Subject: Report Operating System Lecture Practice Synchronization Java
From: IKENOYA Katsutoshi <j05002@ie.u-ryukyu.ac.jp>
Date: Tue, 13 Feb 2007 21:18:55 +0900
To: Shinji KONO <kono@ie.u-ryukyu.ac.jp>

学籍番号：055702B

・修正点

3.モニタの及ぶ範囲
の図(BadWorker2.jpg)を修正しました。

・開発環境
OS : Mac OS X 10.4.8
コンパイラ :gcc version 4.0.1 (Apple Computer, Inc. build 5250)
java version "1.5.0_06"

・実行環境
同上

１.共有資源と競争状態

・ソースの場所：
/net/home/cvs/y05/j05002/OS/report7/processExample/ThreadRace.java
/net/home/cvs/y05/j05002/OS/report7/processExample/SimpleWorker.java

・実行結果(yield()あり)

```
[j05002@OS-report7]% java processExample/ThreadRace
Thread t1 created.
Thread t2 created.
Thread t3 created.
Thread t1 5
[j05002@OS-report7]%
```

・実行結果(yield()なし)

```
[j05002@OS-report7]% java processExample/ThreadRace
Thread t1 created.
Thread t2 created.
Thread t3 created.
Thread t1 10
[j05002@OS-report7]%
```

yield()がない場合、プロセスの切り替えがなく、
並列に実行されないので結果が１０となっている
と考えられる。

2.相互排除

・ソースの抜粋

ソースの場所：
/net/home/cvs/y05/j05002/OS/report7/processExample/ThreadRace.java
/net/home/cvs/y05/j05002/OS/report7/processExample/SimpleWorker.java

省略

```
public synchronized void work() {
int x;
yield();
x = myResource;
yield();
x = x + 1;
yield();
myResource = x;
yield();
}
```

省略

・実行結果

```
[j05002@OS-report7]% java processExample/ThreadRace
```

```
Thread t1 created.
Thread t2 created.
Thread t3 created.
Thread t1 10
[j05002@OS-report7]%
```

実行結果より、synchronized構文によって正しい
結果が得られることが分かる。

3.モニタの及ぶ範囲

ThreadRaceの初期化を

```
SimpleWorker t1 = new BadWorker("t1",-1,null); // does not run
SimpleWorker t2 = new BadWorker("t2",5,t1);
SimpleWorker t3 = new BadWorker("t3",5,t1);
```

に変更して実行した。
BadWorker t1 である必要がないのは、
BadWorkerがSimpleWorkerを継承しているから。

・ソースの場所：
/net/home/cvs/y05/j05002/OS/report7/processExample/ThreadRace.java
/net/home/cvs/y05/j05002/OS/report7/processExample/BadWorker.java

・実行結果

```
[j05002@OS-report7]% java processExample/ThreadRace
Thread t1 created.
Thread t2 created.
Thread t3 created.
Thread t1 5
```

t1が５となり正しく動作していない。
SimpleWorkerでは、sharedResource.work()として
work()全体を共有資源としているが、
BadWorkerでは、
x = sharedResource.getResource();
や
sharedResource.setResource(x);
のようにxだけを共有資源にしているので、
共有資源を読み込む時(x = sharedResource.getResource())と
共有資源の値を更新する時(sharedResource.setResource(x))以外
はロックがされないので、他のスレッドによって上書きされてしまい
このような結果になったと考えられる。
(添付ファイル：BadWorker3.jpg)

4.生産者消費者問題

buffer fullになるように、バッファの大きさを１にし、
生産するデータの個数を３０として実行してみた。

・ソースの場所：
/net/home/cvs/y05/j05002/OS/report7/processExample/ThreadProdCons.java
/net/home/cvs/y05/j05002/OS/report7/processExample/OurThreadMonitor.java
/net/home/cvs/y05/j05002/OS/report7/processExample/OurProducer.java
/net/home/cvs/y05/j05002/OS/report7/processExample/OurConsumer.java

・実行結果

```
[j05002@OS-report7]% java processExample/ThreadProdCons
Consumer c1 eats 30
Consumer c1 eats 29
Consumer c1 eats 28
Consumer c1 eats 27
Consumer c1 eats 26
Consumer c1 eats 25
Consumer c1 eats 24
Consumer c1 eats 23
Consumer c1 eats 22
Consumer c1 eats 21
Consumer c1 eats 20
Consumer c1 eats 19
Consumer c1 eats 18
Consumer c1 eats 17
Consumer c1 eats 16
Consumer c1 eats 15
Consumer c1 eats 14
```

```
Consumer c1 eats 13
Consumer c1 eats 12
Consumer c1 eats 11
Consumer c1 eats 10
Consumer c1 eats 9
Consumer c1 eats 8
Consumer c1 eats 7
Consumer c1 eats 6
Consumer c1 eats 5
Consumer c1 eats 4
Consumer c1 eats 3
Consumer c1 eats 2
Consumer c1 eats 1
Consumer c1 eats 0
[j05002@OS-report7]%
```

insert/remove ともにブロックされなかった。
Java API documentの
java.lang.Object
のwaitメソッドを見てみると、
wait()は、他のスレッドがこのオブジェクトの notify() メソッドまたは
notifyAll() メソッドを呼び出すまで、現在のスレッドを待機させる。
とあった。
つまり、insert/remove ともにブロックされなかったのは、
OurThreadMonitorでバッファが足りなくなった場合はwait()
を用いて待機させて、バッファに空きができてから再開させて
いるからである。

また、消費者がデータを取り損なうことなく、終了するために
生産者スレッドが終了した後、join()を使用して100msだけ
消費者を殺すのを待つようにした。

5.複数の生産者と複数の消費者

生産者、消費者の数がそれぞれ 2 の場合と
生産者の数が 2 、消費者の数が 3 の場合で試してみた。

・ソースの場所：
/net/home/cvs/y05/j05002/OS/report7/processExample/ThreadProdCons.java
/net/home/cvs/y05/j05002/OS/report7/processExample/OurThreadMonitor.java
/net/home/cvs/y05/j05002/OS/report7/processExample/OurProducer.java
/net/home/cvs/y05/j05002/OS/report7/processExample/OurConsumer.java

(1)生産者、消費者の数がそれぞれ 2

・実行結果

```
[j05002@OS-report7]% java processExample/ThreadProdCons
Consumer c1 eats 10
Consumer c1 eats 9
Consumer c1 eats 8
Consumer c1 eats 7
Consumer c1 eats 6
Consumer c1 eats 5
Consumer c1 eats 4
Consumer c1 eats 3
Consumer c1 eats 2
Consumer c1 eats 1
Consumer c1 eats 0
Consumer c2 eats 10
Consumer c2 eats 9
Consumer c2 eats 8
Consumer c2 eats 7
Consumer c2 eats 6
Consumer c2 eats 5
Consumer c2 eats 4
Consumer c2 eats 3
Consumer c2 eats 2
Consumer c2 eats 1
Consumer c2 eats 0
[j05002@OS-report7]%
```

(2)生産者の数が 2 、消費者の数が 3

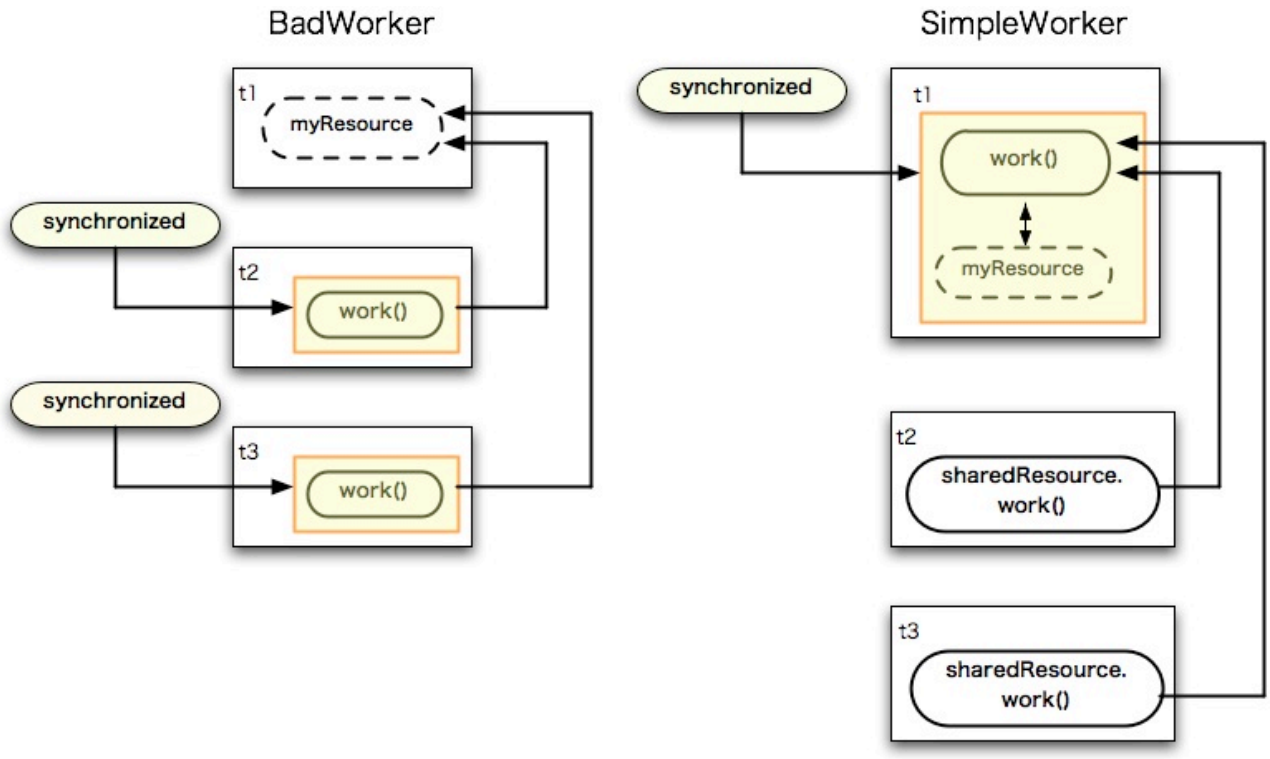
デッドロックの検出が行えるように

```
/*set Timer*/
long timeout = 1000;//1000ms
long repray = 100;//10ms
TimeOut tOut = new TimeOut();
```

```
Timer timer = new Timer();
timer.schedule(tOut,timeout,repray);
```
として 想定した時間以上たってもプログラム
が完了しない場合にタイムアウトするようにした。

・実行結果

```
[j05002@OS-report7]% java processExample/ThreadProdCons
Consumer c1 eats 10
Consumer c1 eats 9
Consumer c1 eats 8
Consumer c1 eats 7
Consumer c1 eats 6
Consumer c1 eats 5
Consumer c1 eats 4
Consumer c1 eats 3
Consumer c1 eats 2
Consumer c1 eats 1
Consumer c1 eats 0
Consumer c2 eats 10
Consumer c2 eats 9
Consumer c2 eats 8
Consumer c2 eats 7
Consumer c2 eats 6
Consumer c2 eats 5
Consumer c2 eats 4
Consumer c2 eats 3
Consumer c2 eats 2
Consumer c2 eats 1
Consumer c2 eats 0
Dead Lock and TimeOut
[j05002@OS-report7]%
```



BadWorker3.jpg	Content-Type: image/jpeg Content-Encoding: base64