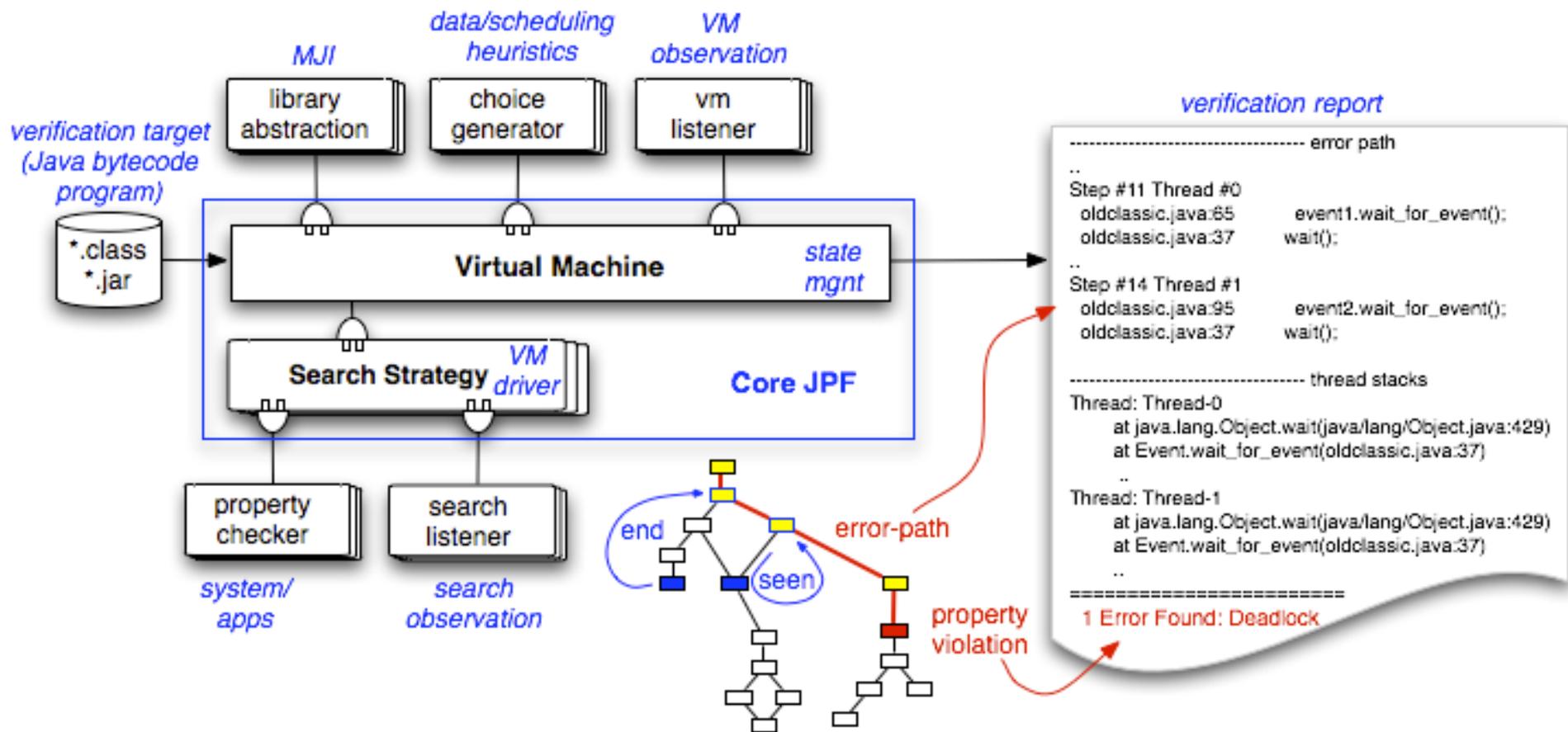
- Java ソースの各ステートメントは, atomic に実行されるわけではない. 一つのステートメントの実行中にも, 他のスレッドが割り込むことがある.
- 本当にすべてのあり得る状態を生成すると, すぐにメモリ不足になってしまうため, 様々な工夫が行われる.
- 安全性検証は, 全状態を調べることで行える. 活性検証には, もっと複雑なアルゴリズムが必要.

Java Pathfinder概要

Java PathFinder (JPF)

- モデル検査器
 - 対象: Java のバイトコード
 - マルチスレッドプログラムの検証
 - 独自のJava Virtual Machine
 - 状態を明示的に生成
 - 拡張可能
- オープンソース
 - sourceforge.net で公開
 - 当初は NASA Ames Research Center
 - W.Visserら

JPF概要図

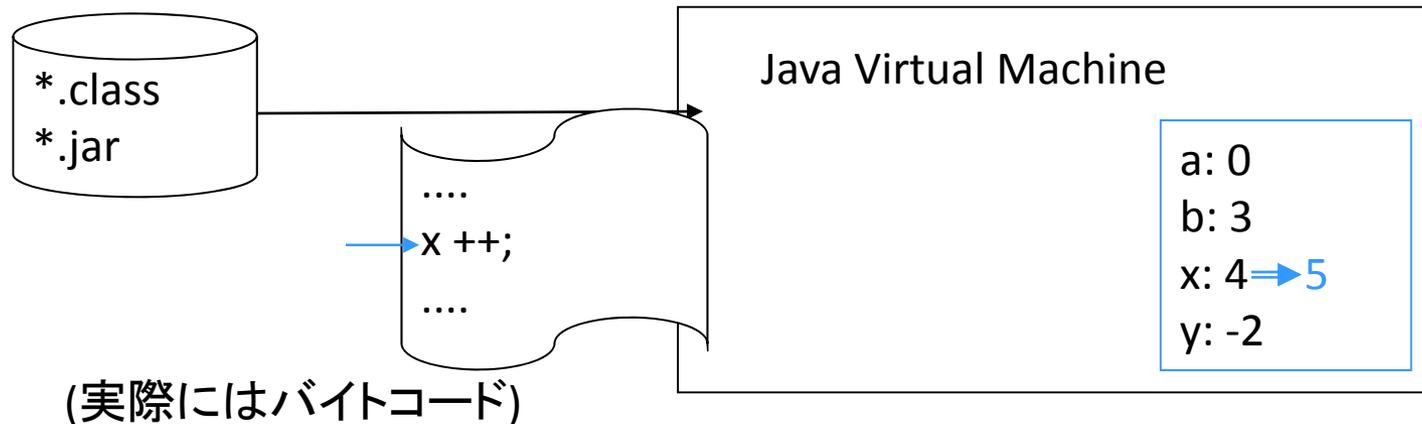


出典: <http://javapathfinder.sourceforge.net/>

JPFの状態空間

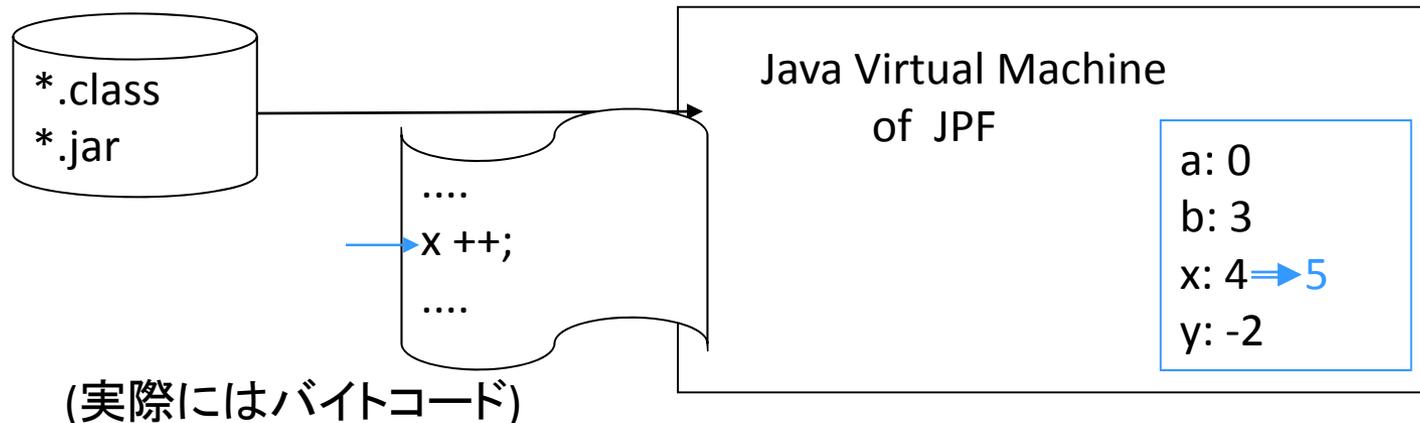
- 状態
 - 各スレッドの実行位置
 - 変数の値
 - ヒープのオブジェクト
- 状態遷移
 - どれか1つのスレッドが、
現在のバイトコードを実行する.
 - 遷移の非決定性:
(主に) スレッドスケジューリングによる

JPFの動作原理(1)



- Javaのクラスファイルは, Java Virtual Machine (JVM) によって解釈され, 実行される.

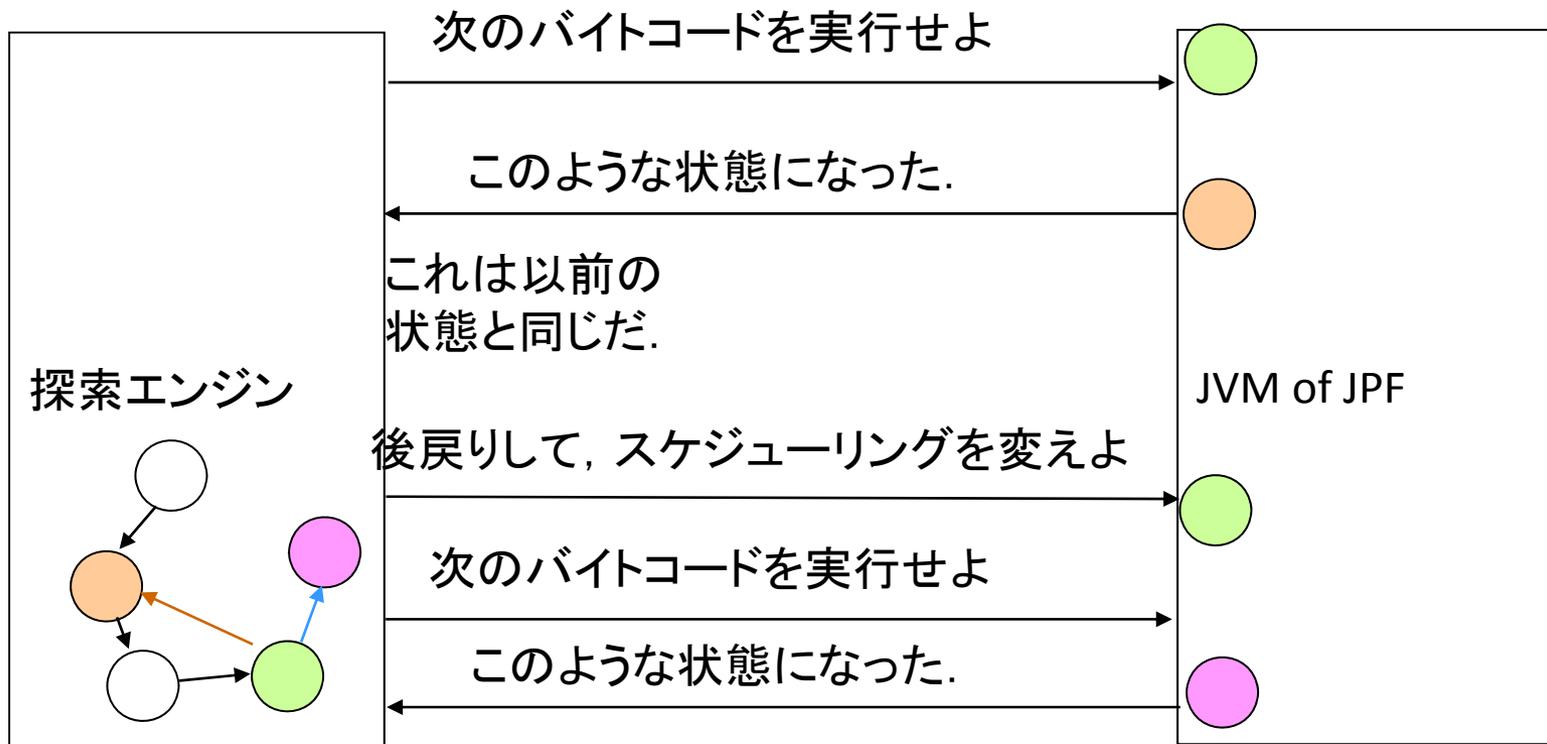
JPFの動作原理(2)



- JPF独自のJVM

- 通常のJVMと同様に, クラスファイルを解釈
- バックトラック可能
- スレッドスケジュールの制御が可能

JPFの動作原理(3)



- 探索エンジン
 - JPFの中心
 - JVM of JPF を駆動する.

効率化

- 半順序簡約
 - スレッド相互間に干渉のない部分は一気に実行
- 状態の表現
 - ハッシュを利用して, コンパクトな表現を実現

検出できる不具合

- デフォルト
 - デッドロック
 - アサーション違反
 - 補足されない例外
- ユーザ定義
 - 基本的には, 安全性性質

(JPF実行例)

JPFのバージョン

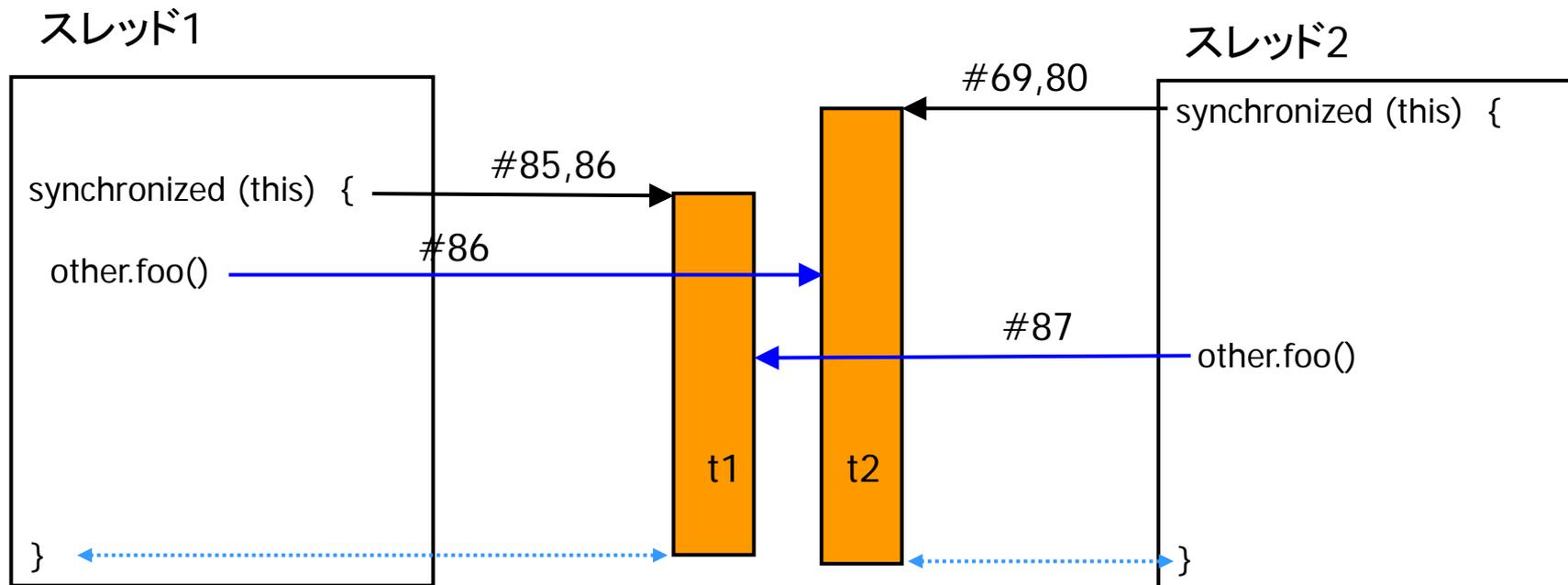
- 入手可能なバージョン: v3 と v4
 - v3: 安定版. ソースで配布. 配布サイトではv1.0aと書かれている.
J2SE1.4 に対応. (Java5,Java6では動作しない.)
 - v4: 開発版. ソースで配布. 頻繁に更新. v3と比べ, 基本的な構成は同じ. 使い勝手が向上している.
Java5,Java6に対応. (J2SE1.4では動作しない.)
- 本日の講義では, v3を用いる.
- インストールと実行方法は, 手順書を参照.

デッドロックの検出

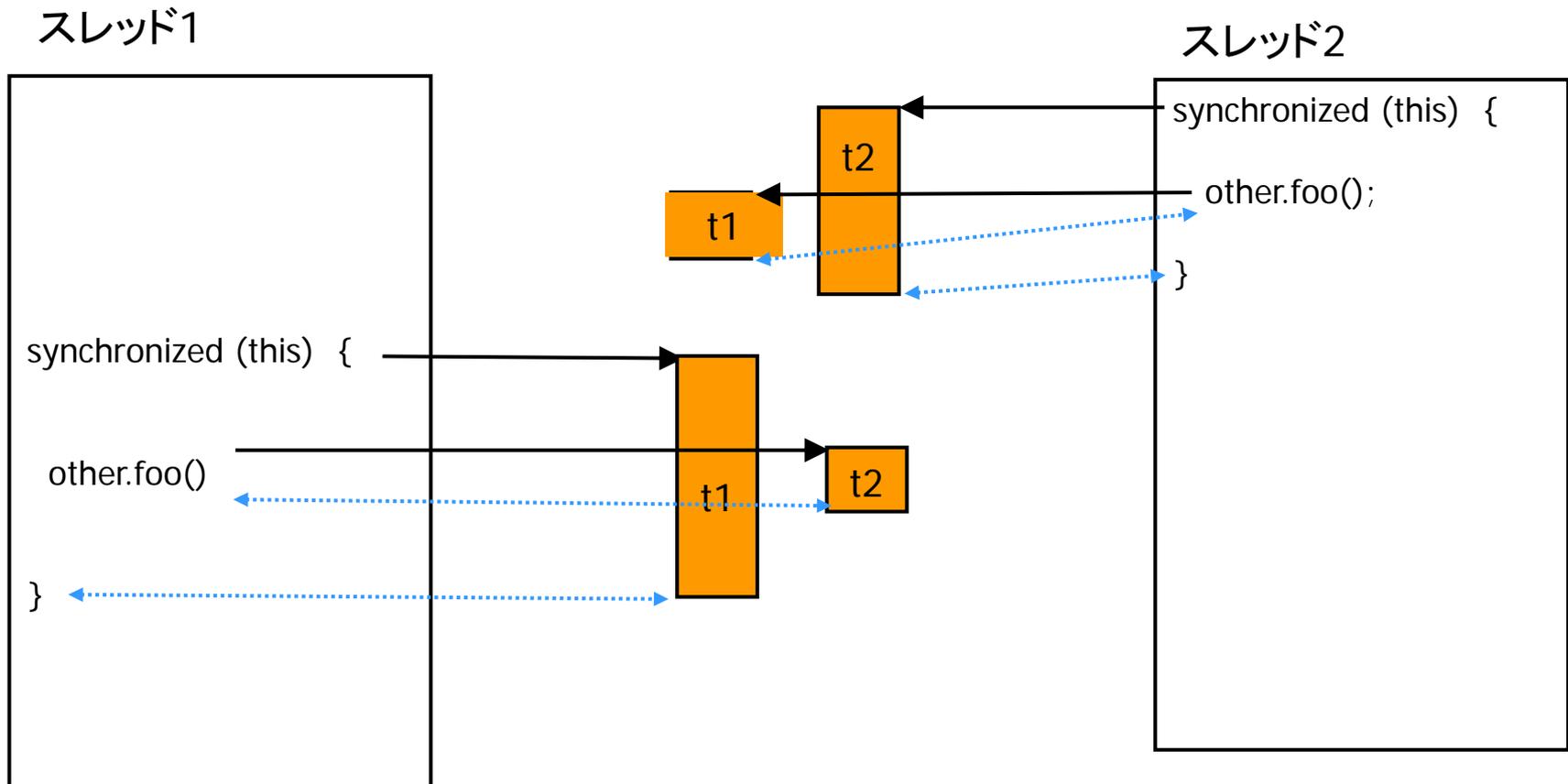
デッドロックの検出

- JPFは、デッドロックを自動的に検出する.
- ソースファイル `deadlock/Deadlock.java`参照

デッドロックとなる理由



(比較) デッドロックが起こらない場合



練習: デッドロック

ファイル misc/Friendly.java

1. javac で Friendly.java コンパイルし, 作成されたクラスを java で実行せよ.
2. Friendly.class を JPF で実行し, デッドロックが起こりうることを確認せよ.
3. デッドロックが起こりうる理由を説明せよ.
4. デッドロックを起こさなくするようにプログラムを改変せよ. ただし, hug() および hugback() は, 他のスレッドに割り込まれずに実行されるように保つこと. JPFを用いて, デッドロックがなくなったことを確認せよ.

注意: 良く出る警告

- 以下の警告が現れることがある.

```
sync-detection assumed to be protected by: ....  
found to be protected by: {.....}  
>>> re-run without '-sync-detection' <<<
```

- 警告の意味: jpf の最適化ルーチンのうちの 하나가 仮定しているプログラムの性質が満たされていない.
→ 結果が正しくないかもしれない.
- 対処: 以下の内容のファイルをカレントディレクトリに作り, jpfのコマンドラインスイッチ `-c` でファイル名を指定する.

```
vm.por.sync_detection = false
```

アサーションの検証

Javaのアサーション

- 形式: `assert p : m`
 - `p`: チェック対象のboolean型の式
 - `m`: エラーメッセージ
 - 「`: m`」の部分は省略可能.
- 指定したboolean式の値がtrueになるべきであるという表明.
- 実行時に式の値がfalseになると, 実行時エラー(`java.lang.AssertionError`)になる.
- コンパイルオプション: `javac -source 1.4`
- 実行時オプション: `java -ea`

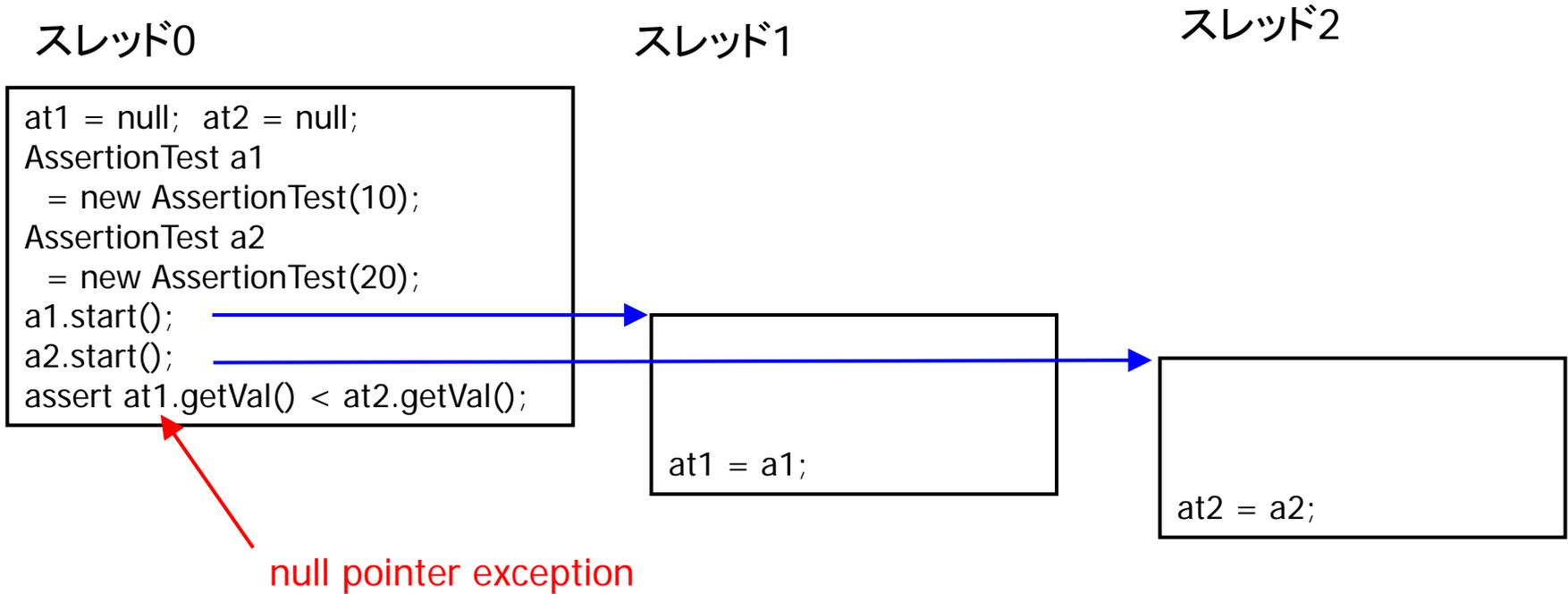
JPFとアサーション

- 通常の実行時には、アサーションの位置に制御が渡ったときに、チェックが行われる。
→ その実行によって通らないパスでは、チェックが行われない。
- JPFによる実行時には、すべてのパスでチェックが行われる。

AssertionTest1

- ソース参照 (AssertionTest1.java)
(フォルダ misc)
- 2つのスレッド a1 と a2
- a1 が先に start. a2が後から start
- run の内容: 自分自身を「登録する」
- 登録の内容: at1 が空いていたら at1 に, そうでなかったら at2 に代入.
- 両方登録が終わったとき, $a1 == at1$ になるか?

AssertionTest1



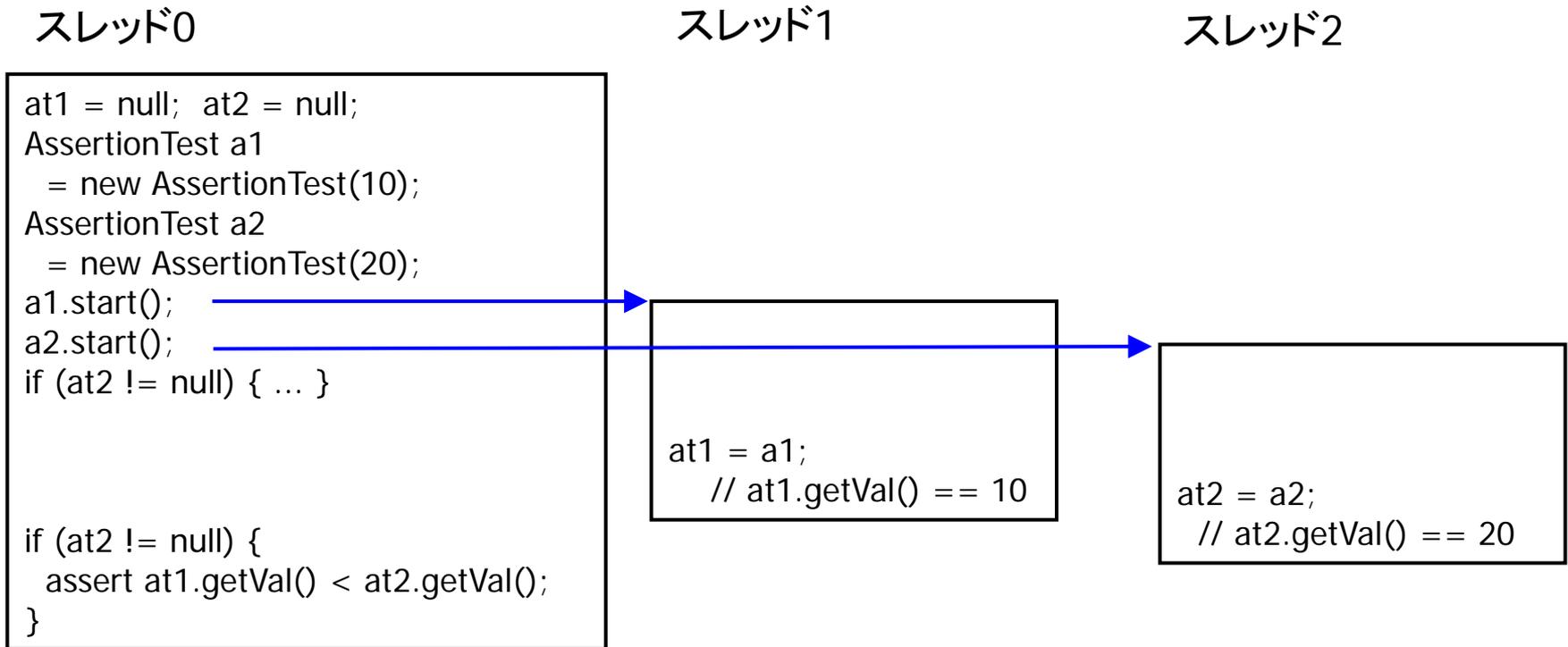
テストで検出できる.

JPFも, エラーを報告する.

エラートレース

- (エラートレース参照)

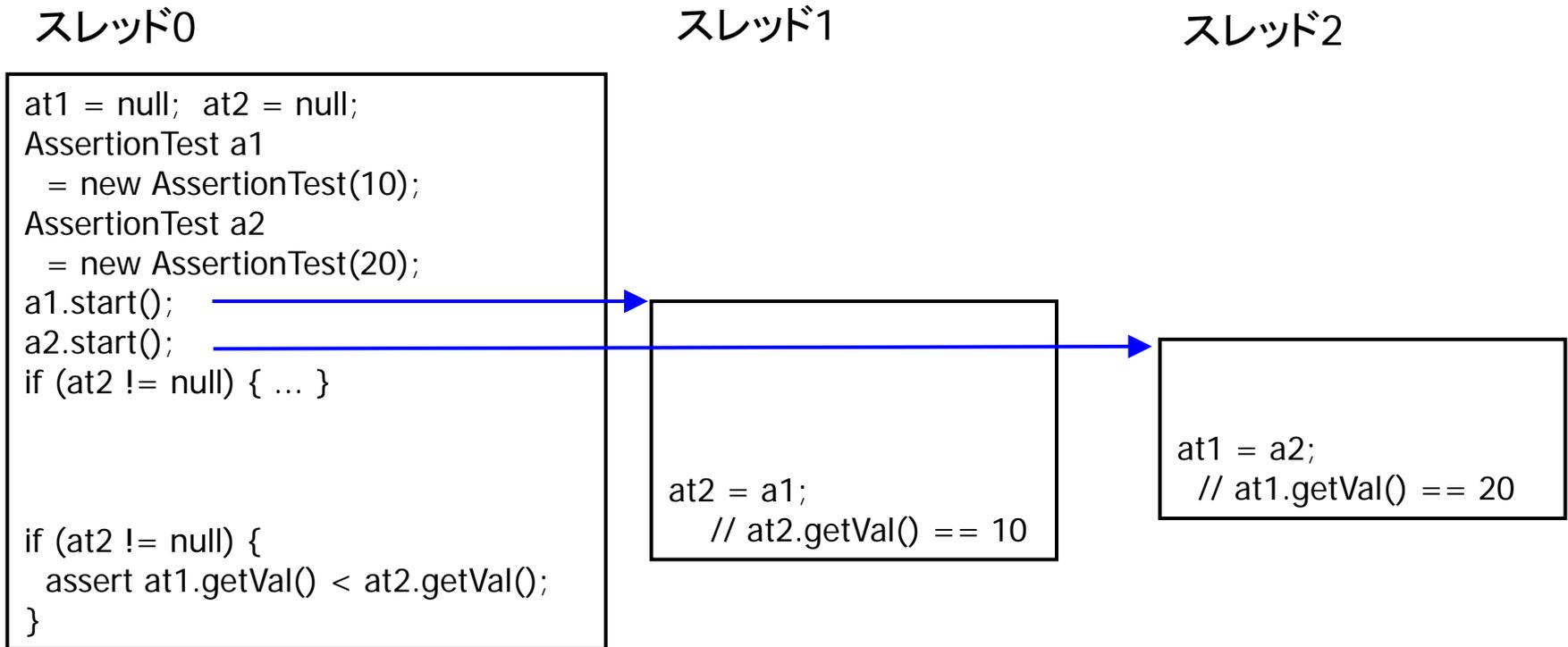
AssertionTest2 (1)



assertion は成功

1CPUマシンでは, こうなることが多い(?)
しかし, 状況によっては.....

AssertionTest2 (1)



assertion は失敗

テストでは検出できるとは限らない.

JPFでは, 検出できる.

エラートレース

- (エラートレース参照)

練習

- misc/AllNum.java は, スレッドスケジューリングにより, 計算結果がさまざまになり得ることを示すソースである.
- $n = 3$ を jpf で実行せよ. エラートレースを読んで, $x = 3$ となる計算過程を示せ.
- $x = 11$ となる計算過程はないことを確かめよ.
- $x = 10$ となる計算過程を示せ.

練習

1. misc/Saving.java をコンパイルし, (通常のJVMで)実行せよ. アサーション失敗が報告されるか.
2. JPFで検証せよ. アサーション失敗が報告されるか. アサーションを

```
saving <= th1.amount + th2.amount
```

に変更するとどうか.
3. アサーションが失敗する理由を説明せよ. (わかりにくい場合にはSaving2.javaを使用してみよ.)
4. アサーションが常に成功するようにプログラムを改変せよ. また, JPFの実行により常に成功することを確認せよ.

レポート課題

提出期限: 2009年7月31日(金)

提出先, 形式などは, 後ほど確認して,

`http://cent.xii.jp/`

`tanabe.yoshinori/09/06/jpflect/`

に記述します.

レポート課題

- フォルダ jpf/rwlock
- クラス MyObj0, MyObj1
 - メソッド refer() ... 参照
 - メソッド update() ... 更新
- クラス App1 (main)
 - 2つのスレッド th1, th2 を起動する.
 - th1: refer(), refer(), refer(), update() の順に呼ぶ.
 - th2: refer(), update(), refer(), refer() の順に呼ぶ.
- クラス Time
 - 実行時間を制御する. 経過時間を表示する.
 - Time.enabled=falseとすると, 無効になる.

レポート課題

- 1: App1+MyObj0, App1+MyObj1について, Time.enabled = true にして, 表示される実行時間を記録せよ. ただし, App1+MyObj? とは, App1.java の, 8-10行目の該当する行を選んだもののこととする.

以下, JPF を用いるときには Time.enabled = false にした方が
良い.

- 2: App1+MyObj0 について, update の実行中に, 他のスレッドでupdateまたはreferが実行されることがあることを確認せよ.
- 3: App1+MyObj1について, updateやrefer の実行中には, 他のスレッドでupdateもreferも実行されないことを, JPF を用いて確認せよ.

レポート課題

- 4: MyObj2.java を適切に作成して, App1+MyObj2 について, 以下の条件a,bが満たされるようにせよ.
 - a. update の実行中には, 他のスレッドでupdateもreferも実行されない.
 - b. 2つのスレッドで同時にreferが実行されることはありうる.

これらの条件が満たされることを, JPFを用いて確認せよ.

App1+MyObj2の実行時間が, App1+MyObj1よりは相当に短くなることを確認せよ.

(Time.enabled=trueとして実験せよ.)

- 練習問題, レポート課題などで質問のある方は,
y-tanabe_atmark_ci_dot_i_dot_
u-tokyo_dot_ac_dot_jp
あてにどうぞ.