Designing a Method for Synchronizing The Producer-Consumer in Java

M. Al-Shuraifi
Department of Computer Science, University of Babylon

Abstract

The producer-consumer term refers to a programming concept in which the producer computes some value and then place it in a shared object. The consumer reads the produced value and does something with it. How do we make sure that the consumer doesn't consume the value until it has been produced? Or, what happens if the producer and consumer work simultaneously? As synchronizing them could be done inside operating systems, still difficult using programming languages. This paper puts the mechanism for synchronizing both the producer and consumer in a programming language taking in account that the pc used has only one cpu. Java has been used to get rid of that difficulty and synchronize them as it is considered as the first concurrent object oriented language COOP.

المقدمة

مصطلح (المنتج-المستهلك) يشير إلى مفهوم برمجي والذي فيه يقوم المنتج بحساب قيمة ما ومن ثم وضعها في كيان مشترك. المستهلك يقرأ القيمة المنتجة من ذلك الكيان المشترك ومن ثم الأستفادة منها في أي عمل كان. كيف نتأكد بأن المستهلك لا يستهلك القيمة حتى تكون منتجة؟ أو ماذا يحدث أذا عمل كل من المنتج والمستهلك معا في وقت واحد(بصورة متزامنة)؟ نظرا لأن تنفيذ عملية التزامن باستخدام نظم التشغيل أصبح أمرا اعتياديا، لازال أنجازها يعتبر صعبا باستخدام لغات البرمجة. هذا البحث يضع الميكانيكية لكيفية تزامن كل من المنتج والمستهلك في لغة برمجية آخذين بالحسبان أستعمال معالج مركزي واحد في الحاسوب. جافا قد أستعملت للتغلب على تلك الصعوبة وبرمجة تزامنهما لأنها تعتبر أول لغة برمجية موجهة كيانية تزامنية COOP.

Introduction

Implementing two processes or more at the same time is called a concurrent processing. The process, here, means a running program. For example, a large file could be downloading from the Web, while someone is typing a letter on the same computer and at the same time. However, people who were using desktop computers in the 1980s don't take this for granted! It can be remembered the days of having to wait for documents to be printed before it could be getting on with anything else [Charatan and Kans, 2002]. Most operating systems today provide some support for doing several things at once, but support in popular languages was limited or nonexistent prior to java.

At first sight it does seem rather extraordinary that a computer with only one central processing unit (CPU) can perform more than one task at any one time. The way it achieves this is by some form of time-slicing; in other words it does a little bit of one task, then a little bit of the next and so on – and it does this so quickly it appears that it is all happening at the same time. Each separate task performed by a single program in known as a thread. Then the interpreter does all the work of making these threads execute not simultaneously but concurrently (i.e. running in an interwoven fashion, overlapped in time. The concurrent execution of these threads can be made to appear as if it were simultaneous execution.) [Arnow and Weiss, 1998]

Going back to the problem this paper discusses and tries to solve it. To exemplify the situation, let's consider the following pseudo code implementing a concurrent buffer:

```
Class Buffer{
…
void put(object obj) { if ("buffer not full")…}
object get() { if ("buffer not empty")…}
 }
```

That was one of the main solutions considered in the past to get rid of the problem. We see, it must make sure that no object is removed from an empty buffer and that no object is inserted into a full buffer. The problem seems to have been solved, but the fact is not. In a sequential setting, the burden of ensuring such constraints resides with the buffer's user. Indeed the buffer is created and used by one thread only, which is responsible for the state of the object. To facilitate usage, the buffer's methods might return certain error codes in case of misuse. This approach is not feasible in a concurrent setting. The buffer will be used concurrently by multiple clients, leaving each of them no idea on the buffer's current state. The burden of enforcing the synchronization constraints must ultimately lie with the buffer itself. Unfortunately, mixing behavioral and synchronization code in class definitions represents an obstacle to code inheritance in programming languages until coming of java which made that possible.

This paper suggests that the producer signals when there is a new value to consume and that the consumer waits until signaled by the producer. Furthermore, the producer has to wait until the consumer has read the preceding value before writing in a new value. This solution needs the synchronization feature which represents a fundamental part in java as it is the first concurrency object oriented language COOL. But before starting to talk about the proposed method, it is necessary to know how the Producer-Consumer works in C++ and compare it with the proposed method.

Producer-Consumer in C++

Writing the synchronization code for a multi-threaded application traditionally has been both difficult (due to the predominance of low-level APIs) and non-portable.

The portability problem arises because neither C nor C++ provide a standard class library for synchronization. As a result, many operating systems provide their own, proprietary APIs for synchronization. Many companies and individuals have successfully tackled this problem by writing (and porting) a *portability layer* that hides the proprietary APIs of the underlying operating system. Although these portability-layer libraries solve the portability problem, they also tend to provide a low-level API. As such, they do *not* simplify the writing of synchronization code. [McHale, 2003]

The *producer-consumer* policy is a part of synchronization problem in C++. , the put-style and get-style operations execute in mutual exclusion; this is to prevent the buffer from becoming corrupted due to concurrent access. This policy can be denoted as follows:

ProdCons[PutOp, GetOp, OtherOp]

OtherOp denotes any other (non put-style and non get-style) operations on the buffer class. For example, perhaps there is an operation on the buffer that returns a count of how many items are currently in the buffer. Such an operation might need to run in mutual exclusion with the put-style and get-style operations to ensure its correct operation. Consider the following (pseudocode) class:

```
class WidgetBuffer {
public:
        ... // constructor and destructor
        void insert(Widget * item);
        Widget * remove();
};
```

The ProdCons policy can be instantiated upon this class as follows:

ProdCons[{insert}, {remove}, {}]

Notice that the WidgetBuffer class has *only* put-style and get-style operations. Because of this, the OtherOp parameter of the policy is instantiated upon an *empty* set of operations names.
The subclass below introduces a new operation, called count():

```
class EnhancedWidgetBuffer: pubic WidgetBuffer {
 public:
        ... // constructor and destructor
        int count();
};
```

The ProdCons policy can be instantiated upon this subclass as follows:

ProdCons[{insert}, {remove}, {count}]

A common variation of the producer-consumer policy is the *bounded* producer-consumer policy. In this case, the buffer has a fixed size. This prevents the buffer from growing infinitely large if one thread puts items into the buffer faster than the other thread can get them. In this policy, if the producer thread tries to put an item into an already-full buffer then it will be blocked until the buffer is non-full. This policy is denoted as follows:

BoundedProdCons(int size)[PutOp, GetOp, OtherOp]

Notice that the size of the buffer is specified as a parameter to the name of the policy. Such parameters are usually instantiated upon a corresponding parameter to the constructor of the buffer. For example, consider the following (pseudo-code) class:

```
class BoundedWidgetBuffer {
 public:
        // constructor and destructor
        BoundedWidgetBuffer(int buf size);
        ˜BoundedWidgetBuffer();
        void insert(Widget * item);
        Widget * remove();
};
```

The BoundedProdCons policy can be instantiated upon this class as follows:

BoundedProdCons(buf_size)[{insert}, {remove},{}]

Threads

In Operating systems terms, a program that is running is known as a process. An operating system can be running several processes for different users at any one time. Not all of these need be active: they could be awaiting their share of processor time, or they could be waiting for some information, such as user input.[Bishop, 1997]

Now, within a single process, the division into separately runnable subprocesses can be made. In Java these are known as threads and a program with threads is called multi-threaded. Each thread looks like it is running on its own. It can communicate with other threads in the same process, though care must be taken when this is done through changing the value of shared variables.

A thread runs independently of anything else happening in the computer. Without threads an entire program can be held up by one cpu intensive task or one infinite loop, intentional or otherwise. With threads the other tasks that don't get stuck in the loop can continue processing without waiting for the stuck task to finish. [Harold, 1997]

In the same way that the operating system shares time between processes, so it must share time among threads. The fairest way to share is to give each thread a time slice, at the end of which it is suspended and the next thread that is ready to run is given a chance. A less attractive method is for a thread to run until it needs information from elsewhere (another thread, or the user) and only then to relinquish control of the processor. The problem with this approach is that a single thread can hog the processor. By now, most systems are using the first approach.

Why use threads?

A simple answer is: Java applications that use threads are able to perform multiple tasks at the same time. For example, a Java program may need to update a graphic on the screen while at the same time accessing the network. Java threads also let us program the way humans normally think. People are constantly performing multiple tasks at any given time. Since people act in a concurrent world, it is much easier to develop programs that behave like the real world.[Berg and Fritzinger, 1998]

Synchronization

Since a thread can change variables in a process that other threads can use, we can see that we need some way of communicating information between threads. Another way to look at this is a way to synchronize access to common or shared information. Thread synchronization provides a mechanism prevents one thread from changing a variable that another thread may be using. Most thread synchronization is controlled through the use of function or method calls where the function uses some sort of control to prevent other threads from accessing the information.[Berg and Fritzinger, 1998]

Then the objective of synchronization is to ensure that, when several threads want access to a single resource, only one thread can access it at any given time. Figure 1 shows this logic. [Horton, 2001]

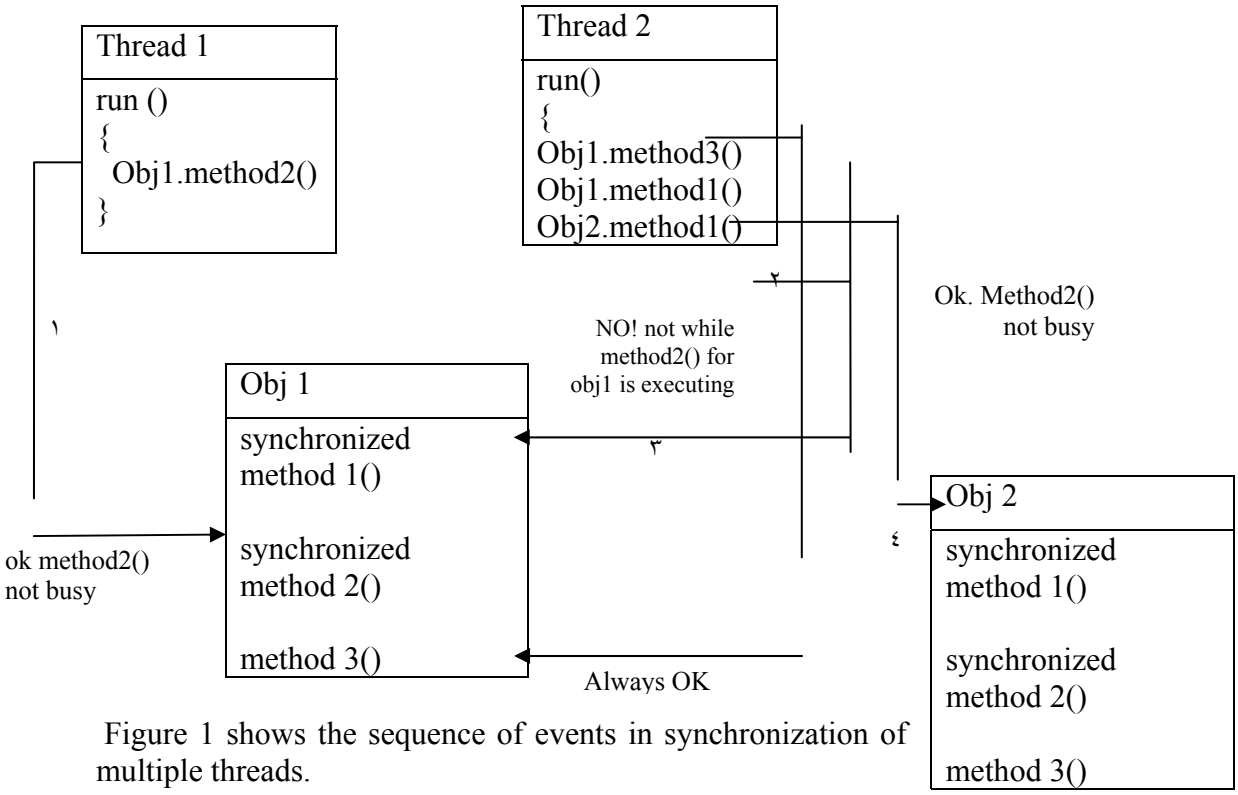

Figure 1 shows the sequence of events in synchronization of multiple threads.

The proposed method

If there are two threads, one called the producer and the other called the consumer. The producer computes some value, that is, "produces it," and then places it in a shared object. The consumer reads the produced value and does something with it, that is, "consumes it." We want this activity to repeat until some terminating condition is reached, and we want the producer and consumer, whenever possible, to execute simultaneously. How do we make sure that the consumer doesn't consume the value until it has been produced? The answer is that the producer signals when there is a new value to consume and that the consumer waits until signaled by the producer. Furthermore, the producer has to wait until the consumer has read the preceding value before writing in a new one. Thus the producer has to wait for the consumer to signal that it has read the new value. This problem is called the *producer-consumer problem*, and it models many real-world computer applications.

Signal-wait synchronization is accomplished with two methods inherited from *object*: *wait()* and *notify()*. When a thread is executing inside a synchronized method for some object, supposed to be *obj*, any other thread that attempts to call any synchronized method for *obj* is blocked. It doesn't need to be the same method for blocking to occur. Synchronization actually takes place at the object, not the thread. Ac it executes, a thread of execution "moves" from object to object by calling methods in different objects.

When *wait()* is called from within a synchronized method, the thread is suspended and the lock for the object is released. That is, other threads are now allowed to make calls to synchronized methods for the object.

When *notify()* is called from within a synchronized method, one thread, if there is one, that is suspended as a result of a *wait()* call while executing in the same object, resumes. This thread won't actually begin execution until the thread calling *notify()*

has returned from its synchronized method and released the lock. The following diagram shows the states that a thread can be in as a result of synchronization. Note that a thread can be in one of three states: It can be waiting for the lock; it can be waiting for a *notify()* from another thread; or it can be running, in which case it is holding the lock.

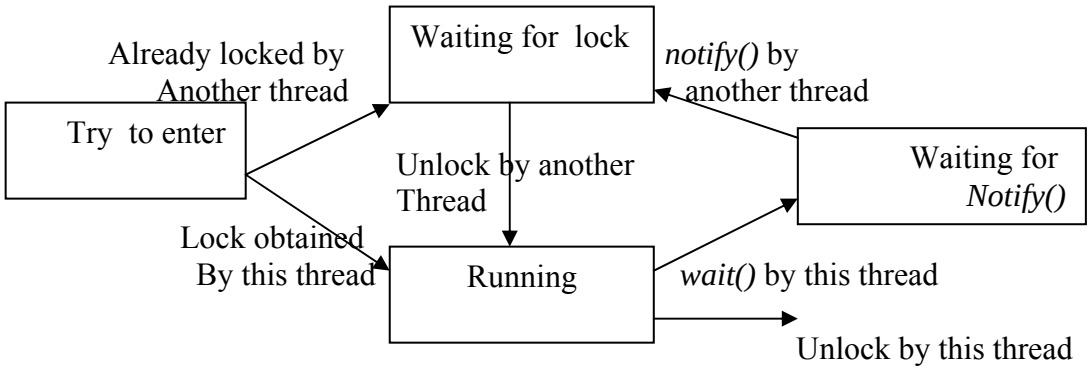

Figure (2) shows the use of "notify()" and "wait()" to synchronize the producer and consumer.

```
Class Producer extends Thread {
Producer(Buffer buf) {
Buffer = buf
}
Public void run(){
For(int i=0; i<10;i++){
Buffer.put(i);
}
}
Private Buffer buffer;
}

Class consumer extends Thread {
Consumer(Buffer buf) {
Buffer=buf;
}
Public void run(){
For(int i=0;i<10;i++){
Int value = buffer.get();
}
}
Private Buffer buffer;
}
```

All synchronization is done in the *Buffer* object. If the buffer wasn't synchronized, the producer and consumer would just execute as fast as they could. The consumer might get the same value several times from the buffer, or the producer might try to put a new value in the buffer before the consumer read the old one.

```
public class Buffer {
private Boolean empty=true;
private int value;
public synchronized int get()
throws Exception {
while (empty) { wait(); }
empty=true;
;
Notify();
Return value;
}
public synchronized void put(int newValue)
throws Exception {
while (!empty) { wait(); }
value=newValue;
empty=false;
notify();
}
}
```

In the class Buffer, it could be seen that both *put()* and *get()* are synchronized. Thus, if one thread is executing *put()*, the other thread will be blocked if it tries to execute *put()* or *get()* for the same object. The mutual exclusion implied by *synchronized* applies to all methods for a single object, not just simultaneous calls to the same method.

Also, the boolean field *empty* has been used to keep track of whether the buffer contains a produced item that hasn't yet been consumed or whether the buffer is empty. The buffer is initially empty, with *empty* set to *true*. It is repeatedly checked whether the buffer is empty. If it is, the loop terminates and proceed to put the new value in *value*, set *empty* to *false*, and call *notify()*. The call to notify() wakes up the consumer if it is waiting inside *get()*. If the buffer isn't empty, *wait()* is called to suspend the producer. This thread remains blocked until some other thread-the consumer in this case-calls *notify()* from within a synchronized method of the same *Buffer* object. The wait() call, like the sleep() call, can throw an exception that we need to catch.

The body of *get()* is almost the same as *put()*. We loop until the buffer isn't empty in this case, setting *empty* to *true* once the loop exist. The call to *notify()* awakens the producer if it is waiting inside the method.

Conclusion and practical results

It's clear from the two figures below, which show the practical results of running the producer-consumer program, the synchronization in work both the producer and consumer. In the first figure, the producer started its work before consumer. It produces some value to be consumed by the consumer. No matter how much it should produce as long as it is not empty.

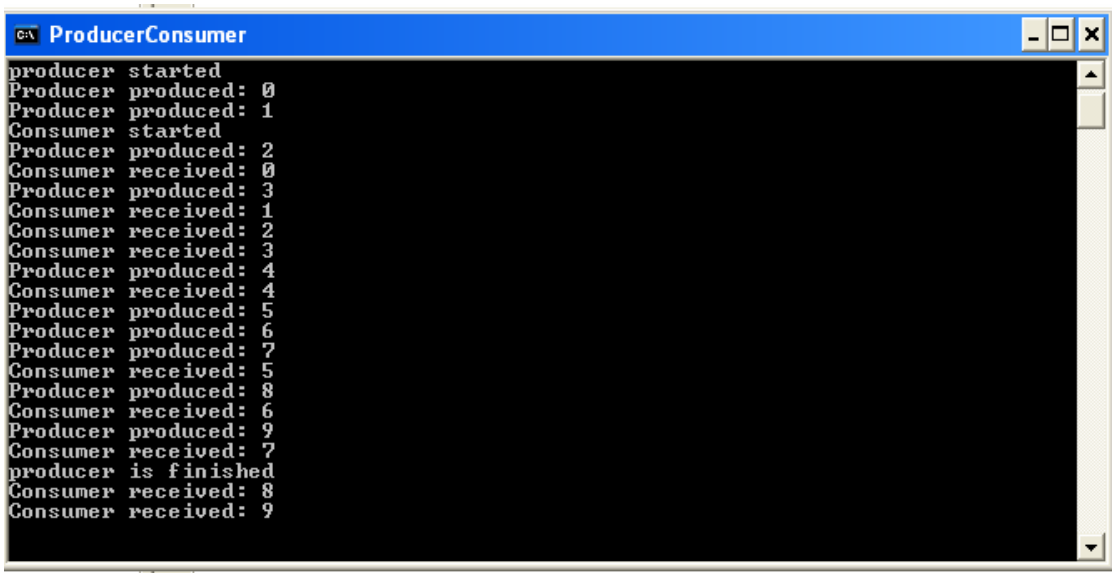

Figure (3) shows the practical results of running the program (the producer started before consumer.)

In the second figure below, although the consumer started to work before producer, it had to wait until producing some value to be consumed.

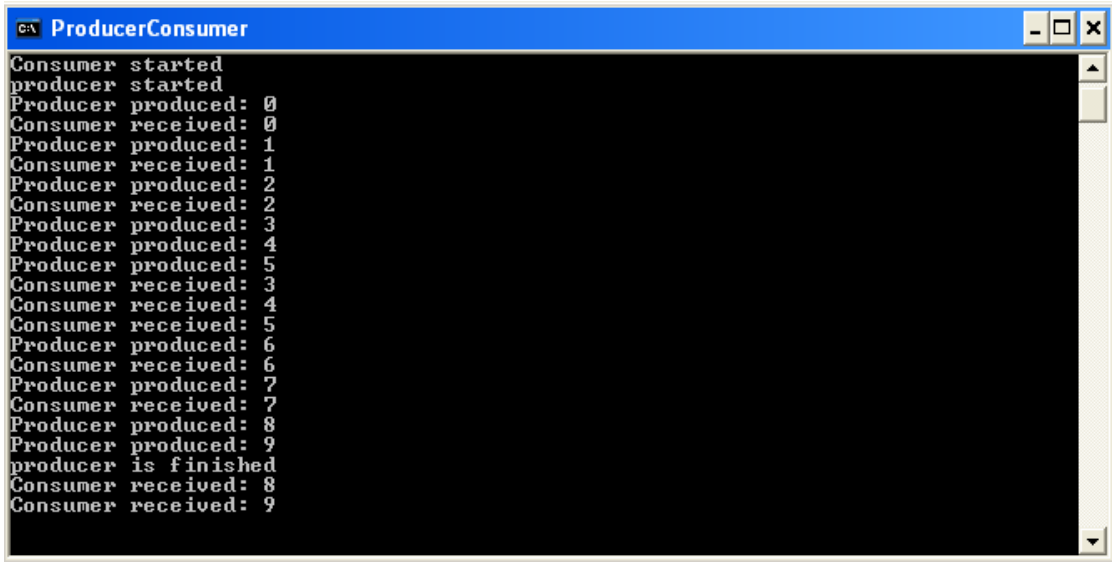

Figure (4) shows another practical results of running the program (the consumer started before producer.)

References

Arnow D. and Weiss G, 1998, Introduction to Programming Using Java, P.364. Addison Wesley Longman, USA.

Berg D. and Fritzinger J., 1998, Advanced Techniques for Java Developers. P.233-P.235, John Wiley & Sons, Inc. USA.

Bishop J, 1997, "Java Gently" programming principles explained. P.375-P.376. Addison-Wesley, UK.

Charatan Q. and Kans A., 2002, Java In Two Semesters. P.546-P.555, McGraw-Hill Education, UK.

Courtois T., 1998, Java Networking & Communications, P.71. Prentice Hall PTR, USA.

Harold E., 1997, Java Developers Resource, P.448. Prentice Hall PTR, USA.

Horton I., 2001, Beginning Java 2, P.480-P482. Wrox, Canada.

Liang Y., 2000, Introduction to Java Programming with JBuilder3, P.586. Prentice Hall, UK.

McHale C.,"Generic Synchronization Policies in C++", IONA Technologies, August 25, 2003.