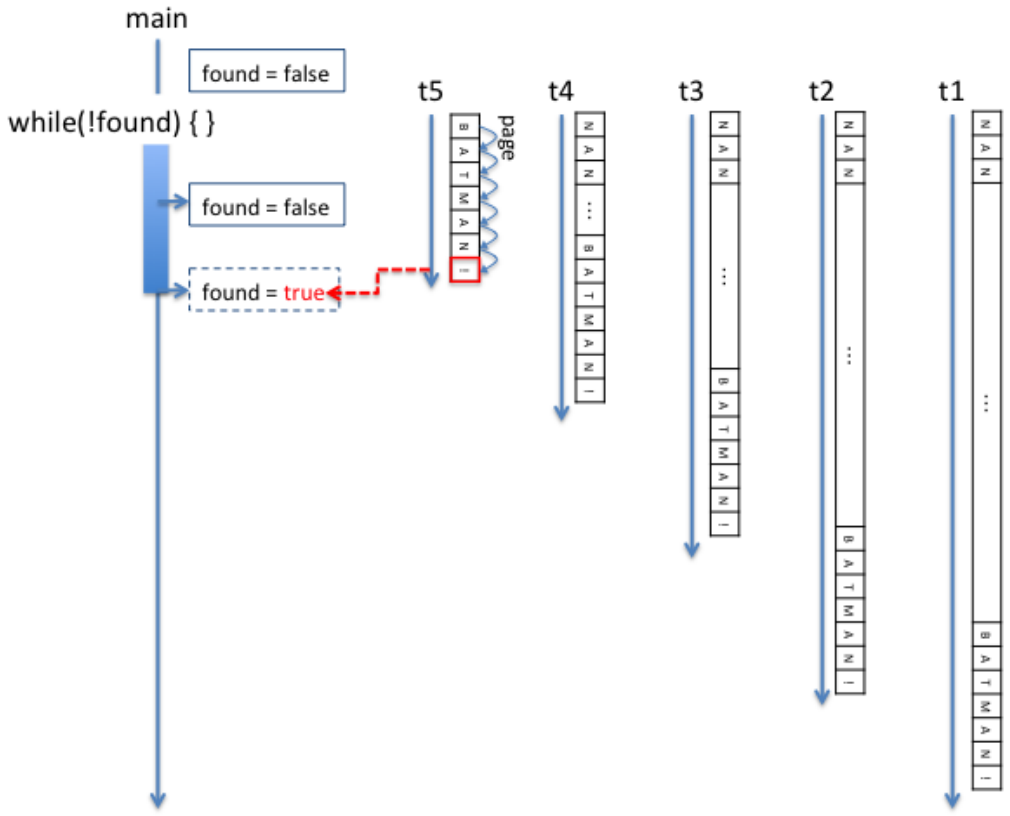
Chapter2. Part3

Volatile variables

- JVMには揮発性変数（**Volatile variables**）というものがある。
- Syncronizedに比べると軽い同期化の仕組み
- ステータスフラグに利用される：計算の完了・キャンセルを表すフラグ等
- 二つの利点
  - 変数への書き込み、読み出し操作の順序替え(**reorder**)がおこらない
  - 書き込みが全てのスレッドで即時に可視化される

- volatile example



```
class Page(val txt: String, var position: Int)
object Volatile extends App {
val pages = for (i <- 1 to 5) yield
  new Page("Na" * (100 - 20 * i) + " Batman!", -1)
@volatile var found = false
for (p <- pages) yield thread {
  var i = 0
  while (i < p.txt.length() && !found)
    if (p.txt(i) == '!') {
      p.position = i
      found = true
    } else i += 1
}
while(!found){}
log(s"results: ${pages.map(_.position)}")
}
```

  - あるスレッドtがfoundにtrueをセット、その後にメインスレッドがfoundを読むと、スレッドtがfoundにセットする前に可視だった値はメインスレッドにとっても可視となる　→　positionも可視

- ThreadSharedStateAccessReordering Exampleは全ての変数をvolatileにすることで修正できる
- Javaと違い、Scalaではローカル変数にvolatileを宣言できる
  - closureやネストされたクラスで利用されるvolatile変数毎に、volatile変数を保持したオブジェクトが作成される
  - 「オブジェクトへ”lift”された」

- volatileのリードは安上がりではあるが、ほとんどの場合Syncronizedに頼るべき
  - volatile変数のリード・ライトの複合操作（count++とか）は同期化無しではアトミックとならない
  - getUniqueIdの例はvolatileだけでは正しく実装できない

```
var uidCount = 0L
def getUniqueId() = this.synchronized {
  val freshUid = uidCount + 1
  uidCount = freshUid
  freshUid
}
```

The Java Memory Model

- 明示してこなかったがこの章を通して大部分JMMについて示してきた
- 言語メモリモデルとは変数への書き込みが他のスレッドでどのように可視となるかを記述した仕様
- メモリモデルを考える際に、逐次的一貫性(**sequential consistency**)と呼ばれるメモリモデルを想定するかもしれない
    - プロセッサーが変数vに書き込むと関連するメモリ領域が即時に変更され、他のプロセッサーがvの値を瞬時に見ることができる

- しかしThreadSharedStateAccessReordering Example で既にみたように、実際のモデルは逐次的一貫性モデルとほとんど関連しない
    - プロセッサはキャッシュ階層を持ち、メインメモリにいずれ書き込まれることしか保障しない
    - コンパイラはレジスタを利用してメモリ書き込みを避けることが許容され、（シングルスレッド内で）直列的なセマンティクスを維持する限り、最適化のため順序替え(reorder)を行う

- メモリモデルは並行プログラムの予測可能な振る舞いとコンパイラの最適化能力のトレードオフである
- すべての言語でメモリモデルは必須でなく、変数の書き換えが許されていない純粋な関数型言語ではメモリモデルは必要ない

- プロセッサアーキテクチャの違いはメモリモデルの違いをもたらす
- SyncronizedやVolatileなどの簡潔なSemantics無しでは全てのコンピュータで同じように動作するScalaプログラムを書くのは難しい
- ScalaはJVMのメモリモデル(JMM)を継承している
- JMMはプログラムの異なるアクション間での事前発生(**happens-before**)の関係を定義している
    - 「もしアクションAがアクションBの事前発生であれば、アクションBはAのメモリ書き込みが可視である」
    - プログラムが動作するマシンに関わらず、同じ事前発生の関係は同じプログラムに対して有効である

- JMMのアクション
    - (Volatile)変数のリード/ライト
    - モニタのロック/アンロック
    - スレッドのスタート/ジョイン

- 事前発生のルール
    - Program order：スレッド内の各アクションは、そのスレッド内のプログラム順序で後からくるそれ以外のアクションよりも事前発生する
    - Monitor locking：モニタのアンロックは、その後の同じモニタのロックより

も事前発生する
    - Volatile fields：揮発性フィールドへの書き込みは、同じフィールドのその後の読み出しよりも事前発生する
    - Thread start：スレッドのstart()呼出しは、開始したスレッドのすべてのアクションよりも事前発生する
    - Thread termination：スレッド内のどのアクションも、他のスレッドによるそのスレッドの終了の検出よりも事前発生する
    - Transitivity：AがBよりも事前発生し、BがCよりも事前発生するなら、AはCよりも事前発生する

- 事前発生の関係はスレッド間でのメモリ書き込みの可視性を保障するために存在する. プログラムの命令文の時間的順序を確立するために存在するわけではない
    - 書き込みAが読み出しBの事前発生と言った場合、BがAの書き込みを見えることが保障される
    - AがBより早く発生するかどうかはプログラムの実行に依存

- JMMはvolatileの読み・書きとモニタのロック・アンロックがreorderされないことを保障
- この関係は次のことを保障
    - プログラム順序でvolatileリードの前のnon-volatileリード(or モニタロック)はreorderされない
    - プログラム順序でvolatileライトの前のnon-volatileライト(or モニタアンロック)はreorderされない

- 事前発生のルールによりさらに高次のルールが導きだされる
    - interuptの呼出しは、interuptされたスレッドによる検出よりも事前発生する
    - 一般的な実装では、interuptの呼出しはモニタを利用しているため

- ScalaのConcurrency APIも事前発生のルールに基づいている
- 全ての書き込みが全ての読込みの事前発生となることを確実に行うことがプログラマーのタスク
    - これが満たされない場合は、データ競合(**data race**)があると言える

Immutable objects and final fields

- データ競合を避けるためには事前発生の確立が必要と説明してきたが例外がある
- finalフィールドのみのオブジェクトで、コンストラクタの完了前に別のスレッドから参照されない場合、そのオブジェクトはimmutableとみなされ同期化の必要がない

- Scalaではフィールドのfinalは、サブクラスでgetterメソッドをoverrideできないことを表す
  - valで宣言するとフィールド自体は常にfinalになる
  - Example

```
//class Foo(final val a: Int, val b: Int)の変換イメージ
public class Foo {
  final private int a$;
  final private int b$;
  final public int a() {return a$;}
  public int b() {return b$;}
  public Foo(int a, int b) {
    a$ = a;
    b$ = b;
  }
}
```

- Scalaは関数型とオブジェクト指向のハイブリッドであるので、言語の特徴の多くはimmutable objectsにマップする
- ラムダはenclosing classの参照やlifted variables をキャプチャすることができる
  - Example

```
object LambdaExample extends App {
  var inc: () => Unit = null
  val t = thread {
    if (inc != null) inc()
  }
  private var number = 1
  inc = () => {number += 1}
}
```

  - ローカル変数numberはラムダによってキャプチャされるためliftする必要がある
  - 無名クラスfunction()で書き直した場合

```
number = new IntRef(1)
inc = new Function() {
  val $number = number
  def apply() = $number.elem += 1
}
```

- incのアサインと、スレッドtによるreadに事前発生の関係はない

- しかしながら、スレッドtでincがnullではない場合、incの実行は正しく動作する
  - $numberがimmutableのラムダオブジェクト項目として適切に初期化されるため
  - Scalaコンパイラはラムダの値が、finalで適切に初期化されていることを保障する

- 現行バージョンのScalaにおいては、ListやVectorなどのCollectionは同期化無しで共有してはいけない
  - Listの中身はfinalではないため

- たとえオブジェクトがimmutableに見えても、スレッド間で共有する場合は常に適切に同期化する