上級プログラミング１（第１１回）

工学部　情報工学科
木村昌臣

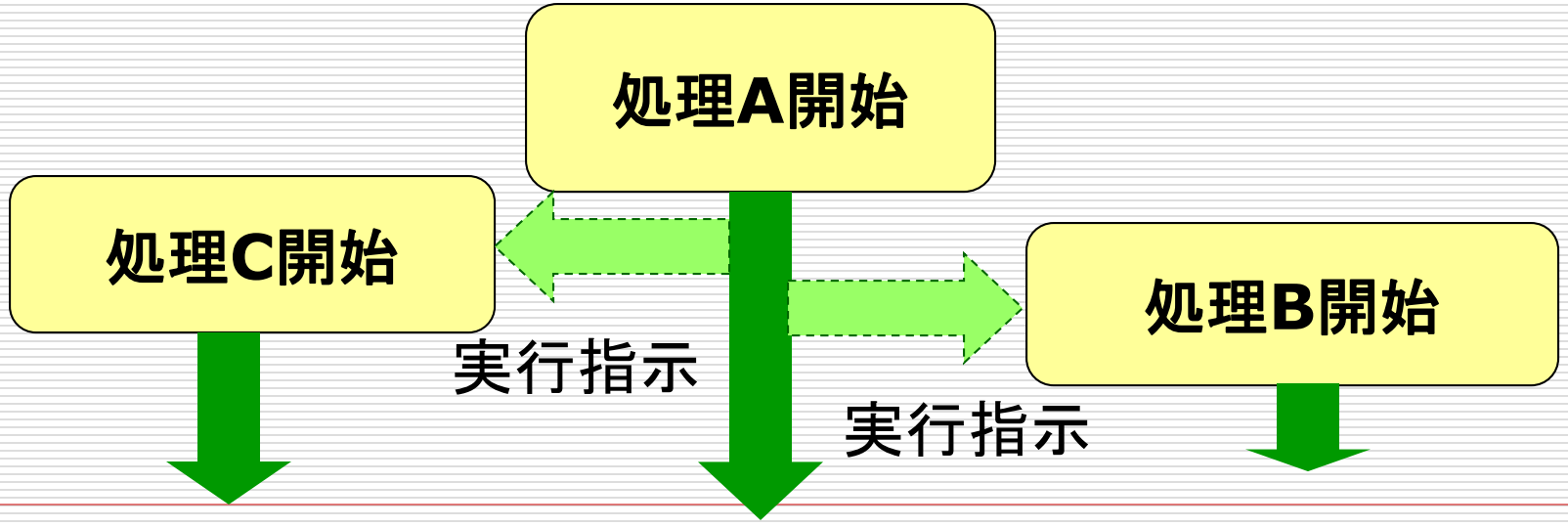
今日のテーマ

- スレッドの復習
- スレッドの同期
  - 銀行口座にたとえて

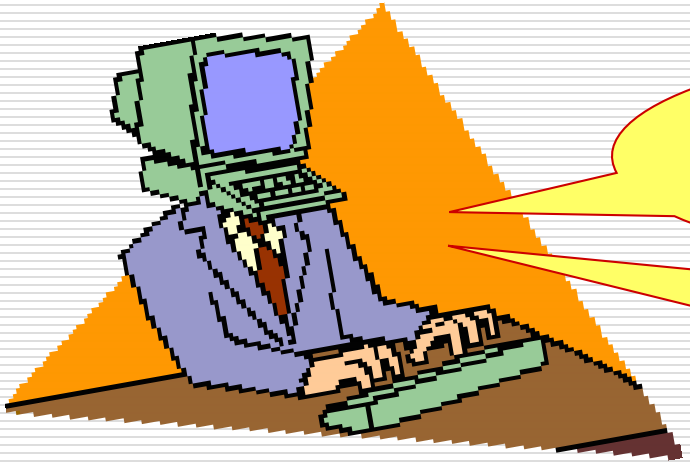
スレッドの復習

マルチスレッド

- 1つのプログラム（正確にはプロセス）の中で、処理の流れ（スレッド）を同時に複数実行するしくみ
- 複数のスレッドが同時に複数実行されるプログラムをマルチスレッドプログラムと呼ぶ（普通のプログラムはシングルスレッドプログラム）



例えていうなら



メール送ろうっと！

ああっ、レポートも
つくらなきゃ！

メール作成開始

別のスレッドを実行

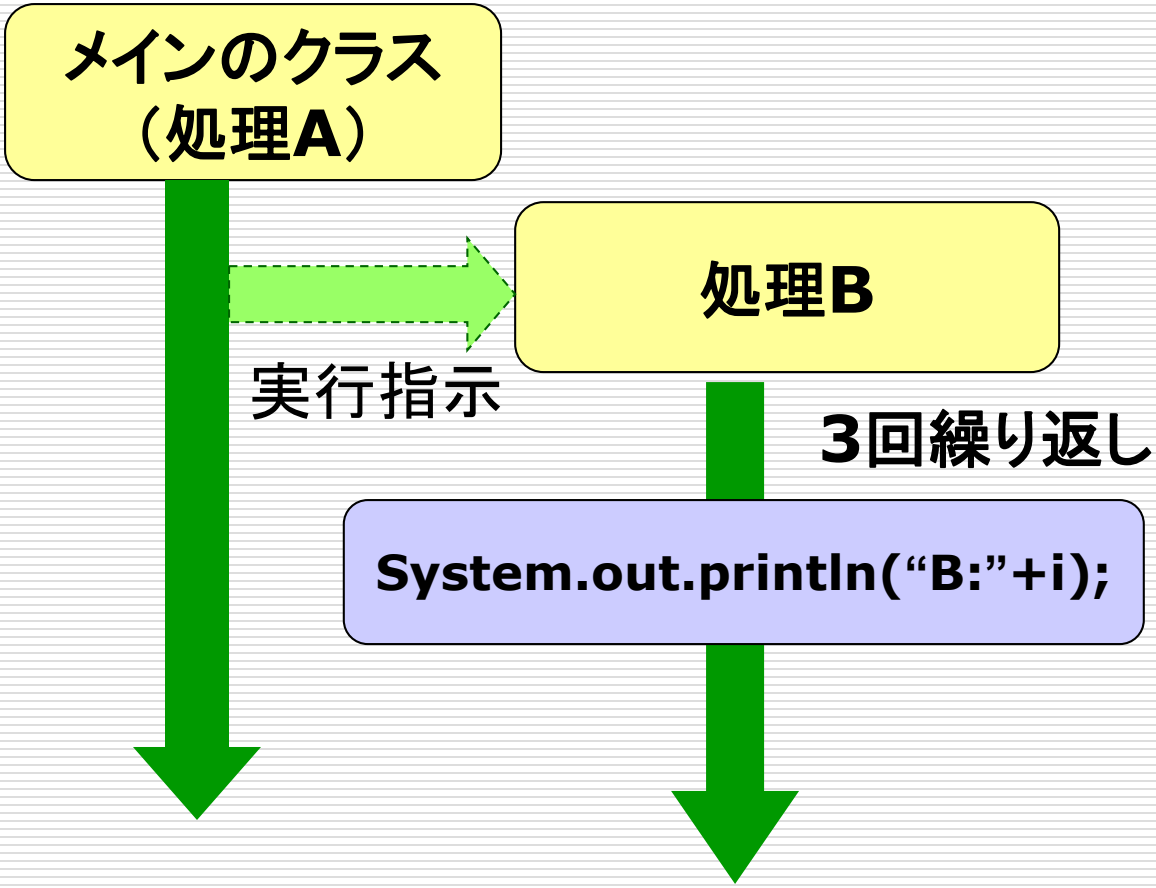
レポート作成開始

複数の処理を
同時進行で行う

マルチスレッドを実現するクラスの作り方

- Threadクラスを継承する方法
  - Threadクラスを継承したクラスのオブジェクトを生成
- Runnableインターフェイスを実装したクラスを利用する方法
  - そのオブジェクトをコンストラクタの引数としてThreadクラスのオブジェクトを生成

マルチスレッドプログラムの簡単な例



ソースプログラム(1) 処理A

```
class ThreadMain {
        public static void main(String[] args) {
            ThreadSmpl shoriB = new ThreadSmpl();
            shoriB.start();
            for(int i=0; i<3; i++){
                        System.out.println("A:"+i);
            }
            System.out.println("Main is ended.");
        }
}
```

処理Bを新規スレッドとしてstart()メソッドで実行後、A:・・・を表示

ソースプログラム(2)　処理B

```
class ThreadSmpl extends Thread {
        public void run(){
                for(int i=0; i<3; i++){
                        System.out.println("B:"+i);
                }
        }
}
```

① Treadクラスを継承する
② run()メソッドをオーバーライドし、同時実行させる処理を記述する。
run()メソッドはstart()メソッドから呼び出される

ソースプログラム(1’) 処理A

```
class TreadMain {
        public static void main(String[] args) {
            ThreadSmpl smpB = new ThreadSmpl();
            Thread shoriB=new Thread(smpB);
            shoriB.start();
            for(int i=0; i<3; i++){
                        System.out.println("A:"+i);
            }
            System.out.println("Main is ended.");
        }
}
```

ソースプログラム(2') 処理B

```
class ThreadSmpl implements Runnable {
        public void run(){
                for(int i=0; i<3; i++){
                        System.out.println("B:"+i);
                }
        }
}
```

① Runnableインターフェイスを実装する
② run()メソッドをオーバーライドし、同時実行させる処理を記述する。

スレッドの休止

Threadクラスのsleepメソッドを使う

```
class ThreadSmpl extends Thread {
    public void run(){
        for(int i=0; i<3; i++){
          try{
                Thread.sleep(1000);
          }catch(Exception  e){
                e.printStackTrace();
          }
                System.out.println(“B:"+i);
        }
    }
}
```

スレッドの同期

二人で使っている銀行口座にお金を預けたい

銀行口座

残金＝¥3000

¥5000

takashi

¥5000

¥5000

naoto

¥5000

次のようにクラスを定義する

```
class Bank{
   int  okane=3000;
   void addOkane(int i){
      int work=okane;
      try{
         Thread.sleep(1000)
      }catch(Exception e){}
        okane=work+i;
   }
   int getOkane(){
        return okane;
   }
}
```

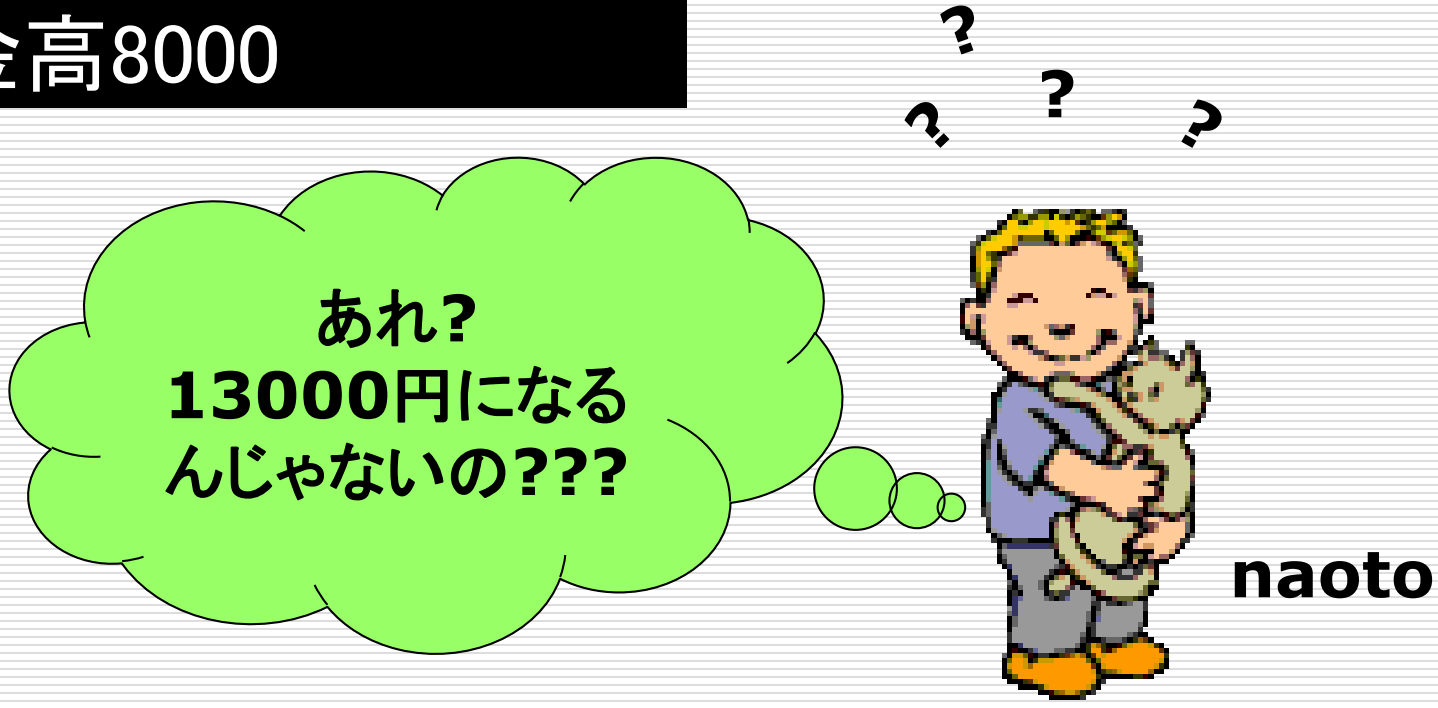
```
class Boy extends Thread{
    int i=0;    Bank b;
    Boy(Bank b){
       this.i=b.getOkane();
       this.b=b;
    }
    public void run(){
       b.addOkane(5000);
    }
}
```

単純に考えるとこれでよさそうだが

```
class NonSync{
        public static void main(String[] args){
            Bank b = new Bank();
            Boy takashi = new Boy(b);
            Boy naoto= new Boy(b);
            takashi.start();   //タカシがお金を預ける
            naoto.start();     //ナオトがお金を預ける
            try{        takashi.join();    //二人が振り込み
                        naoto.join();      //終わるまで待つ
            }catch(Exception e){e.printStackTrace();}
            System.out.println("預金高"+b.getOkane());
        }
}
```

実行してみると・・・

```
C:¥>java NonSync
預金高8000
```



原因は？

銀行口座

残金＝¥8000

¥3000

takashi

¥5000

本当はタカシが終わってから
ナオトが作業を始めるべき!!

¥8000

¥5000

スレッドセーフ

- もとのプログラムをマルチスレッド化しても問題が起こらないこと、もしくはそのためのスレッドの仕組み
  - 単一スレッドを動かすだけでは問題は顕在化しないことが多い
  - オブジェクトや変数を複数のスレッドで共有し、変更を加える場合などで問題が発生する場合がある
- この例は、明らかにスレッドセーフではない

どうしたらいい？

- タカシの操作が終わるまで、ナオトの処理を待たせればよい
- すなわち、スレッド間の同期をとればよい
  - 同期をとるべきメソッドにsynchronized演算子を付ける
  - 同期をとるべき部分をsynchronizedブロックで囲む

メソッドにsynchronized修飾子をつける方法（ほかは変えない）

```
class Bank{
        int  okane=3000;
        synchronized void addOkane(int i){
                int work=okane;
                try{ Thread.sleep(1000);
                }catch(Exception e){}
                okane=work+i;
        }
        int getOkane(){
                return okane;
        }
}
```

あるスレッドがこの処理を終えるまでロックがかかり他のスレッドは待たされる

同期を取る部分をsynchronizedブロックで囲む方法（ほかは変えない）

```
class Boy extends Thread{
    int i=0;    Bank b;
    Boy(Bank b){
       this.i=b.getOkane();
       this.b=b;
    }
    public void run(){
        synchronized(b){
                b.addOkane(5000);
        }
    }
}
```

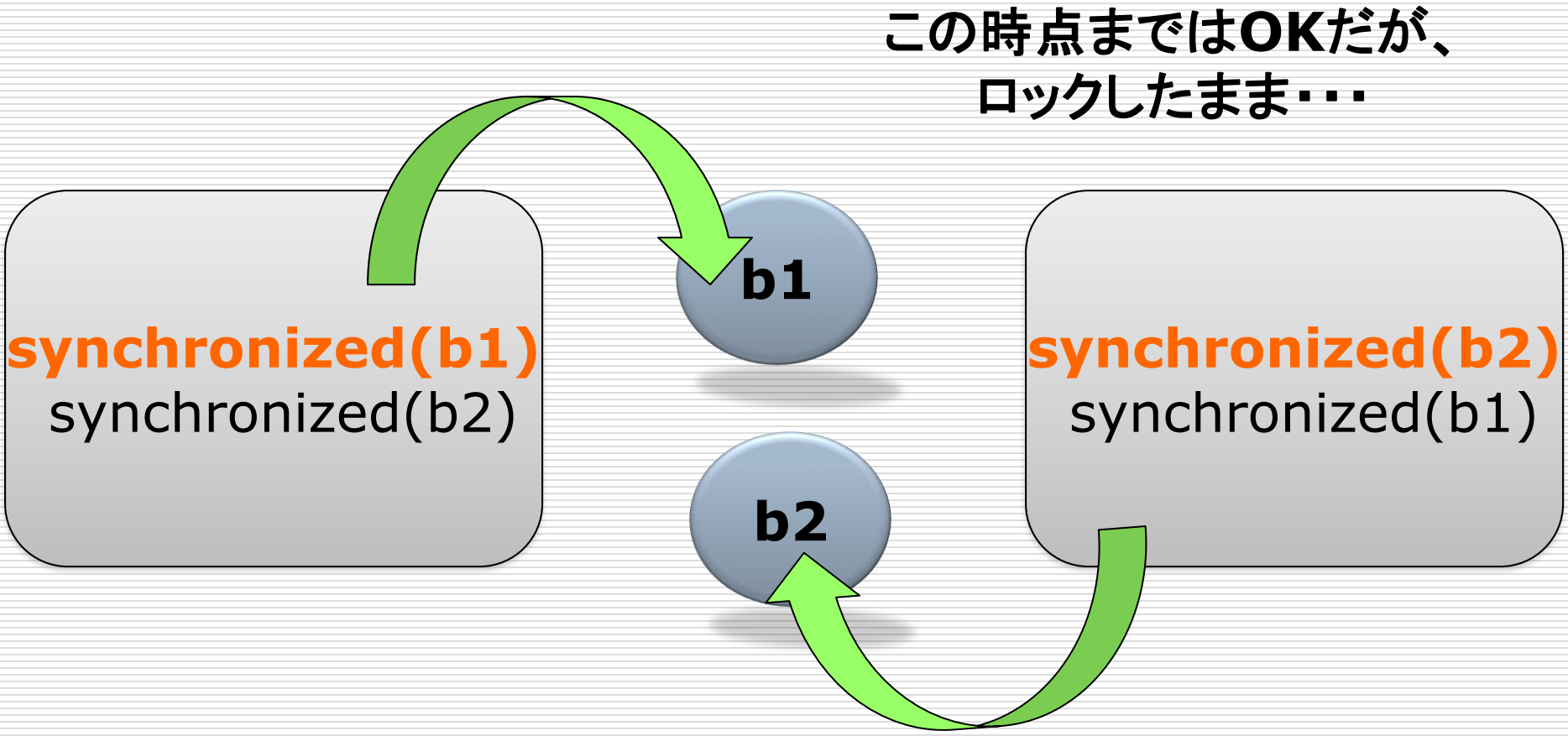
引数のオブジェクト（今の場合b）がアクセスされている間は他のスレッドはアクセスできない（ロックされる）

注意しなければならない点

あるオブジェクトをロックしながら、
別のオブジェクトをロックすると問題が起こる場合がある

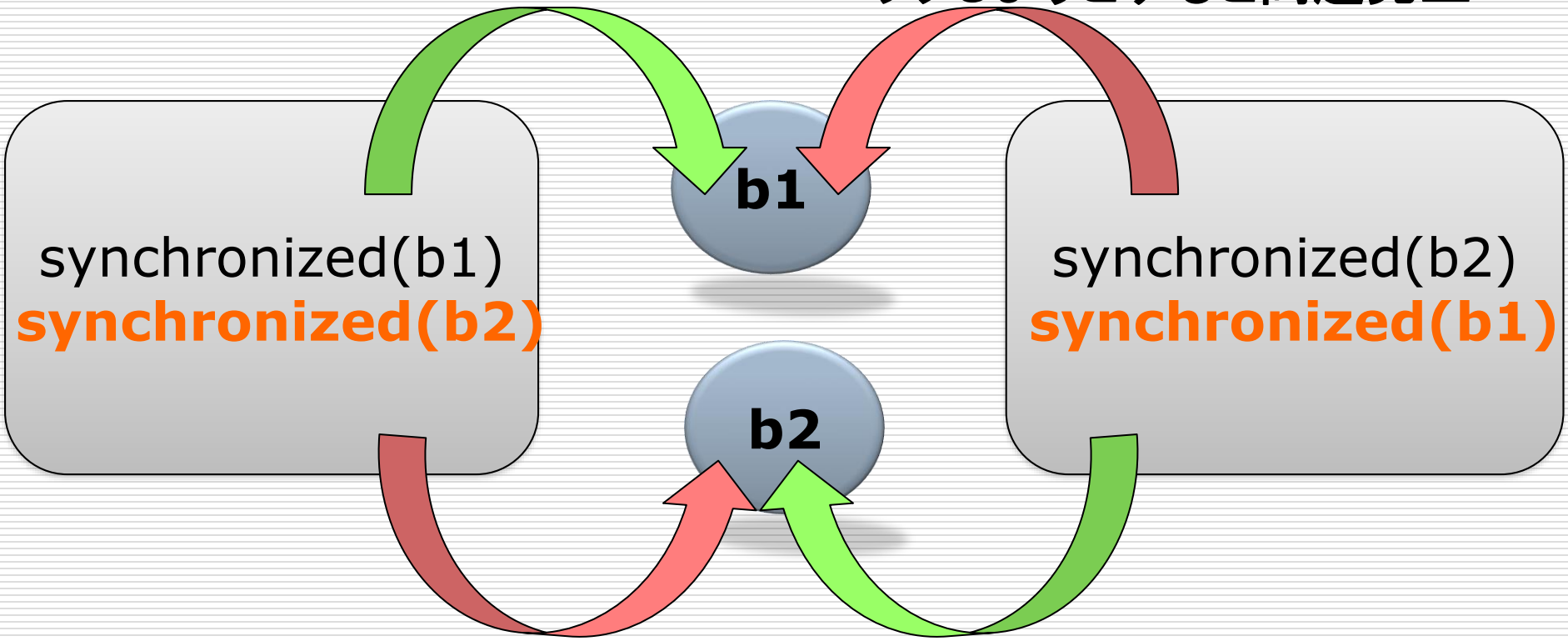
```
synchronized(b1){
        ...
        ...
        synchrnoized(b2){
                ...
                ...
        }
}
```

こんなことが起こりうる(1)



こんなことが起こりうる(2)
・・・・デッドロック

ロックしたまま次のオブジェクトを
ロックしようとすると問題発生



すでにロックされているオブジェクトを互いに待ち続ける

デッドロックの解消法

- ロックをかけられるオブジェクトに優先順位をつけて、必ずその順番でロックをかける