Real-time programming safety in Java and Ada

Bo I. Sandén
Computer Science
Colorado Technical University
4435 N. Chestnut St.
Colorado Springs, CO 80907-3896
U.S.A.
Email: bsanden@acm.org
Phone: (719) 590-6733
http://iis-web.coloradotech.edu/bsanden

Abstract

Both Java and the Real-time Specification for Java contain concurrency-related constructs that are easily abused or simply misunderstood by a programmer without sufficient knowledge of multi-threading. This article lists a number of those constructs and shows how they are avoided in Ada. Many of the mistakes arise when a programmer confuses exclusion synchronization and condition synchronization. The article opens with an explanation of those concepts.

1. Introduction

Few industry-strength languages include multi-threading in their syntax. Among contemporary languages, Ada [1, 2] and Java [3-8] are the most prominent. The philosophy of concurrency is similar in Java and Ada95 and is based on the classic distinction between threads (tasks in Ada) on the one hand, and shared objects on the other. This has been the dominant paradigm for practical multi-threading for decades, although other models exist, such as the rendezvous paradigm of Ada83, which is still supported in Ada95. (The Ada83 paradigm is not further discussed here.) In Ada, shared objects are declared `protected`. In Java, they are instances of classes that have methods marked `synchronized`.

Although the philosophy of concurrency is similar, the attitude to safety and reliability is radically different in Ada and Java. Java threading is adequate for its original purpose: windows programming and applets. But Java is now being fitted with real-time extensions and may be applied to safety critical software. The language has many potentially abusable constructs and programmer pitfalls. This is certainly true for Java in general [9] but is particularly important with concurrent software, which is notoriously difficult to debug.

The Real-Time Specification for Java (RTSJ [10, 11]) does nothing to remove the pitfalls. It intends to make Java useful for real-time applications by circumventing garbage collection and providing interrupt handling, not to make the language less error prone. In fact, using the new classes that represent different types of memory areas or the new construct for asynchronous transfer of control correctly is far from easy. The case that Ada95 is a much safer language for real-time embedded applications than Java can easily be made.

This article is intended for Ada programmers who may be taking the Ada concurrency features for granted. The purpose is to view those features against a backdrop of the pitfalls of a more traditional concurrency implementation without emphasis on safety. I point out a number of Java threading pitfalls and note how they are prevented in Ada. While I briefly summarize the Ada tasking model, the reader is assumed to have an understanding of Ada tasking and sufficient understanding of Java to be able to read small program excerpts, but is not expected to know much about Java multi-threading. For a comprehensive comparison of concurrency features in the two languages, see [12].

1.1. Forms of synchronization

The traditional concurrency model with threads on the one hand and shared objects on the other relies on a distinction between *exclusion synchronization* and *condition synchronization,* described as follows.

Exclusion synchronization is used to stop two threads from operating on the same object at the same time and thereby jeopardizing the integrity of its data. Java provides exclusion synchronization for any synchronized method. Ada provides exclusion synchronization for protected operations. A block of code that is executed under exclusion synchronization is called a *critical section*. It is bracketed by instructions that acquire and release a lock on an object.

Each thread is expected to maintain exclusive access for a very short time, making it unlikely that a thread will ever find an object locked. If it does find an object locked, only a brief wait should be expected. It is even more unlikely that two threads should attempt access to the same object while its is locked. For this reason, one need not be concerned with maintaining a orderly queue of threads pending on an object lock. Implementations of exclusion synchronization typically let a thread that encounters a locked object yield the processor in the hope that it will find the object unlocked when next made running. If the object is still locked, the thread again yields the processor. With multiple processors, one often uses a *spin lock*, that is, the thread enters a loop where it repeatedly attempts access until it is successful. I shall use the term "spin lock" for both the single processor and multi-processor cases.

While a thread, `l`, is operating on a shared object `O`, under exclusion synchronization, it may be preempted by a higher-priority thread, `h`, which also needs exclusive access to `O`. Unavoidably, `h` must wait for `l` to exit the critical section. Such a situation where a higher priority thread is waiting for a lower priority thread is referred to as *priority inversion*. If `l` continues executing at its normal priority, a thread, `i`, of intermediate priority may preempt `l`. This leads to an avoidable situation where `h` is waiting for two lower-priority threads. To avoid it, `l` can be given a priority boost. One possibility is to let `l` *inherit* `h`'s priority once `h` tries to access `O`. The other possibility is to define a *ceiling priority* for `O`, which is used by any thread while it executes a synchronized method on `O`. The ceiling priority must be as high as the priority of any thread that ever operates on `O`.

Condition synchronization is when a thread cannot proceed if a certain condition holds. A buffer, shared by two or more threads provides a classic example. A `Buffer` object has the operations `put()`, called by producer threads, and `get()`, called by consumer threads. A thread that calls `put()` must wait if the buffer is full, and a thread calling `get()` must wait if it is empty. There is no assumption that this wait will be brief. Threads may spend considerable time waiting, and must be queued, typically first-in-first-out per priority. A thread conditionally waiting for an object never holds the object locked and should not normally hold other objects locked.

Condition synchronization must be used to control access to any shared resource that is held long enough for a queue of waiting threads to form. Examples of resources of this nature range from a printer or a database record to resources in the problem domain of a control system such as railroad segments and automated vehicles in a flexible manufacturing system. Access to such a resource is handled by means of an object with a Boolean variable `busy`, say, and operations such as `acquire()` and `release()`. Once a thread has successfully acquired the resource and set `busy` to `true`, it releases the object lock, allowing other threads to call `acquire()` and place themselves on queue.

The term "condition synchronization" gives no hint that condition synchronization is often used to control mutually exclusive access to domain resources and, in general, to resources held for a long time. The terms "competition synchronization" and "cooperation synchronization" for exclusion and condition synchronization respectively are no better [13]. The rationale here is that the producer and consumer threads must cooperate to manage an empty and a full buffer but compete over the access to the operations. Again, this ignores the use of condition synchronization to control exclusive access to a domain resource. An alternative way to characterize synchronization is to distinguish between exclusive access with *short extent* in time (exclusion synchronization) and *long extent* (condition synchronization) [14], but this plays down the conceptual difference between the two kinds of synchronization.

The distinction between exclusion and condition synchronization is crucial in real-time concurrent programming. While Ada and Java both support exclusion and condition synchronization, Ada helps the novice programmer by clearly separating the concepts syntactically. Java does not, and some of the pitfalls result from a confusion of them.

2. Ada concurrency model

Ada95 implements concurrency with two kinds of entities: *tasks* and *protected objects*. You define a task type, which is instantiated as any other type, or a singleton task. *Protected objects* have monitor-like behavior. They can have *protected operations* of three kinds: *functions*, *procedures* and *entries*. These are declared in the specification of the protected type[1] as for example the following:

```
protected type X is
      function F1 (...) return Type1;
      procedure P1 (...);
      entry E1 (...);
private
      --   Attribute variables
      --   Private operations including interrupt handlers
end X;
```

All protected operations have exclusion synchronization built in. The differences between protected functions, protected procedures and entries are as follows:

Protected functions are read-only. They are prohibited from changing the attribute values of the protected object and are subject to a read lock: Any number of function calls on a given object are allowed simultaneously, but not during a procedure or entry call on the object.

---

[1] Ada also provides for singleton protected units.

Protected procedures are allowed to change attribute values. They are subject to a write lock: Only one procedure (or entry) call at a time is allowed on a given object, and not during any function call.

Like procedures, *entries* are allowed to change attribute variable values and are subject to the write lock. In addition, an entry can provide condition synchronization by means of a *barrier condition*, which appears in the body of the protected type. An entry call only proceeds when the condition is true. For example, an entry Get, which is only allowed when the number (`Num`) of items in a buffer is greater than zero, may be declared as follows:

```
entry Get (...) when Num > 0 is ....
```

A task that calls `Get` when `Num = 0` is put on a queue that is FIFO per priority. Each protected object has one queue per entry.

Any variables in the barrier condition are supposed to be attribute variables of the protected object, on which the entry operates. The values of those variables can only be changed by calls to protected procedures and entries on that object.

At the end of each procedure or entry call on a given object, its barriers are evaluated. If a barrier is found to be true, the most eligible task in the corresponding queue is activated and executes the entry body. Tasks that are already in a queue have precedence over new callers according to the principle of "internal progress first".

3. Java concurrency model

In Java, any method in any class can be declared *synchronized.* This is exclusion synchronization: A write lock per object is applied, so that only one synchronized method at a time can operate on a given object. A synchronized method in Java is similar to a protected procedure or entry in Ada in that it is implicitly bracketed by instructions that acquire and release the object lock.

Condition synchronization in Java relies on explicit tests programmed into the synchronized methods, as for example:

```
while (0 == Num) {wait();}
```

If Num is zero, a calling thread calls `wait()` and thereby enters the object's *wait set*. There is one wait set per object, not one per entry per object as in Ada. A thread, `t`, that executes a synchronized method on an object may change the truth value of a condition that may affect one or more threads in the wait set. Before leaving the method, `t` must explicitly `notify` any such threads. The call `notifyAll()` reactivates all threads waiting for conditional access to an object.

The Java thread model hides little from the programmer. This makes it quite flexible for the old hand at concurrency. Apart from the tie-in with object-orientation, the thread model is in fact very similar to what I first encountered when manipulating threads provided by the UNIVAC 494 operating system in assembler programs 30 years ago. In a sense, this makes Java more pedagogical than Ada because it exposes the details of synchronization. But by the same token, the Java model is much more error prone. Very little protects Java programmers from the consequences of their own mistakes. Unfortunately, many programmers are not shy about trying things they don't fully understand and then testing the program to see if it

works. Many concurrency related bugs are subtle enough to pass most tests. An Ada programmer cannot easily make clerical errors with unintended effects on the program behavior.

An advantage of the Java model is that it can be implemented with less overhead than the Ada model, but this issue appears to be losing much of its earlier importance. Further, a Java class with synchronized operations can be part of the inheritance hierarchy. Because a thread can hold multiple locks on the same object, a synchronized method can easily call a synchronized method in a parent class, including one that it overrides. If the parent method has a wait loop, the thread may enter the wait set.

In Ada, a protected type is not extensible, that is, it cannot be subclassed. Although Ada95 includes object-oriented features including inheritance, polymorphism and dynamic binding, these were not extended to protected objects [15].

3.1 Real-time Java

The Real-Time Specification for Java (RTSJ) [10, 11] is an effort to make Java useful for real-time programming. (An alternative specification is given in [16].) One premise is that a real-time program must be predictable so that the programmer can determine *a priori* when certain events will occur. This is not true for standard Java for a number of reasons. First, the garbage collector, which can interrupt any other processing, adds an element of randomness. Second, different scheduling policies cannot be imposed in standard Java. (Scheduling policies such as the rate-monotonic algorithm allow you to prove that a set of threads meet their specified deadlines.) Third, in standard Java, threads placed in a wait set are reactivated in arbitrary order, independent of when they attempted access; the wait set is not a FIFO queue.

To deal with these problems, RTSJ introduces a number of new classes, most of which are necessary for working around the garbage collector. One such class is `NoHeapRealtimeThread` *(NHRT)*, which is a descendant of `Thread`. *NHRT* threads have higher priority than the garbage collector so are not subject to arbitrary delays. This places many restrictions on the programmer, however. For example, an *NHRT* thread cannot allocate objects on the heap. Instead, RTSJ provides for various kinds of special memory areas.

RTSJ also stipulates that threads in a wait set must be kept in FIFO order within priorities. This means that `notify()` reactivates the thread with the highest priority. If there are more than one thread with that priority, the one that has waited the longest is reactivated.

RTSJ uses priority inheritance as the default control policy to address priority inversion. A priority ceiling protocol is also specified. Finally, to further support real-time programming, RTSJ allows the programmer to specify interrupt handlers.

4. Java pitfalls and their Ada solutions

Apart from the extensions provided by RTSJ, real-time Java relies on the threading and synchronization models of standard Java. Next, I discuss in more detail some features of these models from a real-time point of view, focusing on the associated pitfalls. I also discuss how each pitfall is prevented by the Ada95 syntax and semantics. Although programming to RTSJ is in many respects quite involved, I do not address any programming pitfalls that may appear there.

4.1 Defining and starting threads

Java provides the abstract class `Thread`, whose method `run()` represents the logic that a thread performs. It corresponds to the executable part of a task body. A standard way of creating threads is to declare a new class, `T`, that extends `Thread` and overrides `run()` with appropriate processing. Each instance `To` of `T` has its own thread, which is explicitly started by means of the call `To.start()`. Once started, the thread executes `T`'s `run()` method and has access to `To`'s data.

Because Java has no multiple inheritance, an additional mechanism is necessary for the case where a class, `R`, that needs a thread, already extends another class, such as `Applet`. For this situation, Java provides the interface `Runnable`. The programmer makes `R` extend `Applet` and implement `Runnable`. Instantiating `R` creates a *runnable object*, `Ro`, say. To associate a thread with `Ro`, you submit `Ro` as an argument to one of `Thread`'s constructors and then call `start()` on the resulting `Thread` instance. This is typically done in a statement such as:

```
new Thread(Ro).start();
```

Java pitfalls. Once you have a class `R` that implements `Runnable`, Java gives you two ways to create multiple threads that execute `R`'s `run()`method. First, you can instantiate `R` *n* times, and submit each instance once as a parameter to one of `Thread`'s constructors. Now you have *n* instances of `R`, each with a thread, so each thread has its own set of instance variables. But Java also lets you submit the same instance of `R` repeatedly to `Thread`'s constructor. The result is a subtly different case where multiple threads are tied to one object and share its instance variables. Two (or more) of these threads can directly manipulate those instance variables simultaneously, and possibly introduce inconsistencies.

Ada. Ada's model for defining and starting tasks is cleaner. You declare a task type, which is instantiated as any other type, or a singleton task. The task is started automatically, either when the first executable statement is reached after the declarations, or, if a task type is dynamically instantiated, immediately upon instantiation. Each task instance has its own private data.

4.2 Synchronized objects

Java provides all objects with the potential for monitor-like behavior, that is, approximately the behavior of a protected object in Ada. Every object has a lock variable, which is hidden from the programmer and cannot be accessed from methods on the object. Exclusion synchronization is accomplished by specifying a method as `synchronized`, as in:

```
void synchronized m() ...
```

When a synchronized method is called on some object, `O`, its code is implicitly bracketed by statements that acquire and release the lock on `O`. That is, a thread calling `O.m()` locks `O` as a whole, so that no other thread can perform any synchronized method on `O` (or execute any block synchronized with respect to `O` as discussed below). This synchronization feature is built in, which guarantees that the lock is always released when a thread leaves a synchronized method, even if this happens by means of the exception handling mechanism. I shall refer to any instance of a class that has at least one synchronized method or synchronized block as a *synchronized object*.

A Java programmer can choose to specify some but not all the methods of a class as synchronized. This has some useful applications. For example, a method that returns a constant or the value of a single attribute need not be synchronized.

Java pitfalls. The freedom to specify selected methods of a class as synchronized opens the door for mistakes. In the buffer example, the programmer may declare `get()` synchronized and not `put()`. This allows different threads to call `put()` simultaneously. These calls may also overlap with a call to `get()`. This jeopardizes the integrity of the buffer data structure. The program may still work much of the time, but will produce occasional errors, especially when run on a symmetric multi-processor providing true parallelism. Such errors tend to be hard to find by testing – although more easily by inspection by an experienced thread programmer. Omitting the keyword synchronized altogether, for `put()` as well as `get()` would further exacerbate the situation.

A programmer must ensure that the instance variables that the synchronized methods operate on are private so that they cannot be directly accessed and changed by a method operating on some other object. Even if they are private, you must ensure that they are not changed by a static method defined for the class.

Ada. All operations on a protected object require either a read lock or a write lock. You cannot include a non-protected operation in a protected object. A protected function can be used to return constants, etc., as can an unsynchronized method in an otherwise synchronized Java class.

4.2.1 Synchronized blocks

In addition to synchronized methods, Java provides synchronized *blocks*, which have no Ada counterpart. Any block in any method can be synchronized with respect to an object – not necessarily the current instance of the class where the method appears – by means of the syntax:

```
synchronized (Expression) { /* Block B */ }
```

Expression must evaluate to a reference to some object, `Vo` of class `V`, say. As for synchronized methods, exclusion synchronization is implicit, so `B`'s code is bracketed by statements to acquire and release the lock on *Vo*. A synchronized method and a synchronized block are both critical sections.

Consider first the case where `B` is part of some method, `m()`, on class `V` and is synchronized with respect to the current object as follows:

```
class V ...
{
      void m()
      {
            synchronized (this)
            {
                  /* Block B*/
            }
      }
}
```

This design is an alternative to synchronizing `m()` and can be used if only parts of `m()` requires exclusive access. That way, two or more threads can simultaneously execute those parts of `m()` that are outside B, so concurrency may be increased. An alternative design is to make B into a separate, synchronized method called from within `m()`.

As mentioned, the block B in `m()` can be synchronized with respect to any object, not only the current one. In the following excerpt, B is synchronized with respect to object `Wo` of class `W`. This means that before entering the block B, the thread that called `m()` in order to operate on an object of class `V` acquires the lock on object `Wo` of `W`.

```
class V ...
{
      void m()
      {
            synchronized (Wo)
            {
                  /* Block B*/
            }
      }
}
```

Synchronized blocks are useful when different threads need exclusive access to some object in order to perform their own, particular operations on it. Such an object is sometimes a printer or a window to which many threads write their own tailored outputs such as log entries as in the following example:

```
synchronized (myPrinter)
{
      // series of statements producing output
}
```

In this case, it can be inconvenient to make every possible combination of output statements into a method for the printer class.

Java pitfalls. By synchronizing blocks with respect to some object you effectively create "distributed" methods that are not included with other instance methods in the class definition. From looking at a class definition, you cannot tell whether any blocks exist that are synchronized with respect to its instances. A class without synchronized methods in its definition may appear to a maintenance programmer as an un-synchronized class even though there are blocks synchronized with respect to its instances.

Ada has no equivalent to synchronized blocks. All operations on a protected object are specified in its declaration.

4.3 Condition synchronization

The most common idiom for condition synchronization in Java is the statement

```
while (cond) {wait();}
```

I shall refer to this statement as a *wait loop*. It makes the calling thread wait as long as `cond` holds. If `cond` is true, the thread calls `wait()` and thereby places itself in the *wait set* of the current object, `O`, and releases `O`. The wait set contains all threads waiting for *conditional access* to `O`.

The wait loop syntax is somewhat complicated by the need to handle an `interrupted` exception that can be caught by a thread while it is in the wait set. This is possible because this particular exception is thrown by a different thread than the one that must catch it. Unless the exception is propagated to an enclosing scope, a construct such as the following is necessary:

```
while (cond) try {wait();}
     catch(InterruptedException e) {
          /* Take action or ignore the exception */
     }
```

Pitfalls. The wait loop is like an incantation that should always be repeated in almost exactly that form. For example, the variation

```
while (cond) {yield();}
```

stops the calling thread from proceeding against `cond` but does not release the object. This means that other threads that are supposed to change `cond` by calling synchronized methods on the object cannot do so.

A more insidious mistake is to replace the wait loop with the quite similar statement

```
if (cond) {wait();}
```

This statement makes the calling thread enter the wait set and release the object, but only once. When reactivated, the thread continues after the `wait()` call and proceeds in the synchronized method even if `cond` is true [12]. This is particularly dangerous, since `notifyAll()` must often be used and relies on the wait loop. When `notifyAll()` activates a thread that is not to proceed, the thread is supposed to retest its condition and return to the wait set. Substituting `if` for `while` leads to a typically transient error. Under unlucky circumstances, it can remain undetected for some time, perhaps until an additional thread is introduced.

Ada. Condition synchronization is achieved by means of entry barriers, which are built into the syntax. Their format is not susceptible to easy programming mistakes. One possible mistake is to include in a barrier condition a variable that is defined outside the protected object. In that situation, it is possible to change the value of the condition without notifying waiting threads.

4.3.1 Placement of the wait loop

The wait loop in Java most often appears at the very beginning of a critical section and is reached immediately after a thread locks the object. But it can be placed anywhere within a synchronized method or block. As a simple example of one or more statements separating the wait loop from the beginning of a method, you could count the number of calls to a method, `m()`, in an instance variable `CallCounter` in the following way:

```
synchronized void m()
{
     CallCounter ++;
     while (cond) {wait();}
     ....
}
```

Here, `CallCounter` is incremented exactly once for each call, no matter if the calling thread enters the wait set. Its value equals the number or calls to `m()` including those where the thread is still in the wait set. In a slightly more sophisticated example, the statements before the wait loop could maintain a list of the thread identities of the latest *n* callers. One can also instrument the wait loop itself similarly by including statements before and/or after the `wait()`call.

The textbook case for placing the wait loop deeper inside a critical section is when a method allocates resources to calling threads. It may turn out that the request of a calling thread, `t`, cannot be satisfied until additional resources become available. In that situation, `t` can place itself in the wait set, release the object and wait to be notified by a thread that has released resources. Once notified, `t` continues immediately after the `wait()` call with exclusion synchronization in force. If a synchronized method has one or more such `wait()` calls, a thread can effectively execute it in segments separated by those calls, entering a new segment each time it is successfully reactivated from the wait set.

Java pitfalls. The syntactical freedom to place the wait loop anywhere in a critical section allows certain errors. Even if the wait loop is initially placed at the very beginning of the critical section, a maintainer can unintentionally insert statements between the beginning and the wait loop. These statements are executed exactly once by every thread that attempts access to the critical section regardless of the condition. This may be even more treacherous if there are already statements between the beginning of the critical section and the wait loop, as in the `CallCounter` example. The maintainer may not realize the difference in status between statements placed before and after the wait loop.

Ada. Because protected objects in Ada are syntactically distinct, there is no easy way to include a statement such as `CallCounter := CallCounter + 1;` in an entry in such a way that it would be executed exactly once under exclusion synchronization before the barrier has been passed. (Certain elaborate maneuvers are possible if you include in the barrier condition a function call with side effects.)

An Ada entry body cannot be broken into segments where a task would execute a segment, then release the object, put itself on queue and continue with the next segment upon reactivation. In Ada, each such segment must be an entry. A task that is executing an entry can call `requeue` and place itself on the queue of the same or another entry [2]. Requeuing is sometimes considered an advanced Ada topic. An intuitive example of requeuing when a resource turns out to be unavailable is given in [17].

4.3.2 Notification of waiting threads

A Java thread that executes a synchronized method on `O` and changes a condition that may affect one or more threads in `O`'s wait set must `notify` those threads. In standard Java, `O.notify()` reactivates *one* arbitrarily chosen thread, `t`, in `O`'s wait set. If the call is correctly placed within a wait loop, this means that `t` reevaluates the condition and either proceeds in the synchronized method or reenters the wait set. In RTSJ, the most eligible thread is reactivated.

The call `O.notifyAll()` releases *all* threads waiting for conditional access to `O`. This is useful when a condition has changed so that multiple threads can proceed. But calling `notifyAll()` is sometimes necessary even though you want only a single thread to proceed. In standard Java, this is the only way to give preference to the highest priority thread. It is inefficient if there are many threads in the wait set, since they must all attempt access, and only one will succeed [18].

Because there is only one wait set per object, you must also call `notifyAll()` instead of `notify()` if an object's wait set may include threads pending on different conditions. If you change one of the conditions, you must activate all the threads to make sure that a thread pending on that condition is notified, if it is in the set. This is true in RTSJ as well as in standard Java.

When a thread calls `wait()`, `notify()` or `notifyAll()`on an object, it must have the object locked. The wait set is a shared data structure that must be protected from conflicting access, but has no lock of its own.

Java pitfalls. Unlike exclusion synchronization, condition synchronization is not automatic; you have to explicitly notify waiting threads. An obvious pitfall is to forget to insert `notify()` calls at all the necessary places. This is particularly treacherous if a method has unusual exits, as via exception handlers. A related mistake is to call `notify()` instead of `notifyAll()` when threads in the wait set may be pending on different conditions.

A way to reduce the risk of forgotten notifications in standard Java is to include a timeout parameter in every `wait()` call. After the given time, the thread is activated, and if the `wait()` call is placed inside a correct wait loop, the thread reevaluates the condition and either proceeds or reenters the wait set. In RTSJ this can only be used if you don't care about maintaining the FIFO per priority queuing discipline.

Ada. As long as the entry barrier depends only on variables local to the protected object, the most eligible, waiting thread is automatically activated after a protected procedure or entry call on the object has changed the truth value of the condition. Waiting tasks are queued per entry, so precise notification can be achieved: If a single condition is changed, only a task waiting on that condition is activated.

4.3.3 Controlling access to domain resources

Condition synchronization is used to give one thread at a time exclusive access to a shared resource in the problem domain, such as a forklift truck in an automated factory application [14, 19, 20]. In this example, jobs on the factory floor that need the forklift are represented by `Job` threads in the software. A forklift operation may continue for several minutes and must be performed under condition synchronization because we want waiting jobs to form a FIFO queue per priority. The object controlling the forklift – instance `F` of class `Forklift`, say – typically has an attribute, `busy`, that reflects the availability of the forklift, and the synchronized operations `acquire()` and `release()`, where `acquire()` contains a wait loop such as the following:

```
while (1 == busy) {wait();}
```

The corresponding notification call is in `release()`. Statement sequences where the forklift is operated are bracketed by calls to `acquire()` and `release()` whether they appear in `Job`'s `run()` method or in other unsynchronized methods.

While one job is using the forklift, other `Job` threads can call `F.acquire()` and place themselves in `F`'s wait set. The variable `busy` serves as the lock on the physical forklift while `F`'s hidden lock variable only serves to control the access to the variable `busy` itself.

Explicitly calling `acquire()` and `release()` in this fashion is similar to working with a semaphore, and may be counterintuitive if you have been taught that semaphores are a primitive way of controlling the access to a shared resource. A synchronized method or block is a more abstract representation that hides the semaphore operations. But when controlling access to shared resources in the problem domain, we must invert the abstraction by using a synchronized object to implement a semaphore [20].

In the example with the shared printer, we can choose whether to consider the wait to be of long or short extent. In the solution in section 4.2.1, each thread's operations on the printer are enclosed in a synchronized block as follows:

```
synchronized (myPrinter)
{
    // series of statements producing output
}
```

This exclusion synchronization assumes that the printer operations are quick. If other threads try to access the printer during the exclusive access, they spin, waiting for the lock. There is no direct way to ensure that they will ultimately access the printer in a first-in-first-out fashion.

In an alternative solution based on condition synchronization, you define `acquire()` and `release()` methods in the `Printer` class (or another class), introduce a variable such as `busy`, and bracket the series of statements with calls to those methods:

```
myPrinter.acquire();
// series of statements producing output
myPrinter.release();
```

Here, `acquire()` contains a wait loop, and threads that must wait for the printer enter the wait set. A downside is that this solution makes the programmer responsible for inserting one or more `release()` calls to ensure that the printer is released even if an exception is thrown while the output is being produced.

Java pitfalls. By convention, all critical sections should be programmed to minimize the time an object is held locked. A thread that is waiting on a condition should release its object locks and be placed in a wait set. But nothing stops a programmer from making a thread hold an object lock for an arbitrarily long time. A trivial way to do this is to call `sleep(...)` inside a synchronized method. Two other cases are described next.

Controlling domain resources. The confusion of long and short waits may typically occur in real-time applications that control resources in the problem domain. In the automated factory domain, the forklift operation may be implemented by means of the following synchronized block within the `run()` method of the `Job` class:

```
synchronized (F)
{
        // Operate the forklift
}
```

This ensures mutual exclusion of jobs using the forklift and may at first seem more elegant than the solution with semaphores. But if the forklift operation continues for minutes, `Job` threads that need the forklift are not put in a wait set (and FIFO queued per priority in RTSJ) but spin until they find `F` unlocked. Which `Job` thread gets to the forklift next is then quite arbitrary. To avoid this, condition synchronization must be used.

In RTSJ, exclusion synchronization invokes the control policy to minimize the effect of priority inversion. Assume first that the default policy, priority inheritance, is in effect. If a job at priority `l` is currently operating the forklift and a higher priority job, `h`, attempts to get the lock, `l`'s remaining forklift operations will be executed at priority `h`. This skews `l`'s priority relative to any jobs with priorities between that of `l` and that of `h`. The ceiling priority protocol has an even more fundamental effect in that all forklift operations will always be carried out at the highest priority of any job.

Nested synchronized blocks. Another way of inadvertently mixing long and short waits is with nested critical sections. We can insert a wait loop in a nested synchronized block as follows:

```
synchronized(r1)
{
        ....
        synchronized(r2)
        {
                while (cond) {r2.wait();}
                ....
        }
}
```

If `cond` is `true`, the calling thread enters `r2`'s wait set and releases `r2`. But it keeps `r1` locked, and lets other threads that need access to `r1` spin rather than wait in a wait set. Incidentally, the following is also legal:

```
synchronized(r1)
{
        ......
        synchronized(r2)
        {
                while (cond) {r1.wait();}
                ....
        }
}
```

In this case, the calling thread enters `r1`'s wait set and releases `r1` while keeping `r2` locked.

Ada. The Ada syntax is certainly clearer about the distinction between exclusion and condition synchronization. Any protected operation provides exclusion synchronization automatically. Any "potentially blocking operation", that is, essentially anything that can take time, is forbidden in a protected operation,

ensuring that the extent in time of mutual exclusion is kept short. For example, you cannot call an entry of some protected object `r2` while you are executing a protected operation on the object `r1`, which would be the Ada equivalent of nested synchronized blocks.

Condition synchronization requires an entry with a barrier condition. The only way to control access to a shared domain object is by means of a semaphore object similar to the `Forklift` class in Java. A `Forklift` protected object would have an `entry Acquire` with a barrier such as `not busy` and a procedure `Release`.

In the case of the printer, if it is undesirable to define protected procedures for each different combination of printer operations, a semaphore object is the only solution permitted in Ada. It would be a protected object `My_Printer` with the `entry Acquire` and the `procedure Release`.

5. Conclusions

Java was not originally intended as a language for systems with high reliability requirements, but its popularity has prompted its use for ever wider sets of applications. The Real-Time Specification for Java removes some of the obstacles associated with garbage collection but retains many pitfalls.

Java is adequate for many kinds of concurrent software, but for critical real-time applications it remains a considerably riskier choice than Ada, which was intended for such applications. This is so because Java lacks safeguards against programming errors that are easily committed by a programmer without a sufficiently deep understanding of concurrency issues. There is a trade off here where Java's popularity and the availability of Java programmers must be weighed against the risk exposure caused by those programmer mistakes the language readily allows.

References

[1] Barnes, J. G. P. (1998) *Programming in Ada95,* 2nd Ed., Addison-Wesley.

[2] Burns, A. and Wellings, A. J. (1998) *Concurrency in Ada,* 2nd Ed., Cambridge University Press.

[3] Lea, D. (2000) *Concurrent Programming in Java,* 2nd Ed., Addison-Wesley.

[4] van der Linden, P. (2001) *Just Java 2,* 5th Ed., Prentice Hall.

[5] Holub, A. I. (2000) *Taming Java Threads*, Apress.

[6] Hyde, P. (1999) *Java Thread Programming*, Sams Publishing.

[7] Oaks, S. and Wong, H. (1997) *Java Threads*, O'Reilly.

[8] Brinch Hansen, P. (1999). Java's insecure parallelism, *ACM SIGPLAN Notices* **34/4**, 38-45.

[9] Alexander, R. T. and Bieman, J. M. and Viega, J. (2000) Coping with Java programming stress, *IEEE Computer* **33/4**, 30-38

[10] Bollella, G. and Gosling, J. (2000) The real-time specification for Java. *IEEE Computer* **33/6**, 47-54

[11] Bollella, G. and Gosling, J. and Dibble, B. P. and Furr, S. and Turnbull, M. (2000) *The Real-time Specification for Java,* Addison-Wesley.

[12] Brosgol, B. M. (1998) A comparison of the concurrency features of Ada95 and Java, *Proc. SIGAda '98*, (*Ada Letters* **XVIII/6**,175-192).

[13] Sebesta, R. W. (2002) *Concepts of Programming Languages*, 5th Ed., Addison-Wesley.

[14] Sandén, B. I. (1994). *Software Systems Construction with Examples in Ada*. Prentice-Hall.

[15] Wellings, A. J. and Johnson, R. W. and Sandén, B. I. and Kienzle, J. and Wolf, T. and Michell, S. (2000) Integrating object-oriented programming and protected objects in Ada95. *ACM TOPLAS* **22/3**, 506-539. (Reprinted in *Ada Letters* **XXII/2** (June 2002), 11-44.

[16] International J Consortium (2000), *Specification, Real-Time Core Extensions*, Draft 1.0.14, 2 September 2002. http://www.j-consortium.org

[17] Sandén, B. I. (1996) Using tasks to capture problem concurrency. *Ada User Journal* **17/1**, 25-36.

[18] Vermeulen, A. and Ambler, S. W. and Bumgardner, G. and Metz, E. and Misfeldt, T. and Shur, J. and Thompson, P. (2000) *The Elements of Java Style*, Cambridge University Press.

[19] Sandén, B. I. (1997) Modeling concurrent software. *IEEE Software* **14/5**, 93-100.

[20] Carter, J. R. and Sandén, B. I. (1998) Practical uses of Ada95 concurrency features. *IEEE Concurrency* **6/4**, 47-56.