Java's Insecure Parallelism

PER BRINCH HANSEN

Syracuse University, 2-175 CST, Syracuse, NY 13244
pbh@top.cis.syr.edu

Abstract: The author examines the synchronization features of Java and finds that they are insecure variants of his earliest ideas in parallel programming published in 1972–73. The claim that Java supports monitors is shown to be false. The author concludes that Java ignores the last twenty-five years of research in parallel programming languages.

Keywords: programming languages; parallel programming; monitors; security; Java;

We must expect posterity
to view with some asperity
the marvels and the wonders
we're passing on to it;
but it should change its attitude
to one of heartfelt gratitude
when thinking of the blunders
we didn't quite commit.

Piet Hein (1966)

1. PLATFORM-INDEPENDENT PARALLEL PROGRAMMING

Java has resurrected the well-known idea of *platform-independent parallel programming*. In this paper I examine the synchronization features of Java to discover their origin and determine if they live up to the standards set by the invention of monitors and Concurrent Pascal a quarter of a century ago.

In the 1970s my students and I demonstrated that it is possible to write nontrivial parallel programs exclusively in a secure language that supports monitors. The milestones of this work were:

- The idea of *associating explicit queues with monitors* [Brinch Hansen 1972].
- A *class notation for monitors* [Brinch Hansen 1973].
- A *monitor language*, Concurrent Pascal [Brinch Hansen 1975a].
- A *portable compiler* that generated platform-independent parallel code [Hartmann 1975].
- A *portable interpreter* that ran platform-independent parallel code on a wide variety of computers [Brinch Hansen 1975b].

- A *portable operating system*, Solo, written in Concurrent Pascal [Brinch Hansen 1976].
- A *book on abstract parallel programming* [Brinch Hansen 1977].

Monitors and Concurrent Pascal inspired other researchers to develop monitor variants [Hoare 1974a] and more than a dozen monitor languages, including Modula [Wirth 1977], Pascal Plus [Welsh 1979], and Mesa [Lampson 1980]. The portable implementation of Concurrent Pascal was widely distributed and used on a variety of computers ranging from mainframes to microcomputers [Brinch Hansen 1993b].

2. SECURITY AGAINST INTERFERENCE

Hoare [1974b] introduced the essential requirement that a programming language must be *secure* in the following sense: The language should enable a compiler and its run-time system to detect as many cases as possible in which the language concepts break down and produce meaningless results.[1]

For a parallel programming language the most important security measure is to check that processes access disjoint sets of variables only and do not interfere with each other in time-dependent ways.

The *Concurrent Pascal compiler* checked that every process and monitor only referred to its own variables; that processes interacted through monitor procedures only; and that processes did not deadlock by calling monitors recursively (either directly or indirectly).

The *Concurrent Pascal interpreter* ensured mutual exclusion of all operations on the variables of any process or monitor. It even made it impossible for a process and a peripheral device to access the same variable simultaneously.

Unless the parallel features of a programming language are secure in this sense, the effect of a parallel program is generally both unpredictable and time-dependent and is therefore meaningless. This does not necessarily prevent you from writing correct parallel programs. It does, however, force you to use a low-level, error-prone notation that precludes effective error checking during compilation and execution.

From the beginning, Hoare and I recommended extensive compile-time checking of modular parallel programs as an effective way of preventing most time-dependent errors. However, by 1991 Hoare sadly concluded that "a subsequent generation has lost that understanding." If you are unfamiliar with the rationale for interference control, you should study the history of the field from 1971–75. I see no point in repeating what I carefully explained and published decades ago. The reliability of this programming approach has been amply demonstrated in practice.

[1]This definition of *security* differs somewhat from its usual meaning of "the ability of a system to withstand attacks from adversaries" [Naur 1974].

3. SHARED CLASSES

My operating systems book [Brinch Hansen 1973] introduced the first programming notation for monitors, *shared classes*, based on a restricted form of the class concept of Simula 67 [Dahl 1972]. The book includes the bounded buffer shown in Fig. 1. (I have replaced my original Pascal notation with Java syntax.)

```
shared class B
{ int max = 10, p, c, full;
  int[] buffer = new int[max];

  public void send(int m)
  { await (full < max);
    buffer[p] = m;
    p = (p + 1) % max;
    full = full + 1;
  }

  public int receive()
  { await (full > 0);
    int m = buffer[c];
    c = (c + 1) % max;
    full = full - 1;
    return m;
  }

  public B()
  { p = 0; c = 0; full = 0; }
}
```

Figure 1 *A shared class with await statements (1973).*

This notation introduces a class of message buffers of the same type `B`. Each buffer may be shared by parallel threads. The buffer concept is defined in terms of its data representation, the possible operations on it, and its initialization. In Java terminology these class components are known as the *instance variables*, *synchronized methods*, and the *constructor* of a buffer.

A buffer instance `b` of type `B` is declared and used as follows by parallel threads:

```
B b;          b.send(5);          int x = b.receive();
```

For a particular class instance `b`, the following *restrictions* apply:

- The instance variables are *private* to the class instance and can only be accessed within the class.
- The synchronized methods are executed strictly one at a time as *critical regions* on the instance variables.

Hoare [1972] had introduced the concept of a *conditional critical region* that is delayed until a shared data structure satisfies a Boolean condition. In a shared class,

I expressed the same idea by means of an `await` statement, which can occur anywhere within a critical region [Brinch Hansen 1972].

4. EXPLICIT QUEUES

At the time I was concerned about the inefficiency of conditional critical regions which retest Boolean conditions repeatedly until they are true. As an alternative *I decided to let the programmer control the frequency with which scheduling expressions are reevaluated. I did this by associating explicit queues with shared variables. Critical regions can delay calling processes in these queues and resume them later* [Brinch Hansen 1972].

```
shared class B
{ int max = 10, p, c, full;
  int[] buffer = new int[max];
  event e;

  public void send(int m)
  { while (full == max) await(e);
    buffer[p] = m;
    p = (p + 1) % max;
    full = full + 1;
    cause(e);
  }

  public int receive()
  { while (full == 0) await(e);
    int m = buffer[c];
    c = (c + 1) % max;
    full = full - 1;
    cause(e);
    return m;
  }

  public B()
  { p = 0; c = 0; full = 0; }
}
```

Figure 2 *A shared class with an explicit queue (1972).*

Figure 2 illustrates this idea. The buffer class is extended with a single queue variable `e` of type `event`. Every synchronized method now begins with a *waiting loop* of the form

```
while (!condition) await(e);
```

and ends with the statement `cause(e)`. The `await` operation makes a process leave its critical region and enter the queue `e`. The `cause` operation enables all processes in the queue `e` to eventually reenter their critical regions one at a time.

The programmer can control the scheduling of processes to any degree desired by associating each queue with a *group* of similar processes or an *individual* process.

Since every instance of a Java class uses a single queue only, I have imposed the same restriction on Fig. 2.

The key idea is that the queuing operations automatically maintain mutual exclusion of all access to monitor variables during the evaluation of scheduling conditions.

Since my proposal was completely unrelated to the unpredictable event queues of the 1960s, I will call them *explicit queues* in this paper.

All subsequent monitor proposals were based on minor variations of the same ideas: *A monitor is essentially a shared class with explicit queues.*

5. SYNCHRONIZED JAVA METHODS

Only trivial changes are required to turn the shared class (Fig.2) into a correct Java class (Fig.3):

- The class is no longer declared as `shared`.
- The instance variables are declared as `private`.
- The class methods are declared as `synchronized` methods that may cause a run-time `Exception`.
- The queue variable `e` is replaced by a single anonymous queue.
- The scheduling methods, `await` and `cause` are renamed `wait` and `notifyAll`.

I remark in passing that I have no idea what general meaning the Java designers ascribe to a "critical region" that can be interrupted by exceptions (even if it includes an exception handler).

A comparison of Figs. 1–3 makes it clear that a Java class with synchronized methods, waiting loops and `notifyAll` statements is a variant of my earliest ideas in parallel programming: the shared class and scheduling queues, published in 1972–73.

Hoare's [1974a] contribution to the monitor concept was to replace my resume-all queues with first-in, first-out queues (known as *conditions*). Java includes a first-in, first-out variant of the `notifyAll` method, named `notify`.

Gosling [1996, p. 399] claims that Java uses monitors to synchronize threads. Unfortunately, *a closer inspection reveals that Java does not support a monitor concept*:

- Unless they are declared as `synchronized`, Java class methods are unsynchronized.
- Unless they are declared as `private`, Java class variables are `public` (within a package).

Consequently, parallel threads can access shared variables, either directly or indirectly, without any synchronization. One can, in fact, write a Java class that uses both private and public variables accessed by both synchronized and unsynchronized methods.

```
class B
{ private int max = 10, p, c, full;
  private int[] buffer = new int[max];

  public synchronized void send(int m)
    throws Exception
  { while (full == max) wait();
    buffer[p] = m;
    p = (p + 1) % max;
    full = full + 1;
    notifyAll();
  }

  public synchronized int receive()
    throws Exception
  { while (full == 0) wait();
    int m = buffer[c];
    c = (c + 1) % max;
    full = full - 1;
    notifyAll();
    return m;
  }

  public B()
  { p = 0; c = 0; full = 0; }
}
```

Figure 3 *A Java class with synchronized methods (1996).*

I do not see how one can assign any general meaning to a programming notation that invites time-dependent errors as the default case. Nor do I see how a compiler can detect such errors. *The failure to give an adequate meaning to thread interaction is a very deep flaw of Java that vitiates the conceptual integrity of the monitor concept.*

Well, if Fig. 3 is not a Java monitor, what is it then? It is just *a programming style that imitates insecure monitors.* Almost any programming language (including assembly language) enables you to adopt programming styles based on abstract concepts that are not supported directly by the language.

Since every Java object is associated with a single anonymous queue only, it would have been a significant improvement if Java had adopted my original shared classes with the `await` statements and access restrictions described earlier (Fig. 1).

6. JAVA'S MISTAKE

Java's most serious mistake was the decision to use the sequential part of the language to implement the run-time support for its parallel features. It strikes me as absurd to write a compiler for the sequential language concepts only and then attempt to skip the much more difficult task of implementing a secure parallel notation. This wishful thinking is part of Java's unfortunate inheritance of the insecure C language and its primitive, error-prone library of threads methods.

Six years ago, I wrote [Brinch Hansen 1993a]:

> The 1980s will probably be remembered as the decade in which programmers took a gigantic step backwards by switching from secure Pascal-like languages to insecure C-like languages. I have no rational explanation for this trend. But it seems to me that if computer programmers cannot even agree that security is an essential requirement of any programming language, then we have not yet established a discipline of computing based on commonly accepted principles.

In 1975 Concurrent Pascal demonstrated that platform-independent parallel programs (even small operating systems) can be written in a secure programming language with monitors. *It is astounding to me that Java's insecure parallelism is taken seriously by the programming community, a quarter of a century after the invention of monitors and Concurrent Pascal. It has no merit.*

Although the development of parallel languages began around 1972, it did not stop there. Today we have three major communication paradigms: monitors, remote procedures, and message passing. Any one of them would have been a vast improvement over Java's insecure variant of shared classes. As it is, Java ignores the last twenty-five years of research in parallel languages.

7. YOU GOTTA HAVE STYLE

If programmers no longer see the need for interference control then I have apparently wasted my most creative years developing rigorous concepts which have now been compromised or abandoned by programmers.

However, if you agree with me, but consider it futile to swim against the tide of the times, it seems appropriate to end this paper on programming language design by quoting Peter Naur's [1992] comments about the related subjects of writing and programming style [with emphasis added]:

> Good writing is very difficult, even for persons who have complete mastery of everyday spoken language. Good style is achieved only through insight, practice, and effort. By analogy *we can expect good programming style to remain a combination of sound principles, talent, and work, and the fight against poor style is never ending.*

Acknowledgements

It is a pleasure to acknowledge the helpful comments of Jonathan Greenfield, Al Hartmann, Giorgio Ingargiola, Henk Kruijer, Ted Lewis, Michael McKeag, Peter O'Hearn and Charles Reynolds.

References

1. Brinch Hansen, P. 1972. Structured multiprogramming. *Communications of the ACM 15*, 7 (July), 574–578.
2. Brinch Hansen, P. 1973. *Operating System Principles*, Section 7.2 Class Concept, July, 226–232, Englewood Cliffs NJ: Prentice Hall.
3. Brinch Hansen, P. 1975a. The programming language Concurrent Pascal. *IEEE Transactions on Software Engineering* 1, June, 199–207.
4. Brinch Hansen, P. 1975b. Concurrent Pascal machine. Information Science, California Institute of Technology, Pasadena, CA, October.
5. Brinch Hansen, P. 1976. The Solo operating system. *Software–Practice and Experience 6*, 2 (April–June), 141–200.
6. Brinch Hansen, P. 1977. *The Architecture of Concurrent Programs.* Prentice Hall, Englewood Cliffs, NJ. Includes Brinch Hansen 1975a, 1975b, 1976.
7. Brinch Hansen, P. 1993a. Letter to C. A. R. Hoare, February 20.
8. Brinch Hansen, P. 1993b. Monitors and Concurrent Pascal: A personal history. *2nd ACM Conference on the History of Programming Languages*, April, Cambridge MA: 1–35.
9. Dahl, O.-J., Dijkstra, E. W., and Hoare, C. A. R. 1972. *Structured Programming*, 175–220. New York: Academic Press, 175–220.
10. Gosling, J., Joy, B., and Steele, G. 1996. *The Java Language Specification*, Reading MA: Addison-Wesley.
11. Hartmann, A.C. 1975. A Concurrent Pascal compiler for minicomputers. Pasadena CA: Information Science, California Institute of Technology.
12. Hein, P. 1966. *Grooks.* Cambridge MA: The M.I.T. Press.
13. Hoare, C. A. R. 1972. Towards a theory of parallel programming. In *Operating Systems Techniques*, C. A. R. Hoare and R. H. Perrott, Eds., New York: Academic Press, 61–71.
14. Hoare, C. A. R. 1974a. Monitors: an operating system structuring concept. *Communications of the ACM 17*, (October), 549–557.
15. Hoare, C. A. R. 1974b. Hints on programming language research. In *Computer Systems Reliability*, C. Bunyan, Ed., Berkshire, England: Infotech International, 505–534.
16. Hoare, C. A. R. 1991. Letter to the author. Reprinted in Brinch Hansen [1993b].
17. Lampson, B. W., and Redell, D. D. 1980. Experience with processes and monitors in Mesa. *Communications of the ACM 23*, 2 (February), 105–117.
18. Naur, P. 1974. *Concise Survey of Computer Methods.* Lund, Sweden: Studentlitteratur.
19. Naur, P. 1992. *Computing: A Human Activity.* Reading MA: Addison-Wesley Publishing.
20. Welsh, J., and Bustard, D. W. 1979. Pascal-Plus—another language for modular multiprogramming. *Software—Practice and Experience 9*, 11 (November), 947–957.
21. Wirth, N. 1977. Modula: a programming language for modular multiprogramming. *Software—Practice and Experience 7*, 1 (January–February), 3–35.