プログラミング言語各論

第4回

2005年度担当: 中井央

2005年10月13日

概 要

今回はスレッドについて入門的なことがらを扱う。

1 はじめに

最近の計算機では、ウィンドウ環境で複数のプログラムが動いているのが当たり前である。昔の計算機では、1つのプログラムを実行するとそれが終了するまで別のプログラムを動かすことはできなかった。1つの計算機 (CPU) 上で複数のプログラムを動かすには、その複数のプログラムをある単位時間ごとに切替えてやる必要がある。このようなプログラムの単位をプロセスという。

一方、1つのプログラムの中にも複数の計算の流れを考えることが可能である。この流れのことをスレッドと呼ぶ。スレッドを用いると1つのプログラム内のいくつかの作業を並行して行うプログラムを記述することが可能となる。

ちなみに並行と似た言葉で並列という言葉がある。並行して仕事を行うというのは、一人の人が「同時にこなしています」といって、ある一定期間の中でいくつかの仕事をしている場合に例えると分かりやすいかも知れない。その人の脳味噌は1つなので実際には一時に2つの仕事をしていることはないが、ある一定期間で見れば複数の仕事をしていることは良くあることである。一方、ある1つの仕事を複数の人間で手分けをして行っている場合、こちらは並列に仕事をしているといえる。並列に仕事をする方法を大きく分けると同じ仕事内容を均等に各自に与える方法と別々の仕事内容を手分けする場合がある[1]。

[1]並列処理に興味がある人は情報処理演習を受講すると良い。

Java にはプログラムに対し、並行に仕事をさせるための仕組みが備わっている。先ほど述べた仕事の流れ=スレッドを複数用いることで並行して仕事をこなせるようになる。複数のスレッド (を扱うこと) のことをマルチスレッドいう。

なお、最近は CPU を複数搭載した計算機も珍しくなくなってきている (一般には珍しいかも…)。このような計算機で、複数のスレッドを別々の CPU 上で実行するように割り当てれば、それらの処理は並列に行われることになる。

なお、このテキストの作成に当たっては、[1] と [2] を参考にした。

2 まずは簡単な例から

まずはサンプルプログラムを図1に示す。

```
 1 public class Sample01 {
 2   public static void main(String args[]){
 3     Count01 c1 = new Count01();
 4     c1.print();
 5     for(int i=0; i<20; i++){
 6       System.out.println("main:" + i);
 7     }
 8   }
 9 }
10
11 class Count01 {
12   public void print(){
13     for(int i=0; i<20; i++){
14       System.out.println("count:" + i);
15     }
16   }
17 }
```

図 1: Sample01.java

```
 1 public class Sample02 {
 2   public static void main(String args[]){
 3     Count02 c1 = new Count02();
 4     c1.start();
 5     for(int i=0; i<20; i++){
 6       System.out.println("main:" + i);
 7     }
 8   }
 9 }
10
11 class Count02 extends Thread {
12   public void run(){
13     for(int i=0; i<20; i++){
14       System.out.println("count:" + i);
15     }
16   }
17 }
```

図 2: Sample02.java

図 1 はマルチスレッドではないプログラムであり、図 2 はマルチスレッドのプログラムである。図 1 では、4 行目で Count01 の print メソッドの呼出しを行い、それが終了した後、5〜7 行目の for 文に制御が移る。

図 2 では、Count02 クラスは Thread クラスを継承している。4 行目では Count02 クラスの start メソッドが呼ばれている。start メソッドは Thread クラスで定義されているもので、新たなスレッドを開始することを意味する。

図 1 のプログラムを実行すると、**必ず**`count:`の出力の後、`main:`の出力となる。

一方、図 2 のプログラムを実行すると、`main:`の出力が先になるか、`main:`と `count:`が交互に現れるような出力となる。これは図 2 のプログラムでは、main メソッドと `c1.start` によって起動される Count02 の `run` メソッドが並行に実行されるからである。

我々が利用している uni では、CPU を複数搭載しているため、実行する毎に出力結果が異なる。CPU が 1 つの計算機でも実行毎に出力結果が異なることもあるが、多くの場合、c1.start が呼ばれ、Count02 の中の run が実行されるまでに main メソッド中の for 文の実行が先に開始され、結果として `main:`が先に表示される形になっているようだ。繰り返しの回数を 20 ではなく、10000 に変えれば、1 つの CPU を搭載したマシンでも図 2 の出力結果は `main:`と `count:`が交互に現れるものとなる。

この出力結果から、Thread クラスを利用し、複数のスレッドを用いることで、2 つの処理の流れを作り、それぞれをある一定時間で交互に実行していることが読みとれる。

以下の節で、新たなスレッドを作る方法や、スレッドを使う上での問題点とその対策などについて述べる。

演習 1

図 1 と図 2 を入力し、コンパイル、実行せよ。上述の内容を実行結果から確認せよ。

3 マルチスレッドにする方法

Java でプログラムをマルチスレッドにする方法として、Thread クラスを継承する方法と Runnable インタフェースを実装する方法がある。Java では多重継承を許していないため、Thread クラスを継承できない場合、Runnable インタフェースを実装する方法を使う。どちらにしても Thread クラスを用いることには変わりはない。それを以下にそれぞれ示す。

3.1 Thread クラスを用いる方法

この例は図 2 に示した通りである。別のスレッドとして実行したい主体を持つクラスを Thread クラスのサブクラスとして定義し、その実行したい部分を `run` メソッドの中に定義する。

新たにスレッドを起動したいところ (ここでは Sample02 クラスの main メソッド) で別のスレッドとして実行したい主体を持つクラス (Count02) のインスタンスを生成し、その start メソッドを呼び出すことで新たなスレッドが起動され、そのスレッドにおいて run メソッドが呼び出される。

3.2 Runnable インタフェースを用いる方法

図 2 のプログラムを Runnable インタフェースを使うように変更したものを図 3 に示す。

Runnable インタフェースを実装することで、run メソッドを実装することになるが、スレッドを起動するには結局 Thread クラスのインスタンスを使う。

```
 1 public class Sample03 {
 2   public static void main(String args[]){
 3     Count03 c1 = new Count03();
 4     Thread t1 = new Thread(c1);
 5     t1.start();
 6     for(int i=0; i<20; i++){
 7       System.out.println("main:" + i);
 8     }
 9   }
10 }
11
12 class Count03 implements Runnable {
13   public void run(){
14     for(int i=0; i<20; i++){
15       System.out.println("count:" + i);
16     }
17   }
18 }
```

図 3: Sample03.java

演習 2

図 3 を入力し、コンパイル、実行せよ。図 2 の実行結果と同様になることを確かめよ。

3.3 sleep

スレッドを一時停止するメソッドとして sleep が用意されている。sleep は static メソッドであり、ミリ秒単位で停止時間を指定するものとミリ秒に加えナノ秒で指定するものが用意されている。ただし、ここで指定する時間は目安程度と捉えておくことが無難である。たとえば、1001 ミリ秒を指定したからといって、正確に 1001 ミリ秒が計られるわけではなく、システムの内部でそれに近い時間の経過をチェックしているだけである。また、Java は動く環境が様々であることも、指定した時間が正確には計時されない理由である。

図 4 に sleep を使った例を示す。

ここで 6 行目に着目して欲しい。static メソッドであるので、クラス名. メソッド名という呼び出し方をしている。Java の文法では、クラス名の代わりにクラス型の変数を用いることも可能であるが、static メソッドであることを強調するようにソースプログラム中ではクラス名. メソッド名という記述を使うようにしよう。

なお、このメソッドは InterruptedException を投げる可能性がある。interrupt とは割り込みのことで、

```
 1 public class Sample04 {
 2   public static void main(String args[]){
 3     for(int i=0; i<20; i++){
 4       System.out.println("hoge!");
 5       try {
 6         Thread.sleep(1000);
 7       }
 8       catch (InterruptedException e){
 9       }
10     }
11   }
12 }
```

図 4: Sample04.java

sleep によって休止している状態を強制的に止める働きが割り込みであり、外部から割り込みをかけることが可能である。その際には sleep の処理を中断することになり、中断させられた場合、InterruptedException の例外を放出する。このため、sleep を使う場合には try 〜 catch 構文を記述する。

演習 3

図 4 を入力し、コンパイル、実行せよ。sleep に与える値を適当に変えて実行してみよ (100*i のようにすると変速的になる)。

3.4 スレッドの排他制御

マルチスレッドに関わらず、並行処理 (並列処理も含む) によるトランザクション処理の問題がある。例えば、ファイルを何人かで共有しているとする。A さんがそのファイルをエディタで開いたとする。開いたままどう編集しようか考えて寝てしまったとする。その間に B さんが同じファイルを開き、さっさと編集し、保存したとする。そのあと、A さんは起きて編集し、保存したら、B さんが保存した内容は全く残らずに消えてしまう。このような場合、A さんがエディタでファイルを開いている間は、他の人がそのファイルを開けないような仕組みがあれば、このような問題は回避できる。例えば、そのような仕組みを持ったエディタだけを使うことを考える。その仕組みは、例えば次のようになる。A さんがそのファイルをそのエディタで開いた際、そのファイルを今開いていることを示す特別

な名前のファイルをシステム上のどこかに保存する。簡単のため、これを lock ファイルと言う。B さんがそのエディタで同じファイルを開こうとした際、そのエディタは lock ファイルがあるかどうか確認し、あれば今そのファイルは他の人が編集中である旨表示し、エディタを終了させる。このような仕組みがあれば、上述のように B さんが施したファイルへの変更が無駄になってしまうということは回避できる。

```
 1 public class StringBufTest {
 2   static StringBufSub s = new StringBufSub();
 3   static public void main(String args[]){
 4     s.print();
 5   }
 6 }
 7
 8 class StringBufSub {
 9   StringBuffer buf;
10   StringBufSub(){
11     buf = new StringBuffer(
             "01234567890123456789");
12   }
13
14   public boolean chop(){
15     if (buf.length()>0){
16       buf.deleteCharAt(buf.length()-1);
17       return true;
18     }
19     return false;
20   }
21
22   public void print(){
23     int length = buf.length();
24     for(int i=0; i<length; i++){
25       System.out.print(buf.charAt(i));
26     }
27     System.out.println();
28   }
29 }
```

図 5: StringBufTest.java

さて、Java でマルチスレッドのプログラムを書いていると、この A さんと B さんの問題のようなことが起こる可能性がある。すなわち、1 つのデータを複数のスレッドが別々にアクセスすることで予期せぬデータの破壊につながることがある。まずはデータの破壊のサンプルを示し、それを回避する方法を示す。

図 5 はシングルスレッドで、StringBufSub が持つ文字列を一字ずつ画面に表示するだけのものである。14〜20 行目で定義している chop は使用していない。

図 6 は図 5 に手を加えたものである。4〜14 行目までは特別な書き方をしているので、まず、それを解説

```
 1 public class StringBufTest2 {
 2   static StringBufSub s = new StringBufSub();
 3   static public void main(String args[]){
 4     Thread t = new Thread(){
 5       public void run(){
 6         while(s.chop()){
 7           try {
 8             sleep(10);
 9           }
10           catch (InterruptedException e){
11           }
12         }
13       }
14     };
15     t.start();
16     s.print();
17   }
18 }
19
20 class StringBufSub {
21   StringBuffer buf;
22   StringBufSub(){
23     buf = new StringBuffer(
             "01234567890123456789");
24   }
25
26   public boolean chop(){
27     if (buf.length()>0){
28       buf.deleteCharAt(buf.length()-1);
29       return true;
30     }
31     return false;
32   }
33
34   public void print(){
35     int length = buf.length();
36     for(int i=0; i<length; i++){
37       try {
38         Thread.sleep(10);
39       }
40       catch (InterruptedException e){
41       }
42       System.out.print(buf.charAt(i));
43     }
44     System.out.println();
45   }
46 }
```

図 6: StringBufTest2.java

しておく。通常なら、new Thread(); として終りなのだが、その後ろに { と } で括った文の並びが来ている。これは**無名クラスを定義している**。このような書き方をすると Thread クラスを継承したクラスを記述できる。ただし、本来 class というキーワードの後ろにクラス名を記述すべきだが、ここで 1 つインスタンスを生成すれば十分というときにはこのような名前なしのクラスの宣言ができる。要するに 4〜14 行目では、

```
[nakai@uni]% java StringBufTest2
012345678Exception in thread "main" java.lang.Stri
ngIndexOutOfBoundsException: String index out of
range: 9
  at java.lang.StringBuffer.charAt(Unknown Source)
  at StringBufSub.print(StringBufTest2.java:42)
  at StringBufTest2.main(StringBufTest2.java:16)
```

図 7: StringBufTest2.java を実行した様子

Thread を継承した名前のない 5〜13 のような中身を持つクラスを宣言しているのである。

そうすると 4〜14 行目で定義した名無しのクラスについては、15 行目でスレッドとして実行を開始している。一方、main メソッドはそのまま print の実行を行なっており、この 2 つが並行して実行さることになる。

34 行目からの print メソッドでは、for 文によって buf の内容を一文字ずつ画面に表示しようとしている。一方、4〜14 行目では、同じ buf に対して、繰り返し、末尾の方から一文字ずつ削除を行なっている。

さて、この 2 つのスレッドが同時並行に実行されると、print メソッドの方は初めに繰り返し回数を確定し、繰り返し、buf の先頭から一文字ずつ表示しているのだが、その間にもう 1 つのスレッドによって buf の中身はその末尾から一文字ずつ減っていっている。そうすると繰り返しのカウンターが最後までいっていないのに表示する文字がなくなることになる。存在しない文字をアクセスしようとすると例外が発生する。

この例の場合、同じオブジェクトが持つ同じリソース (資源、ここでは buf のこと) へ、2 つのスレッドにより、別々のメソッドからアクセスがあり、そのリソースに変更が加えられることに問題がある。そこで、あるメソッドがそのリソースにアクセスしている間、別のメソッドが待たされる仕組みがあれば、この問題は回避される。これを実現するために synchronized というキーワードを用いたプログラムを図 8 に示す。

1 つのオブジェクトにはモニタと呼ばれる同期のための目印がある。そのオブジェクトの synchronized メソッドはいちどきに 1 つのスレッドだけが実行することができる。実行可能なスレッドはまず、そのオブジェクトのモニタを取る。このことをロックをかけるという。他のスレッドからそのオブジェクトの synchronized メソッドへアクセスしようとするとまず、ロックされて

```
 1 public class StringBufTest3 {
 2   static StringBufSub s = new StringBufSub();
 3   static public void main(String args[]){
 4     Thread t = new Thread(){
 5       public void run(){
 6         while(s.chop()){
 7           try {
 8             sleep(10);
 9           }
10           catch (InterruptedException e){
11           }
12         }
13       }
14     };
15     t.start();
16     s.print();
17   }
18 }
19
20 class StringBufSub {
21   StringBuffer buf;
22   StringBufSub(){
23     buf = new StringBuffer(
            "01234567890123456789");
24   }
25
26   public synchronized boolean chop(){
27     if (buf.length()>0){
28       buf.deleteCharAt(buf.length()-1);
29       return true;
30     }
31     return false;
32   }
33
34   public synchronized void print(){
35     int length = buf.length();
36     for(int i=0; i<length; i++){
37       try {
38         Thread.sleep(100);
39       }
40       catch (InterruptedException e){
41       }
42       System.out.print(buf.charAt(i));
43     }
44     System.out.println();
45   }
46 }
```

図 8: StringBufTest3.java

いるか確認する。ロックされていれば待たされる。ロックしているメソッドが実行を終了した場合、ロックは解放される。このことをアンロックするともいう。

さて、このモニタとは、この節の最初に述べた lock ファイルと同じ役目をするものである。図 6 のように 1 つのデータに複数のスレッドからアクセスする可能性がある場合、いちどきにどれか 1 つのメソッドだけがアクセスできるようにしなければならない。このと

```
 1 public class StringBufTest5 {
 2   static StringBuffer buf =
       new StringBuffer("01234567890123456789");
 3   static int x = 0;
 4
 5   static Object lock = new Object();
 6
 7   static StringBufSub s1 = new StringBufSub();
 8   static StringBufSub s2 = new StringBufSub();
 9
10   static public void main(String args[]){
11     s1.start();
12     try {
13       Thread.sleep(10*
                    Integer.parseInt(args[0]));
14     } catch (InterruptedException e){
15     }
16     x = 1;
17     s2.start();
18   }
19 }
20
21 class StringBufSub extends Thread {
22   StringBufSub(){
23   }
24
25   public void run(){
26     if (StringBufTest5.x == 0){
27       while(chop()){
28         try {
29           sleep(10);
30         } catch (InterruptedException e){
31         }
32       }
33     }
34     else {
35       print();
36     }
37   }
38
39   public synchronized boolean chop(){
40     if (StringBufTest5.buf.length()>0){
41       StringBufTest5.buf.deleteCharAt
           (StringBufTest5.buf.length()-1);
42       return true;
43     }
44     return false;
45   }
46
47   public synchronized void print(){
48     int length = StringBufTest5.buf.length();
49     for(int i = 0; i<length; i++){
50       try {
51         Thread.sleep(100);
52       }
53       catch (InterruptedException e){
54       }
55       System.out.print
           (StringBufTest5.buf.charAt(i));
56     }
57     System.out.println();
58   }
59 }
```

図 9: StringBufTest5.java

き、synchronized を付与したメソッドは、自身 (this) をモニタとする。

演習 4

1. 図 5 を入力し、コンパイル、実行せよ。
2. 図 6 を入力し、コンパイル、実行せよ。図 7 のようなメッセージが表示されることを確認せよ。
3. 図 8 を入力し、コンパイル、実行せよ。

3.5 synchronized ブロック

上で述べたメソッドに synchronized をつける方法は、ここで述べる synchronized ブロックの特殊なケースである。メソッド中のいくつかの文の並びの間だけを対象にして、同期をとる指示をするための synchronized ブロックがある。次のような書式になる。

```
synchronized(OBJ){
  同期を取って実行したい文の並び
}
```

メソッドに synchronized をつけた場合、OBJ の部分は this となる。したがって、図 8 の 34〜45 行目は次のようにも書ける。

```
34   public void print(){
       synchronized(this){
35       int length = buf.length();
...
44       System.out.println();
       }
45   }
```

synchronized ブロックを使う時に指定するオブジェクトはなんでもよい。同時にそのブロックが実行されないようになんらかのオブジェクトにロックをかけられればよい。このためにロックのためだけのオブジェクトを生成するようなプログラムも考えられる。

```
 1 public class StringBufTest6 {
 2   static StringBuffer buf =
       new StringBuffer("01234567890123456789");
 3   static int x = 0;
 4
 5   static Object lock = new Object();
 6
 7   static StringBufSub s1 = new StringBufSub();
 8   static StringBufSub s2 = new StringBufSub();
 9
10   static public void main(String args[]){
11     s1.start();
12     try {
13       Thread.sleep(10*
                      Integer.parseInt(args[0]));
14     } catch (InterruptedException e){
15     }
16     x = 1;
17     s2.start();
18   }
19 }
20
21 class StringBufSub extends Thread {
22   StringBufSub(){
23   }
24
25   public void run(){
26     if (StringBufTest6.x == 0){
27       while(chop()){
28         try {
29           sleep(10);
30         } catch (InterruptedException e){
31         }
32       }
33     }
34     else {
35       print();
36     }
37   }
38
39   public boolean chop(){
40     synchronized (StringBufTest6.lock){
41       if (StringBufTest6.buf.length()>0){
42         StringBufTest6.buf.deleteCharAt
             (StringBufTest6.buf.length()-1);
43         return true;
44       }
45       return false;
46     }
47   }
48
49   public void print(){
50     synchronized (StringBufTest6.lock){
51       int length = StringBufTest6.buf.length();
52       for(int i = 0; i<length; i++){
53         try {
54           Thread.sleep(100);
55         }
56         catch (InterruptedException e){
57         }
58         System.out.print
             (StringBufTest6.buf.charAt(i));
59       }
60       System.out.println();
61     }
62   }
63 }
```

図 10: StringBufTest6.java

3.6 synchronized 使用上の注意

クラスの中にメソッドを定義する際、synchronized メソッドとそうでないメソッドを混在できる。マルチスレッドによって、計算の複数の流れがある場合、1 つのオブジェクトに並行して複数のスレッドからアクセスされる可能性がある。このとき、synchronized がついているメソッドについては同時にどれか 1 つだけしか実行できない。ところで、synchronized がついていないメソッドについてはこの制約はない。

したがって、**同時並行にアクセスを禁止したいリソースに対しては、プログラムを組む人間が注意をしてメソッドを定義する必要がある。**一方で synchronized メソッドにし、アクセスに制限を設けても、別の synchronized がついていないメソッドでそのリソースへアクセスをしていたら、上記の問題は解決されない。

例えば、図 8 においてもどちらかの synchronized をとってコンパイル、実行するとエラーとなる。

さて、ロックについてもう少し理解を深めるための特殊な例題を考えてみた。これを図 9 と図 10 に示す。

図 9 は失敗例で、それを手直ししたものが図 10 である。これまでの例とは異なり、アクセスするデータをクラス StringBufTest5 の中に置き、スレッドとして走らせたいオブジェクトを表現するクラス StringBufSub から、すなわち、外部からアクセスするようにした。情報隠蔽の考え方からすれば、あまり良くない例なのだが、ここでは synchronized の働きを示したかったので、しかたなくこのような構成にしていることを了承されたい。

さて、図 9 について簡単に説明していこう。StringBufTest5 には StringBuffer の static なオブジェクトがある (2 行目)。5 行目はこのプログラムでは使用していない。21 行目からのクラス StringBufSub は Thread クラスを継承している。7, 8 行目でそのインスタンス

を生成している。これらはスレッドとして実行できる。10 行目からの main メソッドでは、まず、s1 をスレッドとして走らせ、少し間を置いて、s2 をスレッドとして走らせている。main の仕事はこれだけである。

クラス StringBufSub では run メソッドを定義し、スレッドとしての実行ができるようにしている。ここでは StringBufTest5 の持つ x の値に応じて、chop メソッドを呼ぶか print メソッドを呼ぶかしている。chop メソッドおよび print メソッドは基本的には図 8 と同じである。異なるのは StringBufTest5 の持つ buf へアクセスしている点である。

実行結果は図 11 のようになる。

```
[nakai@uni]% java StringBufTest5 10
01java.lang.StringIndexOutOfBoundsException:
 String index out of range: 2
  at java.lang.StringBuffer.charAt(Unknown Source)
  at StringBufSub.print(StringBufTest5.java:55)
  at StringBufSub.run(StringBufTest5.java:35)
```

図 11: 図 9 を実行した結果

synchronized をつけているにも関わらず、図 6 の時のようなエラーメッセージが表示されている。

いま、s1 と s2 は別のスレッドとなっている。s1 のメソッドは s1 をモニタとする。s2 のメソッドは s2 をモニタとする。StringBufTest5 の持つ buf へのアクセスに対して排他制御を行ないたいわけだが、それぞれが別々のモニタを持っていても仕方がない。すなわち、buf へのアクセスのためのモニタを用意し、それに対してモニタを持っているものだけが実行できるようにしなければならない。そこで、図 10 では StringBufTest6 に lock というオブジェクト (を持つ変数) を用意した。synchronized ブロックを使用し、lock をモニタとするようにした。

すなわち、排他制御を行ないたい場合、排他制御の対象に対して、1 つのモニタを用意する必要がある。これがたまたま同一オブジェクトであれば、メソッドを synchronized にすることで制御できるが、そうでなければ、制御のためのモニタを設置するようにしなければならない。

図 10 の実行結果を図 12 に示す。

```
[nakai@nakai1]% java StringBufTest6 10
01234567890123
```

図 12: 図 10 の実行結果

演習 5

1. 図 8 で、どちらかの synchronized をとり、コンパイル、実行し、エラーメッセージが出力されることを確認せよ。
2. 図 9 を入力し、コンパイル、実行せよ。実行にあたってはコマンドライン引数として適当な整数を与えること。これは 13 行目で使用され、sleep へ与える引数となる。
3. 図 10 を入力し、コンパイル、実行せよ。実行にあたってはコマンドライン引数として適当な整数を与えること。これは 13 行目で使用され、sleep へ与える引数となる。

※注： 図 9 と図 10 では同じクラス名、メソッド名を用いているが違うプログラムコードとなっている。このため、図 9 をコンパイル、実行し、図 10 をコンパイル、実行したあと、図 9 を実行するとおかしなことが起こる (図 10 の StringBufSub が使われるから)。このため、これらを再度実行してみたい場合には、コンパイルからやり直すのがよい。

3.7 デッドロック

廊下や階段の降りたところなど、自分の進行方向から人が来た場合、「どうぞ」と道を譲ろうとすると「どうぞ」と道を譲られることがあるのではないだろうか。このときお互いが「どうぞ」を繰り返していたら、永遠にどちらも進めなくなる。このような状態をデッドロックという。

Java によるマルチスレッドのプログラミングにおいてもこのような状況に陥るプログラムを書いてしまう場合がある。あるスレッド 1 はオブジェクト A にロックをかけたあと、オブジェクト B にロックをするようなコードを実行しているとする。一方でもう 1 つのスレッド 2 はオブジェクト B にロックをかけたあと、オブジェクト A にロックをするようなコードを実行して

いるとする。このとき、微妙な実行のタイミングで、スレッド 1 がオブジェクト A にロックをかけ、スレッド 2 がオブジェクト B にロックをかけたのち、スレッド 1 がオブジェクト B にロックをかけようとし、スレッド 2 もオブジェクト A にロックをかけようとすると、それぞれロックがかかっているため、ロックが解除されるのを待たなければならない。しかし、お互いがお互いの持っているロックの解除を待っているため、ロックの解除は永遠に行われず、プログラムはいつまで経っても進行しない。

このようなバグは再現性が低い場合が多い。すなわち、いつでも必ずデッドロックに陥るのではなく、上述のようにタイミングが合えばデッドロックになるし、タイミングが悪ければ (良ければ??) デッドロックにはならない。

デッドロックを引き起こす可能性のあるプログラムを図 13 に示す。まず最初は図 13 の 3〜8 行目の try-catch 文をコメントアウトし、コンパイルし、実行してみよう。そうすると、デッドロックが起こらず、実行を終了するはずである (これでデッドロックが起こったら幸運なのかも)。次にコメントアウトしていた部分を活かし、もう一度コンパイル、実行してみよう。プロンプトが返ってこなければデッドロックに陥ったということである。不幸にして (??) プロンプトが返ってきた場合、sleep の引数を大きくしてみよう。

さて、このプログラムを簡単に説明しておこう。2〜15 行目の move では、入れ子になった synchronized ブロックを使用している。このメソッドは 2 つのオブジェクトを引数としてとり、最初のオブジェクトを使ってロックをかけ、その後、2 番目のオブジェクトを使ってロックをかけている。main メソッドからは 2 つのスレッドを起動し、それぞれで move(a, b) と move(b, a) を呼び出している。move(a, b) の流れでは a を使ってロックし、その後、b を使ってロックしようとしている。一方で move(b, a) では逆に最初に b でロックをかけ、その後、a を使ってロックしようとしている。これによって上述のようにデッドロックに陥る可能性がある。

実際には最初のスレッドが先に実行を終えてしまってから、もう 1 つのスレッドが実行された場合、デッドロックにならずにプログラムは終了する。このため、sleep メソッドを呼出し、双方のスレッドで、最初のオブジェクトでロックをかけてからある程度の (十分な) 時間が経過した後に、もう 1 つのオブジェクトでロックをかけるようにすると、デッドロックが起こる。

```
 1 public class DeadLockTest {
 2   static public void move(
       StringBuffer toBuf, StringBuffer fromBuf){
 3     synchronized(toBuf){
 4       try {
 5         Thread.sleep(10);
 6       }
 7       catch (InterruptedException e){
 8       }
 9       synchronized(fromBuf){
10         char c = fromBuf.charAt(0);
11         fromBuf.deleteCharAt(0);
12         toBuf.append(c);
13       }
14     }
15   }
16
17   public static void main(String args[]){
18     final StringBuffer a =
             new StringBuffer("Hello");
19     final StringBuffer b =
             new StringBuffer("World");
20
21     Thread x = new Thread(){
22       public void run(){
23         move(a, b);
24       }
25     };
26
27     Thread y = new Thread(){
28       public void run(){
29         move(b, a);
30       }
31     };
32
33     x.start();
34     y.start();
35   }
36 }
```

図 13: DeadLockTest.java

演習 6

図 13 を入力し、コンパイル、実行せよ。デッドロックが起こるといつまでもプログラムが終了しないので、ある程度、待ったのち、Ctrl-C (Ctrl キーを押しながら c のキーを押す) でプログラムを終了させること。

4 待機と通知

マルチスレッドのプログラムを作成する場合、スレッドどうしが強調して作業ができると効率がよくなる場合がある。

```
 1 import java.util.*;
 2
 3 public class WaitTest extends Thread {
 4   private Vector buf;
 5   WaitTest(){
 6     buf = new Vector();
 7   }
 8
 9   public synchronized void add(String str){
10     buf.add(str);
11   }
12
13   public void run(){
14     for(;;){
15       synchronized(this){
16         System.out.println("buf.size() = "
                              + buf.size());
17       }
18     }
19   }
20
21   static public void main(String args[]){
22     WaitTest w = new WaitTest();
23     w.start();
24
25     for(;;){
26       try {
27         Thread.sleep(1000);
28       }
29       catch (InterruptedException e){
30       }
31       w.add("test");
32     }
33   }
34 }
```

図 14: WaitTest.java

図 14 に示したプログラムは 22 行目で Thread オブジェクトを 1 つ作り、23 行目で新たなスレッドを実行している。新たなスレッドの実行主体は 13〜19 行目の run メソッドであり、永遠に buf.size を表示する。

一方、main のスレッドは 1000 ミリ秒ごとに Vector のオブジェクトに要素 (”test”) を追加していっている。

このプログラムを実行すると、buf.size が繰り返し表示されるが、buf.size が変更されるのは 1000 ミリ秒なので 1000 ミリ秒間ずつ同じ内容が画面上を流れていく。

ここでは buf.size に変化があったら次の表示をするようなプログラムを考える。このためには、例えば、前回のサイズを覚えておき、前回と値が変わったら表示をするようにすることも考えられるが、このプログラムでは常にそれを監視するため、CPU を常に使用していることになる。

Java にはスレッドの実行を通知があるまで一時停止 (待機) させる機能が備わっている。この例題プログラムを図 15 に示す。

```
 1 import java.util.*;
 2
 3 public class WaitTest4 extends Thread {
 4   private Vector buf;
 5   WaitTest4(){
 6     buf = new Vector();
 7   }
 8
 9   public synchronized void add(String str){
10     buf.add(str);
11     notify();
12   }
13
14   public void run(){
15     for(;;){
16       synchronized(this){
17         System.out.println("buf.size() = "
                               + buf.size());
18         try {
19           wait();
20         }
21         catch (InterruptedException e){
22         }
23       }
24     }
25   }
26
27   static public void main(String args[]){
28     WaitTest4 w = new WaitTest4();
29     w.start();
30
31     for(;;){
32       try {
33         Thread.sleep(1000);
34       }
35       catch (InterruptedException e){
36       }
37       w.add("test");
38     }
39   }
40 }
```

図 15: WaitTest4.java

このプログラムでは 33 行目の sleep で一定時間経過を待った後、37 行目で w.add が実行される。一方、並行して実行されているもう 1 つのメソッドでは、17 行目で Vector のサイズを表示した後、19 行目で待ちに

入る。w.add は 10 行目で Vector オブジェクトに与えられた文字列を加え、11 行目で notify メソッドを使って (19 行目で) 待っているスレッドに通知を出す。19 行目で止まっていたスレッドは繰り返しの中にあるので、再び 19 行目まで実行してまた待つ。

wait を使ってスレッドを一時停止するにはなんらかのオブジェクトに対し、ロックをかける必要がある。ロックをかけた上で wait メソッドが呼び出されると wait メソッドはそのオブジェクトの待機集合に入れられ、ロックを解放する (そして眠る)。一方、眠っているスレッドに通知を出すにはやはり (wait のときにロックをかけた) オブジェクトに対してロックをかけ、notify メソッドを実行する。この後、notify を実行したスレッドはロックを解放する。notify メソッドによりあるオブジェクトに通知が行われた場合、そのオブジェクトが持つ待機集合の中の 1 つのスレッドが眠りから目覚める。ここで注意すべきことは、待機集合にはいくつものスレッドが入っている可能性があることである。notify によって必ずしも自分が起こされるとは限らない。待機集合に入っている全てのスレッドに対して通知を行うには notifyAll を使う。さて、自分に対して、通知が来た場合、内部ではそのスレッドがそのオブジェクトのロックを再び取ることで、スレッドを再開する。

演習 7

図 14 と図 15 を入力し、それぞれ、コンパイル、実行せよ。

参考文献

[1] 結城浩: *Java* **言語で学ぶデザインパターン入門【マルチスレッド編】**, SOFTBANK Publishing, 2002.

[2] 村上列: *Java* **スレッド完全制覇**, 技術評論社, 2002.