Javaのスレッドについて

2002年4月14日

2003年4月12日改訂

海谷 治彦

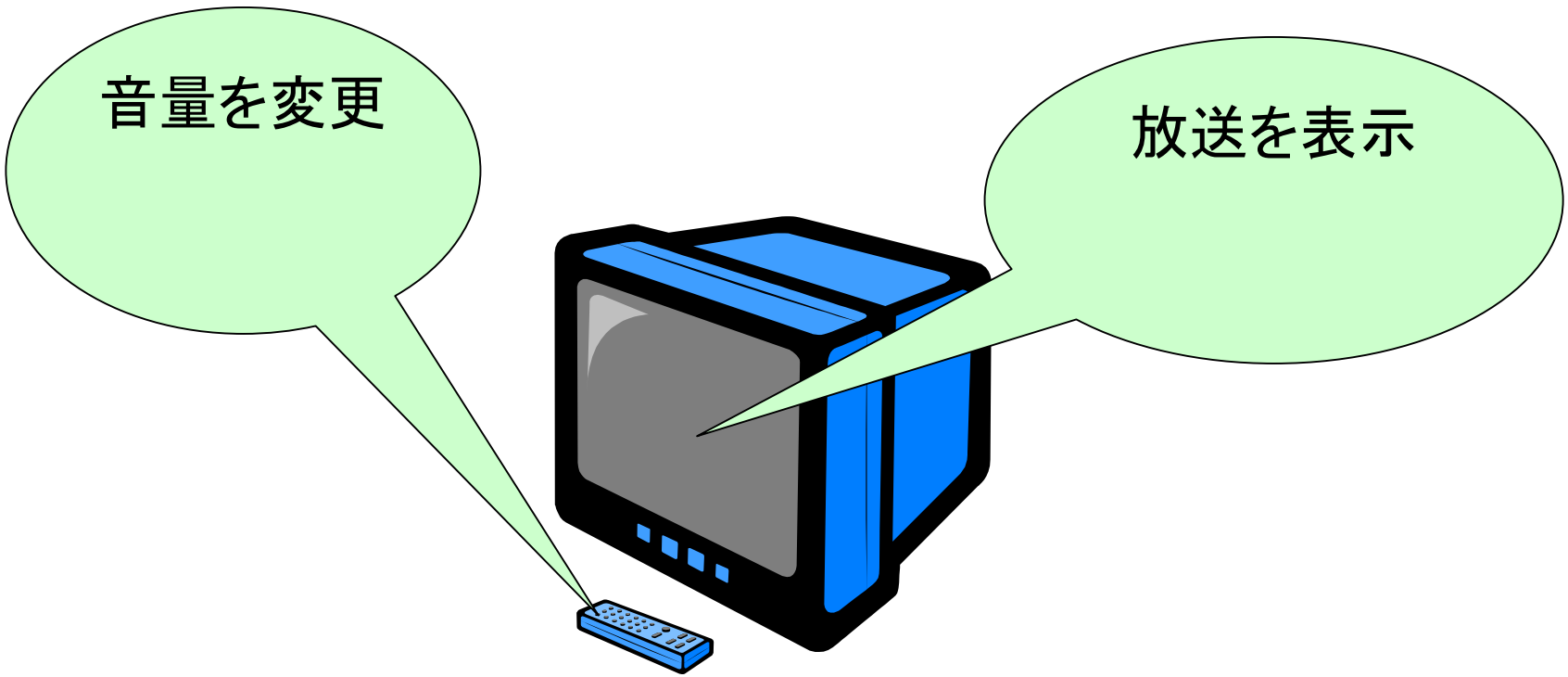
何故にJVMにてスレッドを？

- ソフトウェア実験2後半でやりそりれたから！
- JVMレベルでスレッドがサポートされているため.
- ハードウェア機器等は本質的に並行動作するものであるため.

並列動作

- 複数の処理が同時に動くこと.
- 自然界？ではあたりまえ.
- ハードウェアを伴う機器制御でもあたりまえ.

例: テレビの制御



音量変更中に，放映(画，音)が停止したら，やっぱ怒るよねぇ．

並列と並行 (用語の話)

- **並列 (parallel)**
  - 複数の仕事を同時に行う.
- **並行 (concurrent)**
  - 潜在的に同時進行可能な処理を論理的に表現したプログラムの性質.

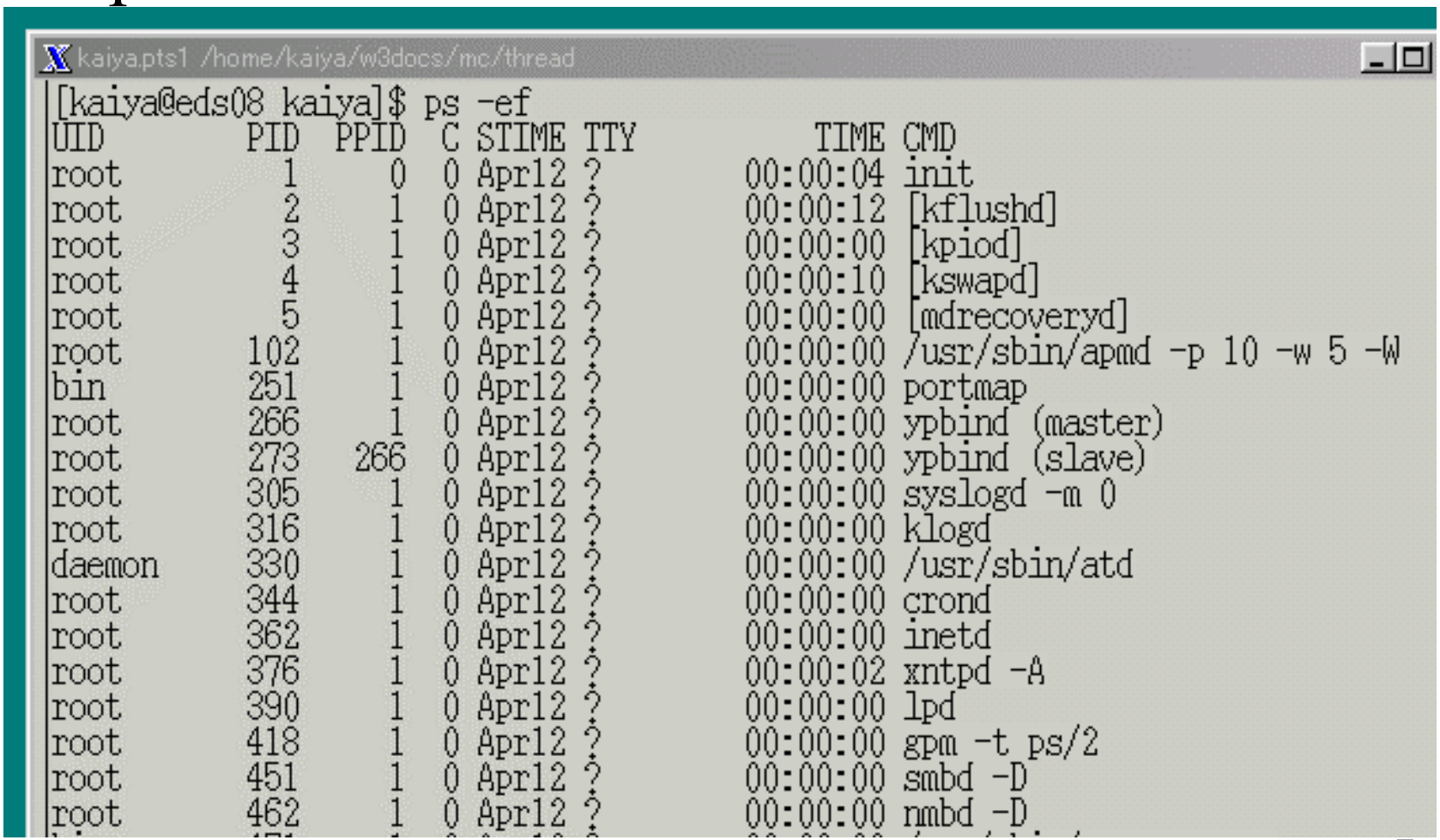
- よって, Javaでは並行処理を記述できるが, それが並列に実行されるかはマシン次第.

プロセス VS スレッド

- プロセス (もしくはタスク)
  - OSレベルで，独立した処理単位
  - ファイル等は共有できるが変数等は共有できない.
  - 詳細はOSの授業にて.
- スレッド
  - 1つのプログラム内での異なる処理の流れ.
  - 1つのプログラム内なので，当然，変数等も共有できる.

複数動くプロセスの観察(linux)

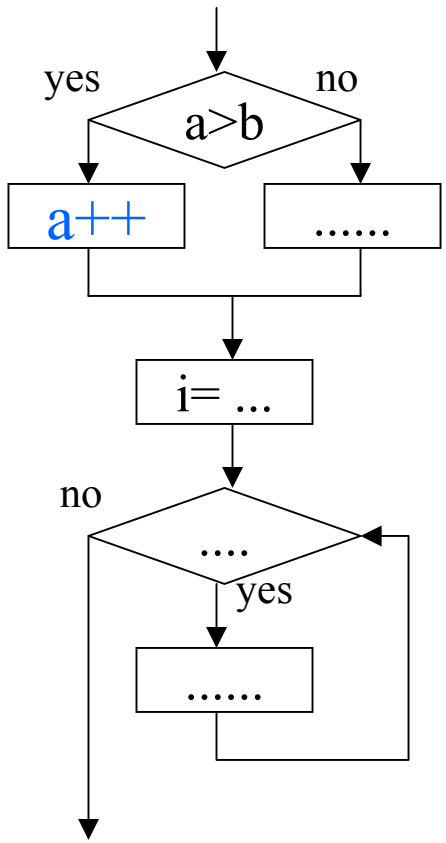
- psコマンドでプロセスを表示できる.



いままでやってきたプログラム

```
main(){
  int a, b, i;
  .....
  if(a>b){
      a++;
  }else{
      ........
  }
  for(i=... ){
      .....
  }
}
```

所謂
フローチャート



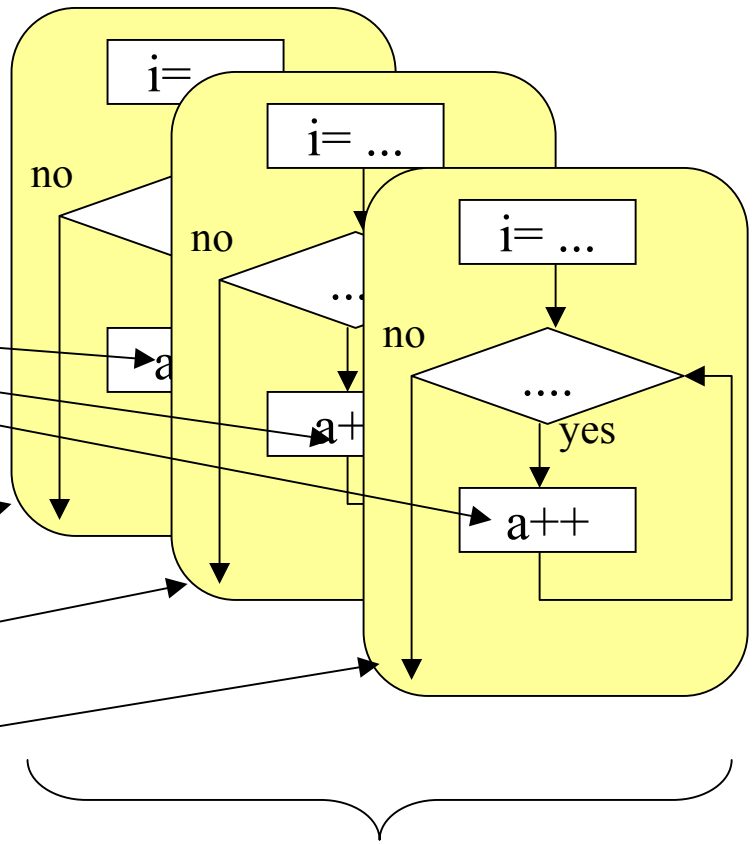
いままで学んできたプログラムは，ある一瞬には，ある1つの命令しか実行されていなかった.

処理の流れは一本である.

スレッドを利用したプログラム

```
class MyThread extends Thread{
static int a=0;
... run(){
    for(int i....){
        a++;
    }
}

.... main(....){
   .....
   new MyThread().start();
   new MyThread().start();
   new MyThread().start();
....
}
}
```

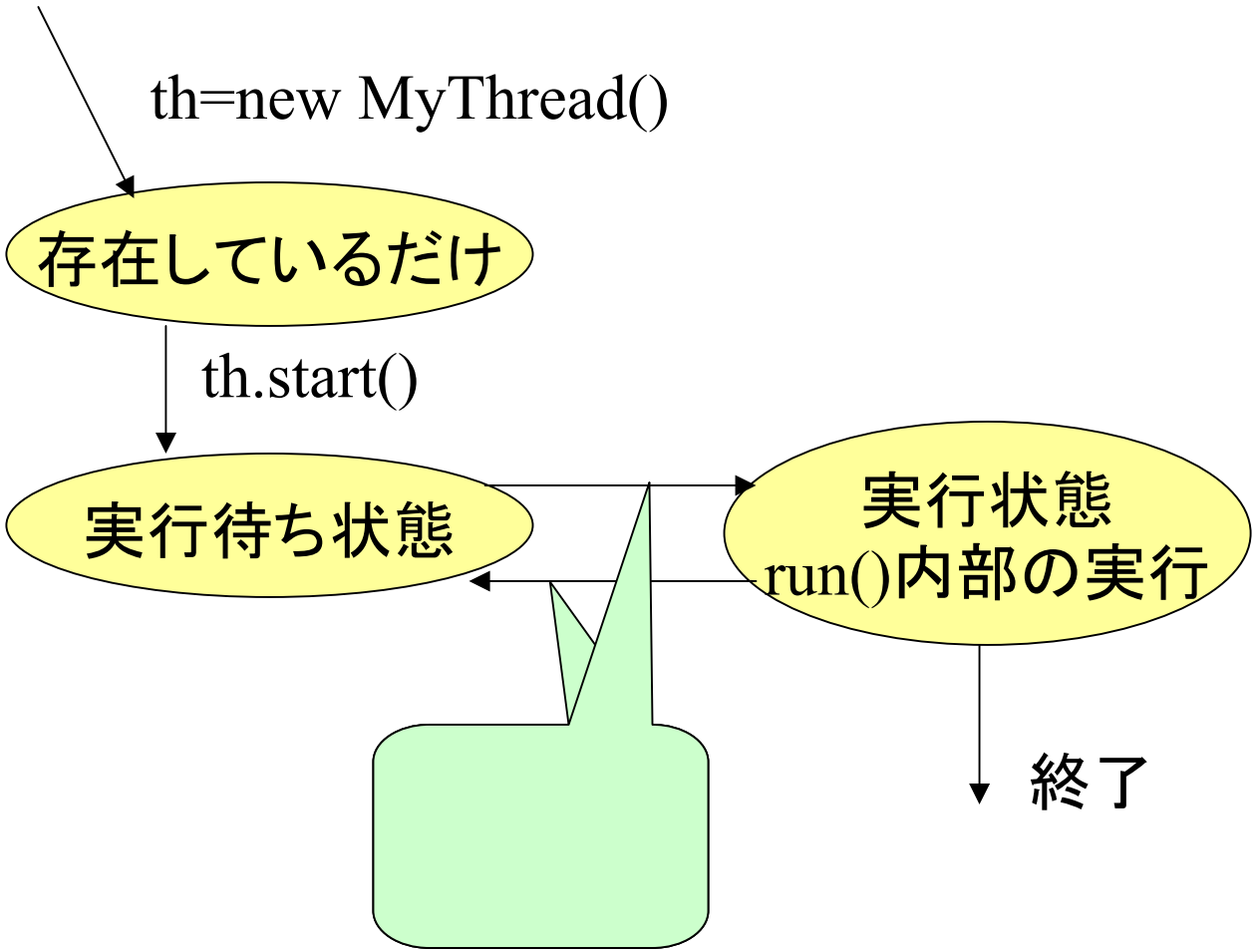


独立並行して動く.
変数も共有できる.

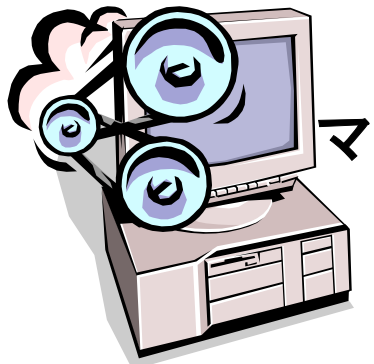
典型的なThreadの作り方

- Threadというクラスのサブクラスを作る.
- public void run() メソッドを実装する.
  - この中が実際並行に動作する処理となる.
- Threadサブクラスのインスタンスを作り，そのインスタンスにstart()メソッドを適用する.
  - run()メソッドが，start()メソッドからOSに指示に従い，呼び出される.
    - run()メソッドは通常，ユーザープログラムでは呼ばない.

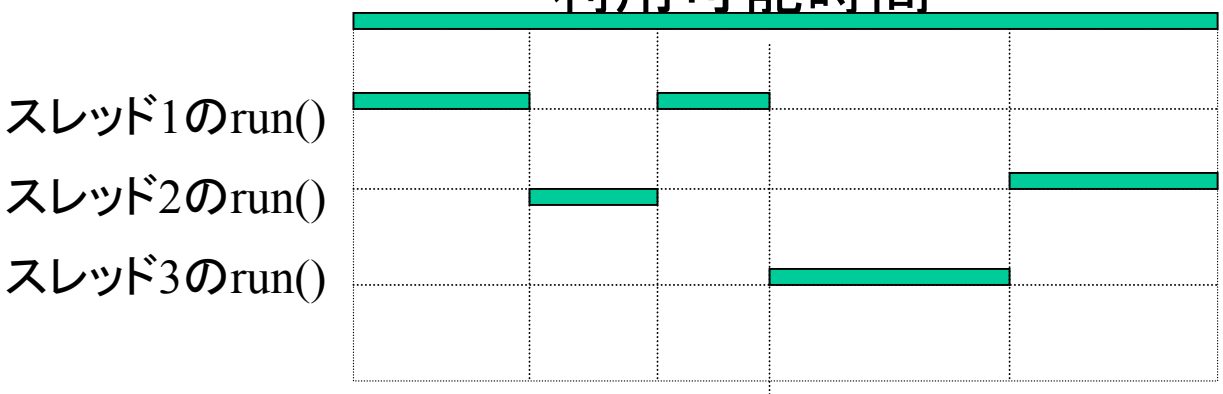
スレッドのライフサイクル (状態遷移図)



スレッドのスケジュール (1CPUの場合)



利用可能時間



例: IDの並行表示

```
public class NumOut extends Thread{
private int id;
    NumOut(int id){ this.id=id; }
    public void run(){
        for(int i=0; i<10; i++){
            System.out.print(id+" ");
        }
    }
    public static void main(String[] args){
        new NumOut(1).start();
        new NumOut(2).start();
        new NumOut(3).start();
    }
}
```

実際のところ，全然ばらばらに表示されない．

スケジュールに対して，run()の内容があまりに短いから．

個々の処理に明示的に「休止」処理を入れれば並行っぽく見える．
(Webページの方参照)

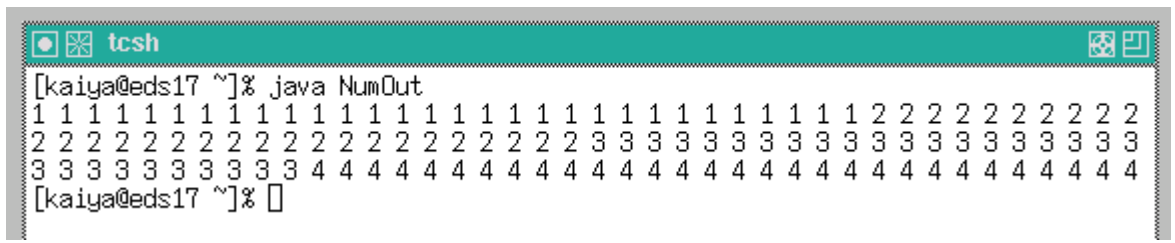
並行処理が並列っぽく見えない？

```
public class NumOut extends Thread{
private int id;
private static int max=30;
private static int tnum=4;

  NumOut(int id){ this.id=id; }

  public void run(){
    for(int i=0; i<max; i++){
      System.out.print(id+" ");
    }
  }

  public static void main(String[] args){
    Thread.currentThread().setPriority(Thread.MAX_PRIORITY);
    for(int i=0; i<tnum; i++)
      new NumOut(i+1).start();
  }
}
```



理由

- 1CPUのマシンでやってるので当たり前.
- それにしても，もちっとバラバラになってほしい.
  - それぞれのスレッドの処理が小さいので，スタートした順に処理が終わってしまう(涙)

明示的な実行権の委譲 yeild

- スレッドには，public static void yeild() が定義されている.
- このメソッドを呼ぶと，実行可能な他のスレッドに明示的に実行権を譲ることができる.
- なんとなく，linux2.2ではうまくyieldが動作しない？ (裏はとってませんが)

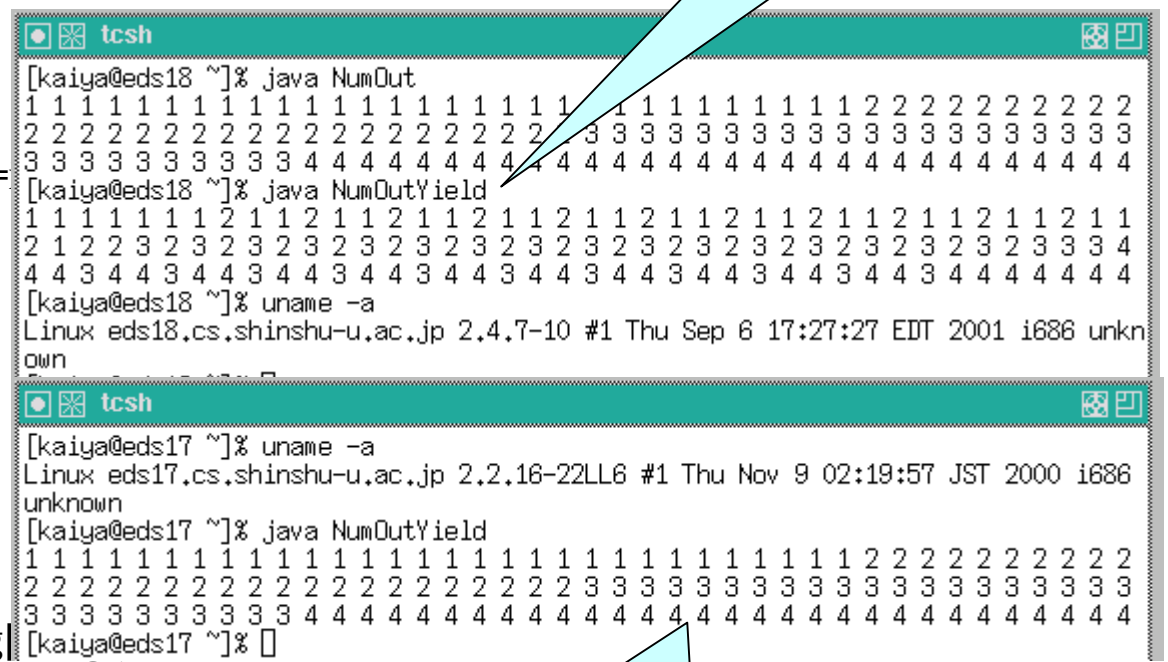
Yieldの効果

```
public class NumOutYield extends Thread{
private int id;
private static int max=30;
private static int tnum=4;

 NumOutYield(int id){ this.id=

 public void run(){
  for(int i=0; i<max; i++){
   System.out.print(id+" ");
   yield();
  }
 }

 public static void main(String[
  Thread.currentThread().setPriority(Thread.MAX_PRIORIT
  for(int i=0; i<tnum; i++)
   new NumOutYield(i+1).start();
 }
}
```



少しは，「並列に」動いているっぽくみえるようになった．

現3年のOSでは，うまく「並列に」動いているっぽく実行できなかった．Linuxカーネルのせい？

休止処理 sleep

- sleep(long x): Threadクラスに定義されており，そのスレッドをxミリ秒停止させる.
- 例えば，ランダムに停止時間を与えたい場合，以下のようなクラスを利用するなど.

```
import java.util.*;
class Waiting{
private Random r;
    Waiting(){ r=new Random();}
    void wait(Thread t){
        int w=r.nextInt(); w=(w>0? w: w*(-1))%500;
        try{ t.sleep(w);}catch(InterruptedException e){}
    }
}
```

sleepをつかいバラバラに見せる

```
import java.util.*;

public class NumOutSleep extends Thread{
// 中略
private Random r=new Random();
  // 中略
  public void run(){
    for(int i=0; i<max; i++){
      System.out.print(id+" ");
       try{Thread.currentThread().sleep(r.nextInt(2));}
      catch(Exception e){}
    }
  }
  // 中略
}
```

並行処理の大切なこと

- 安全性(safety): 好ましくない状態にならない
  - 干渉(interfere)がおきない.
  - デッドロック(dead lock)が無い.
- 生存性(liveness): やろうとしたことはいづれ処理される. (永遠に待たされることは無い)

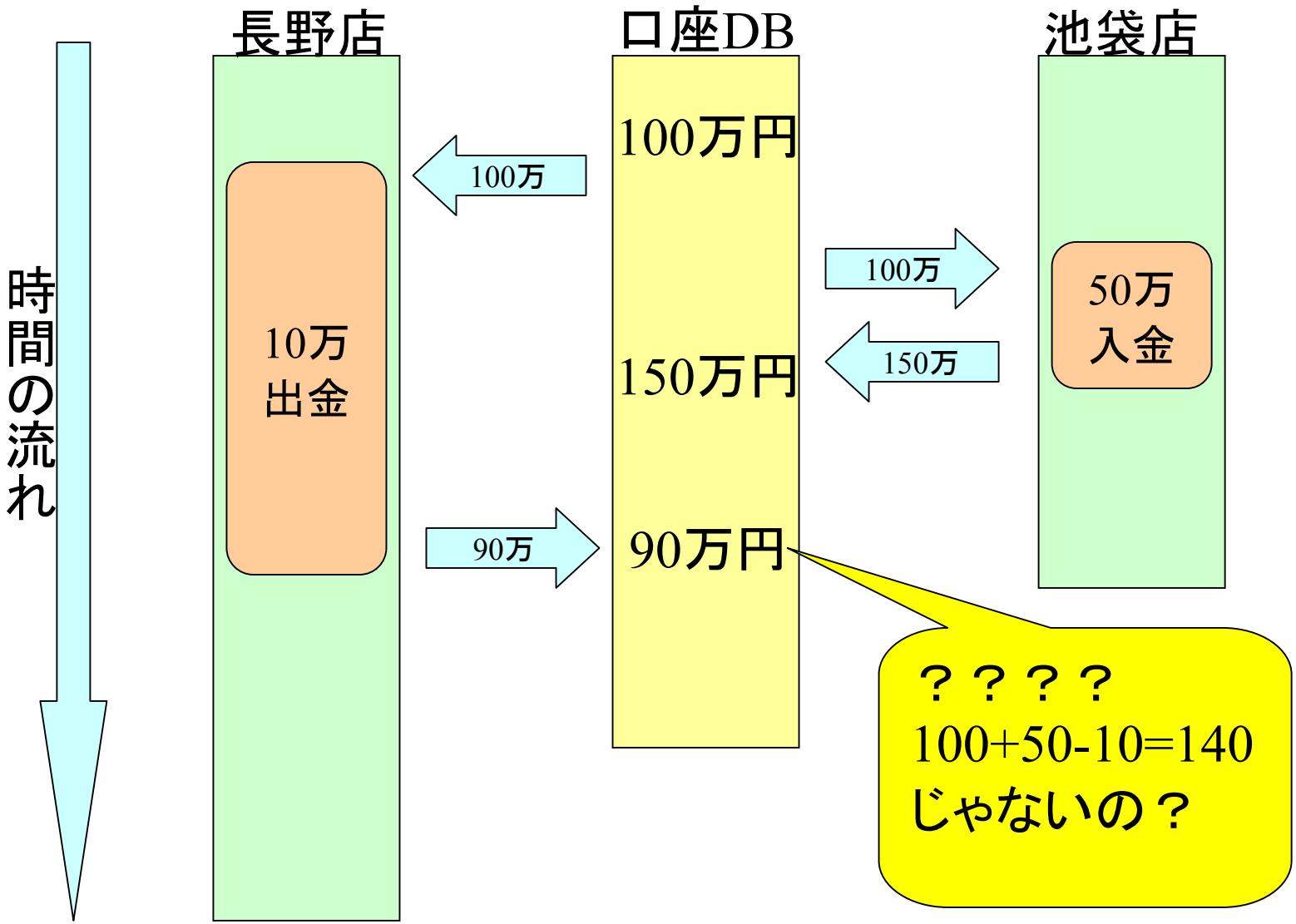
干渉とデットロック

- 干渉(interfere) 並行処理のミスでデータ等の一貫性が失われること.
  - 後述の排他制御である程度回避可能.
- デッドロック(dead lock)
  - 複数の資源を同時に必要とする複数のプロセス(スレッド)が，資源の一部をそれぞれに確保し，残りの資源が空くまで，それぞれが永久に待ってしまうこと.
  - 「哲学者の例題」参照.

排他制御による干渉の予防

- (スレッドだけでなく)並行処理一般に重視しなければいけない問題.
- 要は共有資源を同時に処理しちゃいけない.
- 一般にはロックという方法で排他制御するのが一般的.

有名な干渉の例


長野店
口座DB
池袋店
時間の流れ
100万円
100万
100万
50万
入金
10万
出金
150万
150万円
90万
90万円
？？？？
100+50-10=140
じゃないの？

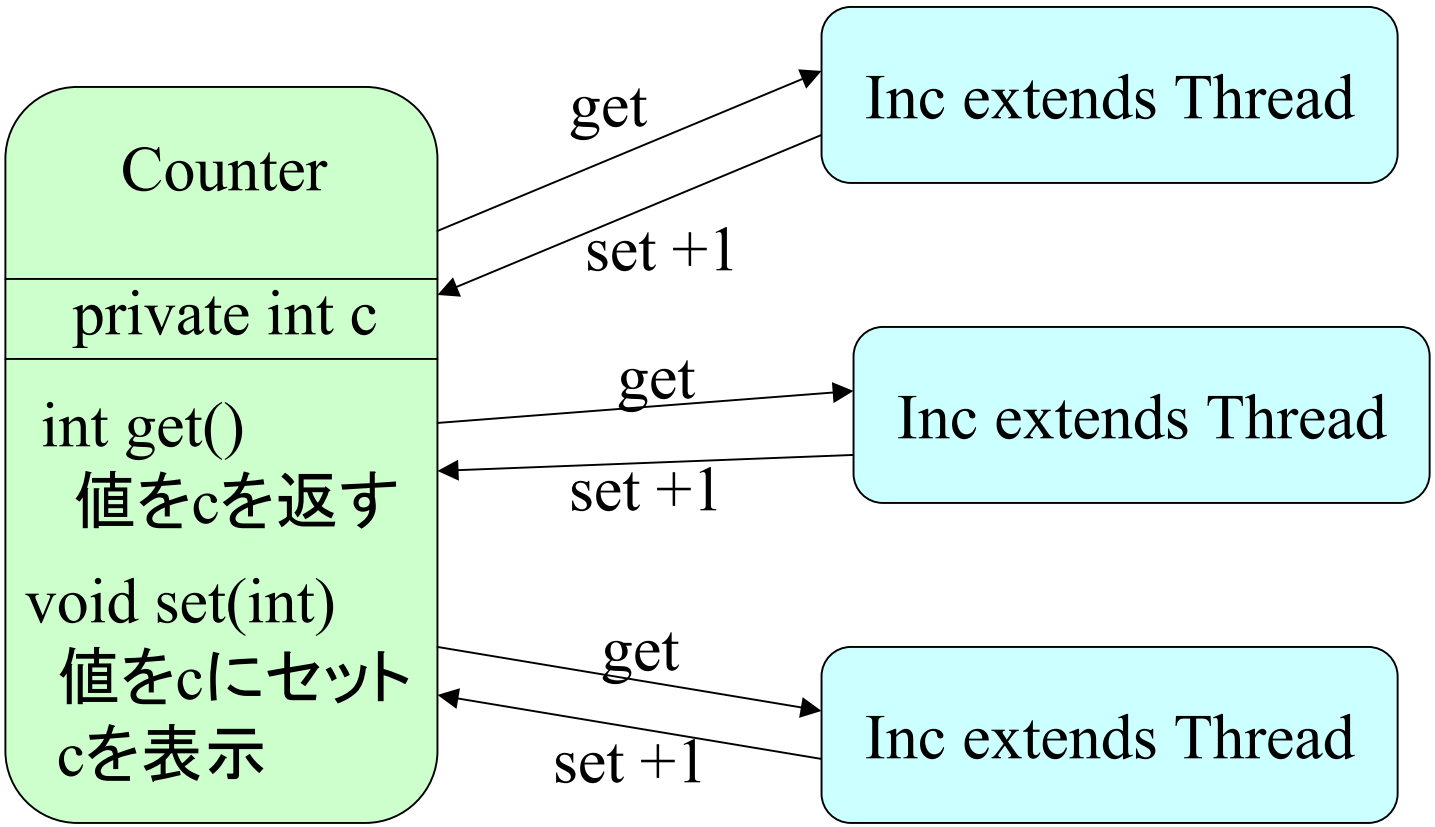
干渉発生のTIPS

- 1つの連続した処理が終わる前に，古いデータを読んで，別処理を行っているのがマズい．
- 逐次処理(not並行処理)の場合は問題なし．
- スレッドによる並行処理の場合，データ(変数，ファイル，レコード等)を特定のスレッドに一定期間のみ占有させる必要あり．

Javaにおける排他制御

- Synchronizedブロック
  - インスタンスをもとにした，処理ブロックのロック.
- Synchronized メソッド
  - メソッド毎のロック

synchronized ブロックの例



干渉が起きる理由

```
class Inc extends Thread{
// 中略
  public void run(){
    for(int i=0; i<often; i++){
      int tmp=c.get();
      tmp++;
      try{ sleep(r.nextInt(2));}catch(Exception e){}
      c.set(tmp);
    }
  }
// 中略
}
```

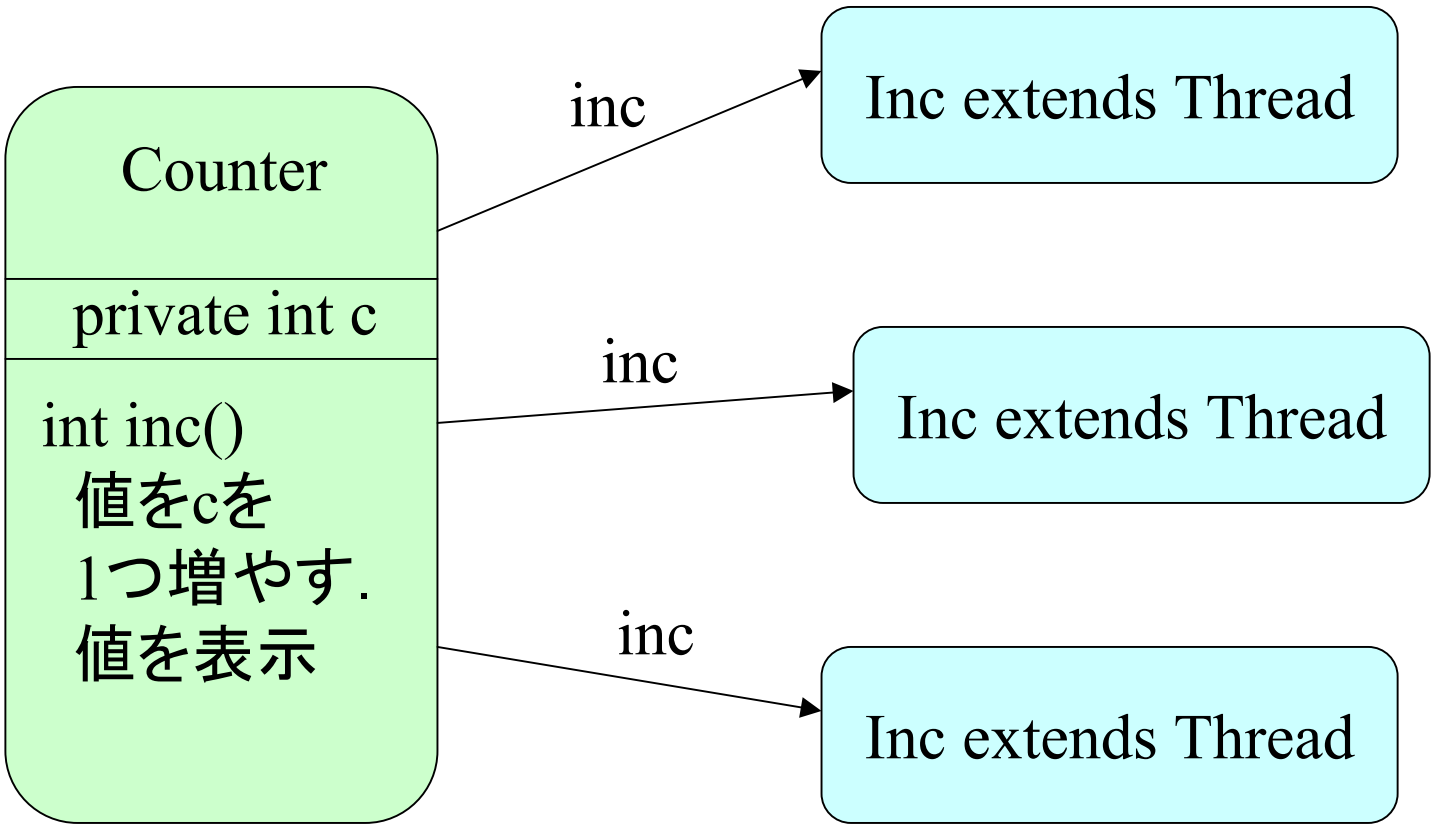
詳細はHPを参照

干渉の抑制

この処理の間は，
インスタンス c は
この処理を行うスレッド
に占有される．

```
public void run(){
   for(int i=0; i<often; i++){
    synchronized(c){
      int tmp=c.get();
      tmp++;
      try{ sleep(r.nextInt(2));}
      catch(Exception e){}
      c.set(tmp);
    }
   }
}
```

synchronized メソッドの例



干渉が起きる理由

```
class Counter{
private int c;
private Random r=new Random();

  public void inc(){
    int tmp=c;
    tmp++;
    try{Thread.currentThread().sleep(r.nextInt(2));}
    catch(Exception e){}
    c=tmp;
    System.out.print(c+" ");
  }
}
```

干渉の抑制

このメソッドが呼ばれたインスタンスは，メソッドの処理が終わるまで，メソッドを呼んだスレッドに占有される．

```
class Counter{
private int c;
private Random r=new Random();

 synchronized public void inc(){
   int tmp=c;
   tmp++;
   try{Thread.currentThread().sleep(r.nextInt(2));}
   catch(Exception e){}
   c=tmp;
   System.out.print(c+" ");
 }
}
```

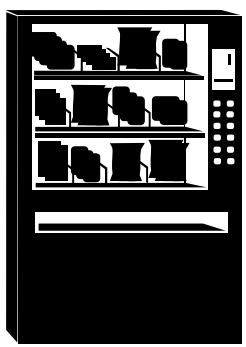
デッドロックについて

- HPにある「哲学者」の問題等を参照.
- 詳細は省略.

共有資源と待ち行列

スレッドの待ち行列

Javaのロックメカニズムでは,
ロックされる共有資源(インスタンス)と,
それの利用を待っているスレッドの
待ち行列が存在する.

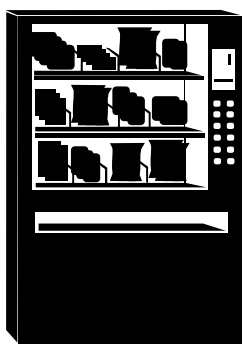


共有資源 (インスタンス)

行列から一時外れる

スレッドの待ち行列

インスタンスがある条件を満たしていないと，処理を行えないスレッド(A)があるとして，その条件は他のスレッド(B)が満たせるとしたら，Aは順番を譲って，他のスレッドが処理を終えるのを待たねばならない.



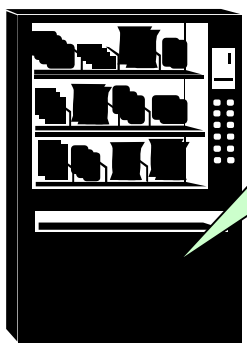
共有資源 (インスタンス)

例

スレッドの待ち行列

タバコ業者で一す.
(マイルドセブン補給できます)

マイルドセブン
がほしい.



マイルドセブン
品切れ

共有資源 (インスタンス)

wait, notify, notifyAll

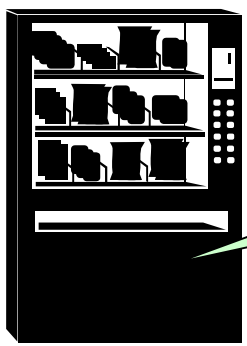
- あるインスタンスをロック中のスレッドが呼び出せるメソッド.
- Objectクラスで定義されている.
  - ⇒ 全てのJavaクラスで定義されている.
- wait() これを呼び出したスレッドを待機状態にする.
- notify() 待機中のスレッドを1つ再開させる.
- notifyAll() 待機中のスレッドを全て再開させる.

例

スレッドの待ち行列

タバコ業者でーす.
(マイルドセブン補給できます)
補給がすんだら，notify等を呼んで，
待機中のスレッドを列にもどす.

マイルドセブンがほしい.
wait() を呼んで待機.
⇒ 一時，列から離れる.

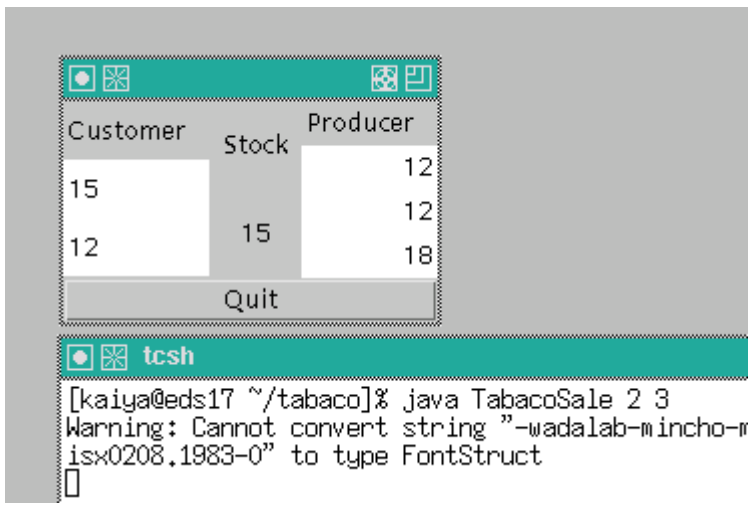


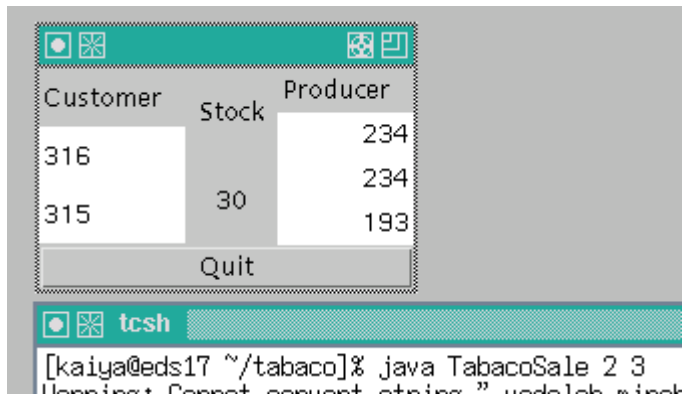
マイルドセブン品切れ
業者が補給すれば，
品切れは解除される.

共有資源 (インスタンス)

プログラムの例

所謂，生産者･消費者問題





生産過剰気味の構成にしてあるので，すぐにストック上限(本プログラムでは30に固定)あたりをうろうろする．
しかし，止まりはしない．

コード (Stock.java)

```
class Stock extends IntLabel{
private static final int max=30;

// 中略

// ストックから販売する
synchronized void buy(){
    while(empty()){
      try{ wait(); }
      catch(Exception e){}
    }
    super.dec();
    notifyAll();
  }
```

```
// ストックに補給する
synchronized void supply(){
    while(full()){
      try{ wait(); }
      catch(Exception e){}
    }
    super.inc();
    notifyAll();
  }

}
```

コード (Customer, Producer)

```
class Customer extends LabelUpdater{
  Customer(IntLabel l, Stock s){
    super(l,s);
  }

  public void run(){
    while(true){
      stock().buy();
      inc(); // 買った個数を記録
      sleeping(1000);
    }
  }
}
```

```
class Producer extends LabelUpdater{
  Producer(IntLabel l, Stock s){
    super(l, s);
  }

  public void run(){
    while(true){
      stock().supply();
      inc(); // 納品した個数を記録
      sleeping(1000);
    }
  }
}
```

双方Threadのサブ(サブ)クラス

以上