Introduction to Process Synchronization Using the *Java* Language

Harry J. Foxwell, *Java* Technologist
Sun Microsystems Computer Corporation
harry.foxwell@east.sun.com

August 1997

Abstract:

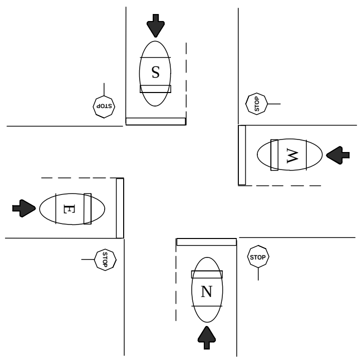
For many types of computer applications, more than one process or program must share access to a common data structure. This concurrent access requirement gives rise to several classic problems: how to avoid *deadlocks* and *race conditions*. These problems have been extensively studied, and numerous solutions have been proposed.

This paper introduces the concepts necessary for the beginning ***Java*** programmer to understand *process synchronization*, and discusses how the ***Java*** language allows programmers to write applications that avoid synchronization problems.

Introduction - The Problems

Deadlock

Every workday, Neil, Sue, Ed, and Wayne drive to their respective offices. They travel through the intersection of Oak Street and Maple Avenue, where there are "4-way" stop signs:



Each of these drivers strictly obeys the traffic rule that says if two cars travelling in perpendicular directions arrive at the intersection at exactly the same time, then the driver on the left yields to the driver on the right. So, if Neil and Ed happen to arrive at the intersection simultaneously, Ed must yield to Neil.

Several days ago, Neil, Sue, Ed, and Wayne happened to arrive at this intersection at exactly the same moment. Each of them, following their interpretation of the traffic rule, started waiting for the driver on the right to move through the intersection. They are still waiting.

This unfortunate situation is known as *deadlock*, and it occurs frequently, although briefly, in many real-life situations. When it happens with people, someone usually grows impatient and "breaks the rule", allowing activity to continue. Computers, on the other hand, don't grow impatient no matter how long they wait, and they have to be told explicitly what rules to follow. The rules must prevent deadlock, or the computer will stop working.

For example, suppose one computer program has "locked" FILE.1 for exclusive use, and another program has locked FILE.2. If the first program also decides that it needs FILE.2, while at the same time the second program decides that it needs FILE.1, deadlock will occur if there are no explicit rules for locking and releasing files. Both programs, and perhaps the computer itself, will stop executing instructions.

Race Conditions

George and Martha have a joint account containing $100 at Maxwell Bank. They each have an ATM (Automatic Teller Machine) card that allows them to check their account balance and to withdraw funds. One day, both George and Martha each decided that they needed $75. George conducted the following ATM transactions at 1:35PM:

```
George: Ask for Balance
  ATM1: Balance is $100
George: Withdraw $75
  ATM1: OK (delivers money)
  ATM1: Balance is $25
```

At 1:48PM, Martha, at a different location, conducted this ATM transaction:

```
Martha: Ask for Balance
  ATM2: Balance is $25
```

Martha then assumes that George withdrew money from the account.

However, suppose George and Martha both arrive at two different ATMs at 1:35PM. They start their transactions:

```
George: Ask for Balance
  ATM1: Balance is $100
Martha: Ask for Balance
  ATM2: Balance is $100
George: Withdraw $75
  ATM1: OK (delivers money)
Martha: Withdraw $75
  ATM2: Insufficient Funds!
```

At this point, Martha is puzzled because the ATM told her there was $100 in the account and she only asked for $75! This resulted from a *race condition*, where the transaction component that executed first did not block other transactions until the first set completed. Even worse, if the ATM software does not prevent certain race conditions, both George *and* Martha could each walk away with $75! Good for them, bad for the bank.

Fortunately, real ATM software is designed to prevent these race condition problems. How this is accomplished is explained in the **Solutions** section of this paper.

Producing and Consuming

In a now classic TV comedy routine, Lucille Ball removes completed cakes from an automated bakery, places them in boxes and stacks them for shipment. She occasionally pauses to lick some icing off a cake, causing her to fall behind the continuous flow of cakes. Eventually, even when she tries to work faster, the machine gets the better of her, and cakes start falling to the floor.

The automatic bakery is an example of a "producer", and Lucy is a "consumer"; the desired solution is for the producer to stop item production until the consumer catches up, then to resume production. Every item produced must be consumed.

For computers, the solution to this problem is critical. Data sent from one process or system must arrive complete at another process or system; no data can be lost. Copying files from one location to another, sending email, and establishing two-way communications via modems or networks all require synchronization of producer and consumer processes. In most cases these processes work at disparate speeds, requiring one process to pause while the other continues.

Some Classic Computer Science Problems

The requirements of producer and consumer processes, resource allocation, and the need to avoid deadlocks and race conditions, are highlighted by several classic problems in computer science.

In 1965, E. Dijkstra proposed the "Dining Philosophers" problem as a metaphor for process synchronization**[1]**. In this problem, five philosophers sit around a circular table either thinking or eating. Each philosopher requires two forks to eat and there is a single fork between each philosopher. When hungry, a philosopher tries to take a fork, waits to take a second fork, then eats and puts down the forks. The challenge is to design a set of rules and signals that prevent a deadlock, which would occur if each philosopher picked up his right fork at the same time, then patiently waited forever for a left fork. This problem is relevant to computer systems with multiple processors (philosophers) that need shared resources (forks) to run programs (food).

Another class of synchronization problems concerns multiple processes that write to a common data area while additional processes read from that same data area. The "Readers and Writers" problems, described by P. Courtois in 1971**[2]**, requires a solution that prevents race conditions; reader processes must wait for writer processes to start and complete.

Some synchronization problems involve waiting in queues, such as the "Sleeping Barber Problem"**[3]**. One or more barbers sleep in their shop until customers arrive and wake them for a haircut. If several customers arrive, they sleep in a queue until a barber is ready. Rules are required to prevent everyone in the shop from falling asleep. In computer systems, processes waiting for other processes to complete will "sleep" until the events they are awaiting wake them up, such as the completion of I/O operations. Procedures are required that will avoid sleeping processes waiting forever to wake each other up, or processes missing wakeup events entirely.

Solutions

Some Classic Solutions

Deadlocks and race conditions in computer programs are extremely difficult to reproduce and debug, because their occurrence depends on the precise timing and order of sets of instructions. Solutions were therefore devised that were "provably correct" in the sense of a mathematical algorithm or theorem.

One requirement of these solutions concerns access to shared data: when one process is modifying that data, other processes must be excluded from accessing that data. The section of a process that accesses shared data is called its "critical section", and all processes must follow the rule of "mutual exclusion": only one process may be executing instructions in its critical section at a given instant.

In order to both test whether another process was in its critical section and exclude other processes when necessary, an "atomic operation" is required. Such an operation looks like a single instruction to the programmer, and is guaranteed by the hardware and operating system to complete without interruption. The "test-and-set-lock" instruction is an example of this. A process tests whether a lock is set on its critical section. If so, it continues to test until the lock is removed, then it sets the lock itself, enters and completes its own critical section, and unsets the lock. While this procedure works, it requires that all processes waiting to enter their critical sections continuously run to test the lock, a waste of CPU resources. This continuous testing is called "busy waiting". There is a better solution known as "sleep and wakeup".

Dijkstra proposed the use of "semaphores" to solve the mutual exclusion problem. A semaphore is a shared data item used to indicate permission for a process to enter its critical section, along with atomic operations for changing the semaphore. Additionally, processes waiting to enter their critical sections would "block" or "sleep" instead of continuously testing a lock. The blocking process must wake up the sleeping process that was waiting for the semaphore to change. This solves the process synchronization problem, and it is widely used in multiprocess programming. However, correctly programming sleep and wakeup semaphore actions to prevent deadlock can be difficult, so additional solutions have been proposed.

In 1974 C.Hoare**[4]** described the use of "monitor" programs which simplify the programmers task of managing the critical sections of multiple processes. Implemented in the operating system itself, a monitor guarantees that only one process may be executing its critical section; other processes controlled by the monitor are suspended.

Current Solutions

As both computer hardware and operating systems continued to evolve, simpler yet more powerful concepts were needed to implement multiple concurrent processes running on multiprocessor systems. Starting in the late 1980s, operating systems such as **Mach**, **OS/2**, and updated versions of **UNIX** implemented fine-grained control of sequences of instructions called "threads".

A thread is the smallest sequence of instructions schedulable as a unit by the operating system**[5]**. Threads in the same process share the same address space and global variables. Each thread, however, has its own program counter, stack, register contents, and state. Each thread can execute independently and concurrently, although threads that share access to common data should be synchronized. Modern programming languages and operating systems provide access to threads and the means to control and synchronize them. Critical sections of programs may be easily implemented as independent, mutually exclusive threads.

The use of multi-threaded programming techniques can significantly improve system performance, improve interprocess communications, and simplify the task of writing multiprocess programs. Certain data processing tasks are inherently multi-threaded, for example a distributed file system server that must handle multiple simultaneous client requests, or a complex, multi-user database program. Certain mathematical computations, such as array or matrix calculations, may also benefit by organization into groups of independent threads. In fact, some **FORTRAN** compilers can recognize independent computation loops and produce multi-threaded executables.

Multi-Threaded Programming in Java

Support for the Thread class of objects is one of the *Java* language's most important features. Threads may be used in both application programs and in *applets*, which are programs embedded in a Web page and executed by the *Java* interpreter built into the browser that is displaying the page. The *Java* interpreter has a thread "scheduler" that manages all the threads of a program and decides which ones are to be run.

Programs that need to perform several independent or cooperating tasks should be organized into threads, some of which may need to be synchronized.

The *Java* programmer can create and execute multiple threads in a program, and can explicitly control these threads with methods that are defined in the *java.lang* package. Most of the interesting and useful *Java* applications and applets *require* the use of threads, such as those with concurrent animation and audio, or applications that share access to data.

To create a thread in *Java*, the programmer defines a subclass of the Thread class:

```
public NuThread extends Thread
   {
   public void run();
      {
      // do stuff...
      }
   }
```

Then an *instance* of that thread must be created with the *new* operator and executed with the *start()* method:

```
...
NuThread nt1 = new NuThread();
nt1.start()
...
```

The *start()* method calls the *run()* method to execute the thread.

The programmer can now manage this thread with any of the following methods:

```
nt1.sleep(N);   // sleep for N msec
nt1.stop();     // terminate & cleanup
nt1.suspend();  // suspend execution
nt1.resume();   // resume execution
```

A thread's *run()* method may be declared to be *synchronized*. The *Java* interpreter guarantees that only one synchronized method may modify an object at a given instant by setting a lock on that object; no other thread may modify the object until the lock is released.

In the example above, NuThread may be defined as a synchronized thread as follows:

```
public NuThread extends Thread
   {
   public void synchronized run();
      {
      // do stuff...
      }
   }
```

Synchronized threads must inform each other when one thread is finished and another waiting thread may begin. Threads can be told to wait until notified by another thread, or to wait until a given time has expired:

```
wait();    // wait until notified

wait(N);   // wait until notified
          //  or N msec
```

A running thread may inform a single waiting thread using the *notify()* method, or may notify all waiting threads using the *notifyAll()* method.

A Simple Example

Suppose you want one *Java* thread to send messages to another *Java* thread. That is, you want to illustrate the Producer-Consumer Problem using *Java***[7]**. Listing **PC1.java** (Appendix) defines a shared variable that receives messages from the *put()* method, and is then read by the *get()* method. The **Produce_Consume** class creates and starts the Producer thread and the Consumer thread, but in program **PC1.java** these threads are *not* synchronized. This means that the two threads pay no attention to each other when writing or reading the shared message variable, with *incorrect* results:

```
Producer put: 343
Consumer got: 343
Consumer got: 343
Consumer got: 343
Consumer got: 343
Consumer got: 343
Consumer got: 343
Consumer got: 343
Consumer got: 343
Consumer got: 343
Consumer got: 343
Producer put: 118
Producer put: 273
Producer put: 847
Producer put: 716
Producer put: 379
Producer put: 990
Producer put: 607
Producer put: 0
Producer put: 192
```

The Consumer thread read the first message and kept running, completing before the Producer even sent its second message! What is needed is for the Consumer to *synchronize* with the Producer, so that all messages get consumed. This is accomplished by declaring that the *put()* and *get()* methods are *synchronized* threads (see listing **PC2.java** in the Appendix), that they each *wait()* while the other thread has locked access to the shared variable, and *notify()* the waiting thread when each of them are finished with their task. When **PC2** is run, the Producer thread writes a message then waits for the Consumer thread to read it. While the Producer is writing, the Consumer is blocked from running. When the Producer notifies the Consumer that it is finished, the Consumer thread runs and blocks the Producer. This produces the desired alternation of producing and consuming:

```
Producer put: 997
Consumer got: 997
Producer put: 531
Consumer got: 531
Producer put: 204
Consumer got: 204
Producer put: 2
Consumer got: 2
Producer put: 798
Consumer got: 798
Producer put: 850
Consumer got: 850
Producer put: 83
Consumer got: 83
Producer put: 955
Consumer got: 955
Producer put: 181
Consumer got: 181
Producer put: 257
Consumer got: 257
```

Every message produced gets consumed; no messages are "lost".

The **Shared_Buffer** class in this example implements a *monitor* for the synchronized *put()* and *get()* methods, guaranteeing that only one thread (critical section) may execute and access the shared variable **msg**, ensuring mutual exclusion.

Conclusion

Beginning *Java* programmers must understand several important concepts beyond syntax, control flow, and variable declaration: the *Object Oriented* nature of the language, and the *control and synchronization of threads*. *Java* was designed to simplify the writing of multi-threaded programs. Additional examples of such programs may be found in the tutorial books and Web pages mentioned in the Bibliography.

References

[1] Tannenbaum, pg. 56.
[2] Tannenbaum, pg. 58.
[3] Stallings, pg. 191.
[4] Stallings, pg. 198.
[5] Berg & Lewis, pg. 8.
[6] Flanagan, pg. 161.
[7] Student Guide, pg. 4.16.

Bibliography

Berg, D., Advanced Techniques for Java Developers, John Wiley & Sons, 1997

Berg, D. and Lewis, **B.**, Threads Primer, SunSoft Press, Mountain View CA, 1996

Flanagan, D., Java in a Nutshell, 2nd Ed., O'Reilly & Associates, 1997

Lea, D., Concurrent Programming in Java, 2nd Ed., Addison-Wesley, 1997

Oaks, S., Wong, H., Java Threads, O'Reilly & Associates, 1997

Stallings, W., Operating Systems, 2nd Edition, Prentice Hall, Englewood Cliffs NJ, 1995

Student Guide, Advanced Java Programming, Sun Education Services, Mountain View CA, 1995

Tannenbaum, A., Modern Operating Systems, Prentice Hall, Englewood Cliffs NJ, 1992

Suggested Reading

Campione, M., The Java Language Tutorial, 2nd Ed., Addison-Wesley, 1997

Lemay, L., Teach Yourself Java in 21 Days, 2nd Ed., Sams.net Publishing, Indianapolis IN, 1997

Suggested WWW Browsing

http://java.sun.com
http://www.gamelan.com

Trademarks

Appendix

```
// PC1.java
//
// Simple example of UNsynchronized producer and consumer
// threads...producer thread writes messages (random
// numbers between 0 and 1000) into a shared variable (m),
// while the consumer thread reads the messages.
//
// For simplicity, all classes are defined in this one
// source file.  To compile the program, execute
// 'javac PC1.java'.  This will produce 4 class files:
//
//     Produce_Consume.class    - the main program
//     Consumer.class           - the consumer thread
//     Producer.class           - the producer thread
//     SharedBuffer.class       - the "monitor", unsynchronized
//
// To run the program, enter 'java Produce_Consume'.
// This will display the messages as they are sent
// and received by each thread.
//

class Produce_Consume
  {
  public static void main(String args[])
    {
    SharedBuffer m = new SharedBuffer();
    Producer prod = new Producer(m);
    Consumer cons = new Consumer(m);

    prod.start();
    cons.start();
    }
  }

class SharedBuffer
  {
  private int msg;
  public int get()
    {
    return msg;
    }

  public void put(int message)
    {
    msg = message;
    }
  }

class Producer extends Thread
  {
  private SharedBuffer shbuf;
  public Producer(SharedBuffer m)
    {
    shbuf = m;
    }

  public void run()
    {
    int j;
    for (int i = 0; i < 10; i++)
      {
      j = (int)(Math.random() * 1000);
      shbuf.put(j);
      System.out.println("Producer put: " + j);
      try
        {
        sleep((int)(Math.random() * 100));
        }
      catch (InterruptedException e) { }
      }
    }
  }

class Consumer extends Thread
  {
  private SharedBuffer shbuf;

  public Consumer(SharedBuffer m)
    {
    shbuf = m;
    }

  public void run()
    {
    int message = 0;
    for (int i = 0; i < 10; i++)
      {
      message = shbuf.get();
      System.out.println("Consumer got: " + message);
      }
    }
  }
```

Appendix

```
// PC2.java
//
// Simple example of synchronized producer and consumer
// threads...producer thread writes messages (random
// numbers between 0 and 1000) into a shared variable (m),
// while the consumer thread reads the messages.
//
// For simplicity, all classes are defined in this one
// source file.  To compile the program, execute
// 'javac PC2.java'.  This will produce 4 class files:
//
//     Produce_Consume.class    - the main program
//     Consumer.class           - the consumer thread
//     Producer.class           - the producer thread
//     SharedBuffer.class       - the "monitor", synchronized
//
// To run the program, enter 'java Produce_Consume'.
// This will display the messages as they are sent
// and received by each thread.
//

class Produce_Consume
  {
  public static void main(String args[])
    {
    SharedBuffer m = new SharedBuffer();
    Producer prod = new Producer(m);
    Consumer cons = new Consumer(m);

    prod.start();
    cons.start();
    }
  }

class SharedBuffer
  {
  private int msg;
  private boolean locked = true;

  public synchronized int get()
    {
    while (locked == true)
      {
      try
        {
        wait();
        }
      catch (InterruptedException e) { }
      }
    locked = true;
    notify();
    return msg;
    }

  public synchronized void put(int message)
    {
    while (locked == false)
      {
      try
        {
        wait();
        }
      catch (InterruptedException e) { }
      }
    msg = message;
    locked = false;
    notify();
    }
  }

class Producer extends Thread
  {
  private SharedBuffer shbuf;

  public Producer(SharedBuffer m)
    {
    shbuf = m;
    }

  public void run()
    {
    int j;
    for (int i = 0; i < 10; i++)
      {
      j = (int)(Math.random() * 1000);
      shbuf.put(j);
      System.out.println("Producer put: " + j);
      try
        {
        sleep((int)(Math.random() * 100));
        }
      catch (InterruptedException e) { }
      }
    }
  }

class Consumer extends Thread
  {
  private SharedBuffer shbuf;

  public Consumer(SharedBuffer m)
    {
    shbuf = m;
    }
```

```
public void run()
  {
  int message = 0;
  for (int i = 0; i < 10; i++)
    {
    message = shbuf.get();
    System.out.println("Consumer got: " + message);
    }
  }
}
```