Architectures, Languages and Patterns
P.H. Welch and A.W.P. Bakkers (Eds.)
IOS Press, 1998

Java Threads in the Light of occam/CSP - II

Peter H. WELCH
Computing Laboratory, University of Kent at Canterbury, CT2 7NF.
P.H.Welch@ukc.ac.uk

Abstract. Java provides support for parallel computing through a model that is built into the language itself. However, the designers of Java chose to be fairly conservative and settled for the concepts of threads and monitors. Monitors were developed by Tony Hoare (and others) in the early 1970s as a structured way of using semaphores to control access to shared resources. Hoare moved away from this, in the late 1970s, to develop the theory of Communicating Sequential Processes (CSP). One reason was that the semantics of monitors are not WYSIWYG, so that designing robust parallel algorithms at this level is seriously hard. This tutorial will look at how this impacts on threaded applications written in Java.

Fortunately, it is possible to introduce the CSP model into Java through sets of classes implemented on top of its monitor support. By restricting interaction between active Java objects to CSP synchronisation primitives, Java thread semantics become compositional and systems with arbitrary levels of complexity become possible. Multi-threaded Web applets and distributed applications become simpler to design and implement, race hazards never occur, difficulties such as starvation, deadlock and livelock are easier to confront and overcome; and performance is no worse than that obtained from directly using the raw monitor primitives.

The advantages of teaching parallelism in Java purely through the CSP class libraries will be discussed. These libraries were developed jointly at Kent and Oxford Universities in the U.K. and the University of Twente in the Netherlands.

This paper was developed from material first presented at the *Java Threads Workshop* [1]. It presents the basic threads model of Java, discusses why they may be a good thing but why they need special care in their management, runs through the monitor mechanisms provided in Java for their control and points out weaknesses in that control. Finally, the CSP primitives are introduced and the case for ignoring the monitor methods presented.

This work is one of the foundation stones of the *JavaPP* project [2], which spun out from the above workshop. The other founding stones [3, 4, 5, 6] were first presented at the WoTUG-20 conference last year. The latest version of the JCSP library, developed from the work described in this paper, may be downloaded from [7].

1 The world is parallel – enjoy

1.1 Concurrency is everywhere

The natural world is certainly not organised through a central thread of control. Things happen as the result of the actions and interactions of (unimaginably) large numbers of independent agents, operating at all levels of scale from nuclear to astronomic. Computer systems aiming to be of real use in this real world need to model, at the appropriate level of abstraction, that part of it for which it is to be of service. If that modelling can reflect the natural concurrency in the system, it ought to be much simpler

Yet, traditionally, concurrent programming is considered to be an advanced and difficult topic – certainly much harder than serial computing which, therefore, needs to be mastered first. But that tradition is wrong.

Java recognises this, a little bit, by binding some notions of concurrency into the language, rather than leaving it to semantic-busting changes introduced through some external library. We will quarrel with the antiquity and semantic imprecision (so far) of some aspects of the Java thread primitives, but at least they give us a platform on which something sound and easy-to-use *can* be built.

Java threads are not exactly highlighted in the current range of books. It is easy to get the impression that they are a somewhat exotic feature with only a narrow field of application. Warnings are frequently given as to the difficulties in their safe use – we are advised to use them only when we absolutely can't get away with not using them! This does not square with our view that concurrency is a powerful abstraction that *simplifies* the design and implementation of systems, to be used every day on a broad field of application.

1.2 Threads : problems and opportunities

A 'normal' program has a single thread of control. Instructions execute one at a time in the order written in the code and dependent upon the state of the data being processed. A 'concurrent' program has many threads of control whose instruction sequences proceed 'simultaneously'. Each thread may be executed by a different processor (e.g. on an SMP platform) and each processor may have a different speed. More commonly, all the threads may be scheduled on a single processor, with their individual flows of logic arbitrarily interleaved. Either way, no assumptions can be made about the relative rates of progress of individual threads.

Threads also live in a shared memory space. They may declare and manipulate their own data and, optionally, make them available to other threads (although this is not usually a good idea). They may also operate on shared (external) data structures. In which case, some rules for coordinating access need to be designed *and enforced* if chaos is to be avoided. For example, it is not a good idea to allow two threads to update the same piece of data at the same time.

There are many ways to arrange this coordination, but all of them expose new dangers. For example, a group of threads may get into a state where each one is waiting for another to do something that will enable it to proceed – **DEADLOCK**! Or a group of threads may get into an infinite cycle of interaction amongst themselves and refuse to respond to anything outside their group – **LIVELOCK**! Or a thread may get blocked forever, waiting for a condition that the other threads never set up for it – **STARVATION**! And if we avoid all this, we can still get the coordination wrong and, intermittently, allow uncontrolled access to some shared resource and get unpredictable system corruption – **RACE HAZARD**!

On the other hand, if we can master these problems, systems do become simpler – with benefits across the whole engineering life-cycle. We can also achieve significant gains in performance and responsiveness through the direct targeting of parallel hardware. These include not only shared-memory (or virtual shared-memory) multi-processors, but also parallel use of a single processor (for computation) with communication hardware (e.g. for internet links).

Of course, no performance benefits will arise if the overheads of multi-threading (e.g. for coordination and context switching) swamp the processing the user actually wants the system to do. The *faster* our processors become, the *lower* must be these overheads. In 1987, the T800 transputer was the fastest microprocessor in the world, with a one microsecond floating-point time and a one microsecond (hardware assisted) context switch. In 1997, processors can be hundreds of times faster at floating-point, but fifty times slower at context-switching – even with 'lightweight' threads mechanisms! Something has gone wrong.

1.3 Some objects are more oriented than others

There is a superficial resemblance between the concepts of *object* (as provided by O-O languages) and *process* (as given by the occam/CSP model[8, 9, 10, 11, 12, 13]). Both concepts encapsulate data structures and the algorithms to manage those data structures – but:

- in occam/CSP, the algorithms form an autonomous thread of control that directly expresses the behaviour of the object *from its own point of view*;
- in Java (and most other O-O languages), the algorithms are implemented as *methods* which need to be invoked by some external agent (another object). This invocation forms part of that external agent's thread of control. The algorithms express the behaviour of their object *from the point of view of the calling agent*.

So, we reach the curious conclusion that, until the concepts of thread and object are bound together, object algorithms are not really *object-oriented* in the natural language sense of the term, but are *caller-oriented* – in the same way that procedures, in a traditional language such as Pascal, are caller-oriented.

1.4 Get a life

The magic ingredient that gives us the useful paradigm shift is the thread. In occam/CSP, this is automatic – processes are *live* objects executing their own logic via their own threads of control. In Java, if we want to give life to our objects, we have to do some extra work to set this up – but, at least, it can be done and it's not really that hard.

1.5 Living together

However, a system cannot just consist of active objects. They need to collaborate to get the overall job done. They can't simply call each other's methods to determine and/or alter each other's state – that leads to race hazards and unpredictable behaviour. Some ways to coordinate these interactions need to be found.

In occam/CSP, processes interact with each other via *channels*. Channels are *passive* objects – they have no life of their own and are just there to be used by the processes they connect. They provide a communication mechanism between processes that is *synchronised*.

A process may write to a channel at any time, but has to wait for some other process to read from that channel before it can continue. Similarly, a process may read from a channel at any time, but has to wait for another process to write before it can continue. Whoever gets to the channel first – the writing process or the reading one – is *blocked*. Blocking is entirely passive, a blocked process consuming no processor time.

Note that a channel is used just for synchronisation and communication – it does not buffer the data in transit. A good implementation will cause the data (or, possibly, a reference to the data) to be transferred directly from the writing to the reading process. Channels may be 1-1 (private line), any-1 (multiplexing), 1-any (de-multiplexing) or any-any (public exchange).

Notice also that it is entirely up to the process whether it chooses to read or write from a channel. For example, if some server process has run out of a crucial ingredient it needs to provide its service, it just doesn't read from the channel along which requests arrive. This leaves clients neatly and passively queued on the request channel, from which they can easily be retrieved (once the missing ingredient has been replenished) simply by performing the necessary reads.

In Java, we don't have channel primitives but we have plenty of passive objects – these are 'normal' objects that do not contain any threads. Happily, Java provides enough synchronisation mechanisms to allow us to build channels out of passive objects. We have done this – but it was not easy ...

1.6 The trouble with monitors

The Java threads model is (somewhat loosely) based upon the concepts of *monitors* and *condition-variables*. These were developed in the early 1970s (by Dijkstra, Brinch-Hansen, Hoare and others) as a structured way of using the more primitive notions of *semaphores*. The standard reference quoted by Sun is [14].

Nevertheless, monitors and condition-variables are not easy to use. Their semantics are volatile and do not compose. Individual methods have to be sensitive to the behaviour of their siblings, upon which they rely to establish conditions that will allow them to proceed if they get stalled. To write and understand one (synchronised) method, we need to write and understand all the (synchronised) methods at the same time – we can't knock them off one by one! To develop n methods, we need to consider $O(2^n)$ possible interactions. This type of logic does not scale and is not for everyday work.

In the late 1970s, Hoare broke entirely new ground with CSP. Some authoritative sources claim that Java threads are based upon CSP, but this is plainly wrong. Would that it were true! CSP semantics are denotational – a mathematical term for **WYSIWYG**. As we have noted, CSP processes can refuse individual events if they are not in a state to accept them. Each process has its own contract and looks after itself. This type of logic does scale – we don't have to understand the whole in order to understand the part. The whole *is* the orthogonal composition of the parts.

2 Synchronising parallel threads – the Java primitives

A thread starts, runs for a while (possibly forever) and then finishes. Whilst it is running, it may get *blocked* for various reasons. Sometimes this is voluntary – for example, it may just choose to sleep for some period (to allow other threads a greater share of the processor or simply because its next actions aren't yet due). Sometimes this is forced – because a resource that it needs (like some user input or buffer space or some data being calculated by another thread) is not yet available.

Some texts distinguish between two types of 'running' threads: those that are actually being executed (on a uni-processor, there will be just one of those at any one time) and those that are able to run but are waiting their turn for the processor (normally there will be many more threads than processors and the interleaving of runnable threads on any processor is managed by the underlying threads kernel). However, systems should be designed that are independent of this underlying scheduling, so we make no such distinction.

This tutorial is about ideas (semantics) rather than detailed syntax. For a formal presentation of the Java thread primitives, see one of the many textbooks (such as [15]). We assume *a little* familiarity with the Java syntax for classes, methods and flow-control, but will generally be fairly relaxed.

2.1 Basic threads in Java

A *thread* is created when we declare an instance of a class extending the built-in Java class `Thread`. For instance:

```
class Thing extends Thread {
  ...  attributes
  ...  constructor and other method(s)
  public void run () {
    ...  do things
    suspend ();
    ...  do some more things
  }
}
```

If we now declare an instance and start it:

```
Thing thing = new Thing ();
thing.start ();              // thing starts to run
```

the `run` method of `thing` will start to run, logically in parallel with the thread that has just declared it. Given the way it has been programmed, the newly spawned thread will do some things and then, voluntarily, `suspend` itself. If no other thread does anything to `resume` it, it will stay in that blocked state forever.

Suppose our spawning thread continued:

```
thing.start ();              // thing starts to run
...  do our own stuff
thing.resume ();             // kick thing back to life (maybe)
...  carry on                // is thing running again?
```

After spawning thread `thing`, we spend some time doing 'our own stuff' and, then, execute `thing.resume`. There are two possibilities: *either* `thing` has reached and executed its `suspend` instruction (in which case, it will be properly resumed) *or* it has not (in which case, nothing happens). In the latter case, `thing` will eventually suspend itself. The problem is that, in our code marked 'carry on', we have no idea whether the thread `thing` is running or not – presumably, we wanted it to be running! This is an example of a *race hazard*: the state of the system is not being controlled by us but by the rates of progression of the various threads (over which we do not have absolute control).

If we know something about the scheduling of the spawning and spawned threads, and about the relative timings of their 'do things' and 'do our own stuff' code fragments, we might be able to deduce that the `suspend` happened before the `resume` and that all is well. If the timings worked out wrong, we could try to fix things by adding a delay:

```
thing.start ();              // thing starts to run
...  do our own stuff
sleep (2*seconds);
thing.resume ();             // kick thing back to life (maybe)
...  carry on                // is thing running?
```

but this is desperate stuff! The problem is that there is no notion of *synchronisation* built into the `suspend` and `resume` thread methods and, without it, we can't keep control of what's going on in a way that is robust and independent of the underlying threads management. We don't recommend the use of `suspend`/`resume`.

[*Aside:* there is another way to create a thread in Java. If we want an object to run a thread *and* extend some other class, we can't proceed as above. Java does not allow multiple inheritance – we can't extend from both the `Thread` class and some other one. To get around this, extend from that other class and say that our extended class also `implements` an `interface` called `Runnable` (which just contains the `run` header). To create the thread, construct a raw `Thread` object passing in an instance of our `Runnable`. The information in this tutorial is independent of whichever way of creating threads is used.]

2.2 Another reason to synchronise

Suppose we have a simple Counter class:

```
class Counter {
  private long count = 42;
  public void increment () {
    count++;
  }
  public void decrement () {
    count--;
  }
  public long value () {
    return count;
  }
}
```

and a particular instance:

```
Counter x;
```

Suppose we have two threads operating on the same `x`. Suppose they make calls whose interleaved sequence is indicated by:

```
      .                42                .
      .                42                .
      .                42         x.increment ()
      .                43                .
      .                43                .
 x.increment ()        43                .
      .                44                .
      .                44                .
      .                44         x.increment ()
      .                45                .
      .                45                .
 x.decrement ()        45                .
      .                44                .
      .                44                .
```

where time flows down the page, the left column indicates when the first thread invoked `x`-methods, the right column indicates when the second thread invoked them and the middle column traces the value of the private `x.count`.

Because the calls did not overlap, the state of the counter was updated correctly. However, we can't assume the calls will never clash. Suppose the scheduling on the first thread was a little earlier:

```
      .                42                .
      .                42                .
 x.increment ()        42         x.increment ()
      .                43                .
      .                43                .
      .                43                .
      .                43                .
      .                43                .
 x.decrement ()        43         x.increment ()
      .                42                .
      .                42                .
```

This time, the two threads attempted to increment the count at (around about) the same time. What happened is that one thread had loaded the value `42` into its processor's registers when the processor switched context to the other thread. The second thread completed its increment operating on the `42` value still in memory, writing `43` back. Context switched back again to the first thread, which completes its interrupted increment using the `42` value it had originally loaded, and also writes `43` back into memory. Two increments happened, but the value in `x.count` only went up once.

Later, there was another clash between an increment and a decrement. In this case, the decrement was the interrupted operation, finished last and left `42` back in `x.count`. It could equally have happened the other way around, leaving a final value of `44`.

These errors are also the result of race hazards due to a lack of synchronisation between the threads.

2.3 Monitors

Java provides a synchronisation mechanism based upon the concept of *monitors*. A monitor is a class whose methods can only be executed by one thread at a time. It enables atomic (i.e. safe) updating of data within the monitor by any number of threads. In Java, all we do is add one keyword to the method declarations:

```
class Counter {                       // this is a monitor

  private long count = 42;

  public synchronized void increment () {
    count++;
  }

  public synchronized void decrement () {
    count--;
  }

  public synchronized long value () {
    return count;
  }

}
```

In this case, each instance of a `Counter` object has a *lock* which has to be acquired by any thread invoking one of its methods. Only one thread can hold this lock at any one time. So, if one thread is in the middle of `x.increment` when another thread calls `x.decrement`, the second thread will have to wait until the first call has finished.

So, with the clashing scenario that caused data corruption before, what would happen this time is:

```
      .                    42                   .
      .                    42                   .
  x.increment ()           42              x.increment ()
      .                    43                <block>
      .                    44                   .
      .                    44                   .
      .                    44                   .
      .                    44                   .
  x.decrement ()           44              x.increment ()
      .                    43                <block>
      .                    44                   .
      .                    44                   .
```

where the first thread acquired the lock first each time and the second one had to wait. But the `x.count` would have been securely updated whichever thread had won the race – there is no hazard. So far so good!

One final point about Java's monitors: what happens when there are *lots* of threads vying for the lock at the same time? The Java language definition is imprecise here. What happens is left to the threads management kernel that underlies the Java Virtual Machine (JVM). In most implementations, the threads form an orderly queue for the lock. In that case, so long as no method executes forever and so long as no deadlock occurs, each thread is bound to reach the front of that queue and invoke the method on the monitor – no *starvation* here. However, we should note that this simple and fair mechanism is not guaranteed by the Java language.

2.4 Conditions

Sometimes we queue up for something only to find, when we finally get served, that they've run out of the particular item we wanted! Well, we can always go away and try again later. But that gives no guarantee that we'll ever get back at the right time to find what we wanted – a danger of *livelock* and *starvation*.

Some shops provide a *stand-by* area where we can wait if the item we want isn't available. When a delivery arrives, we can be notified to come and get it.

These stand-by areas are called *condition variables* in monitor theory. Hoare's paper allowed for as many (named) condition variables as the designer of the monitor wanted, but Java allows only one (and, therefore, it is unnamed).

There is another difference between Hoare monitors and those in Java. Hoare specifies that when a waiting thread is notified, it *immediately* acquires control of the monitor lock (from the notifier) and resumes execution. This is only reasonable, since the reason it was notified is that the condition for which it was waiting has been established. It needs to be able to exploit that condition before some other thread grabs the lock, consumes some resource and negates it. Otherwise the notification will have been in vain!

But there is no such immediacy from Java - in fact, notifiers retain the lock. Worse, many JVMs send the notified thread from the stand-by area to the *back* of the queue for the lock – a queue it has already been through once! By the time it gets back to the front of this queue, the condition which triggered its notification may no longer apply and it has to go back on stand-by. As we shall see, this can cause serious trouble.

Java provides three forms of waiting and two kinds of notification. They can only be invoked once the calling thread is in possession of the monitor lock for that object (e.g. inside a `synchronized` method for that object):

- `wait ()`

  The calling thread is blocked and the monitor lock released. If there are several threads that get blocked like this for the same object, Java does not say how they are managed. Many implementations hold them in a queue, but it is not safe to rely on this. When some other thread invokes a `notify` method on this object, *one* of the waiting threads is released – hopefully, the one that has been waiting the longest. The released thread has to re-acquire the monitor and many implementations just put it to the back of the queue of all threads trying for this lock. Meanwhile, the notifying thread retains the lock and carries on.

- `wait (long timeout)`

  This is the same as the above, except that after `timeout` milliseconds without release-by-notification, the thread is released anyway (typically, to the back of the monitor queue).

- `wait (long timeout, int titch)`

  This is the same as the above, except that the timeout period is `timeout` milliseconds plus `titch` nanoseconds. The value of `titch` should be in the range `0` through `999999`.

- `notify ()`

  This releases *one* of the waiting threads – see the remarks for `wait` above. If there are no waiting threads, nothing happens.

- `notifyAll ()`

  This releases *all* the waiting threads (if any).

  [*Aside:* this method is *not* envisaged in Hoare's paper on monitors [14]. It somewhat clobbers the key property that Hoare had specified that notified threads immediately take over the monitor lock and resume execution – they can't *all* do that! Current recommended Java practice is to use `notifyAll` whenever possible – not `notify`. We shall ignore this recommendation.]

2.5 Example : a simple FIFO buffer

Consider a passive object that provides FIFO buffering between active writer threads and active reader threads. Its top-level structure is:

```
class Buffer {

  private final int max = 100;  // buffer size

  private final int[] buffer    // space to hold the buffered data
    = new int[max];

  private int size = 0;         // number of items currently held
  private int lo = 0;           // index of oldest item in the buffer
  private int hi = 0;           // index of next free slot in the buffer

  public synchronized int read () {...}

  public synchronized void write (int n) {...}

}
```

This is a simple integer buffer with a fixed size (100). It could easily be modified to buffer arbitrary `Objects` and have a user-chosen size, but that's not relevant to our problem.

The private data structures are guaranteed atomic update by the `synchronized` methods that operate on them. But what happens when a reader thread acquires the monitor lock and finds the buffer is empty? Rather than giving up and exiting, the reader can `wait` in the stand-by area:

```
  public synchronized int read () throws InterruptedException {
    int index;
    if (size == 0) wait ();
    // hopefully, size is now greater than zero :-)
    index = lo;
    lo = (lo + 1) % max;
    size--;
    return buffer[index];
  }
```

[*Aside:* just ignore the `throws InterruptedException` stuff! This exception *may* be thrown by the `wait` method we are about to use. Rather than handle this here, we are throwing it back to the thread that called this `read` (who may know something sensible to do with it!)]

Executing the `wait` releases the monitor lock and this thread becomes blocked. Any other reader threads invoking this `read` method will similarly be put on hold. Hopefully, a writer thread will come along, put something into the buffer, increment the `size`, update the `hi` pointer and issue a `notify`.

Issuing a `notify` causes one of the waiting reader threads to move out of the stand-by area and resume execution. However, first it has to re-acquire the monitor lock ... and current implementations make it queue up again! If it finds itself behind another reader and behind no other writers, by the time it gets back the buffer will be empty again.

Consequently, Java authorities (such as [15]) tell us never to wait for a condition using an `if`, but to use a `while`:

```
public synchronized int read () throws InterruptedException {
  int index;
  while (size == 0) wait ();
  // if we ever exit the above, size *will* be greater than zero :-)
  index = lo;
  lo = (lo + 1) % max;
  size--;
  return buffer[index];
}
```

The problem with the above is proving we will ever exit the `while`-loop. If there is only one reading thread, we will go round at most once. But if there are more than one, we have no guarantees and just have to hope for the best!

There are other problems. Viewing the above method in isolation, it makes no sense. The semantics of the given `while`-loop implies no exit, since nothing happens in the loop body to alter the `while`-condition. Of course, we can't reason about this method in isolation – it depends intimately on its sibling `write` method:

```
public synchronized void write (int n) throws InterruptedException {
  while (size == max) wait ();
  // if we ever exit the above, there *will* be room in the buffer :-)
  buffer[hi] = n;
  hi = (hi + 1) % max;
  size++;                 // implies size > 0
  notify ();              // in case there is a waiting reader
}
```

Executing the `notify` method summons one of the waiting reader threads out from the stand-by area. If there were no waiting threads, nothing happens – the unnecessary call is said to be *benign*. Notice that we issue the `notify` immediately after doing something that ensures that the condition being waited for (in this case, that `size` is non-zero) has been established.

Note also that a writer thread, having acquired the monitor lock, will also have to `wait` if the buffer is full. This means that we need to go back to the `read` method and get it to issue a `notify` in case there was a waiting writer:

```
public synchronized int read () throws InterruptedException {
  int index;
  while (size == 0) wait ();
  // if we ever exit the above, size *will* be greater than zero :-)
  index = lo;
  lo = (lo + 1) % max;
  size--;                // implies size < max
  notify ();             // in case there is a waiting writer
  return buffer[index];
}
```

[*Aside:* a simpler (one-place) version of this `Buffer` class is given in Sun's on-line tutorial [16]. Their class is called `CubbyHole` and is programmed in an identical style to the version of our `Buffer` we have reached so far. And it has the same deficiencies.]

The continual juggling between the two methods to get their mutually dependent semantics correct is not good news – and we are not finished yet!

When the reader issues the `notify` and there *was* a waiting writer, what ought (arguably) to happen is that the reader immediately relinquishes the monitor lock to the notified writer (because the condition for which it was waiting has been set up). If that happened in this case, the notified writer would immediately fill up the last place in the buffer. Unfortunately, that would overwrite the `buffer[index]` this method still has to return!

However, that doesn't happen in Java – the notified writer is released to fight again for control of the monitor, whose lock is retained by the reader thread which carries on executing. So, the above code looks OK. If we are paranoid, we may prefer to be independent of this aspect of Java's semantics. Rather than remembering the *index* to the item we are going to be returning, we could save the whole thing:

```
public synchronized int read () throws InterruptedException {
  int save;
  while (size == 0) wait ();
  // if we ever exit the above, size *will* be greater than zero :-)
  save = buffer[lo];
  lo = (lo + 1) % max;
  size--;                // implies size < max
  notify ();             // in case there is a waiting writer
  return save;
}
```

The `save` variable is local to this method and will be a different variable for each invoking (but stalled) thread. That fixes that.

However, there is one last thing that is unsatisfactory about the above code. I really don't like the calling of `notify` when there are no waiting threads. It's not just the unnecessary overhead – it's the lack of synchronisation involved (a bit like the way `resume` works on threads that have not executed a `suspend`). Such sloppiness will lead to trouble in the long run.

In this case, the sloppiness can be removed by maintaining `private` counts (inside the `Buffer` object) of the number of waiting readers and writers. These could be incremented just before executing a `wait` and decremented just before executing a `notify`. The nice thing is that the call to the `notify` can then be made dependent on whether there is any waiting thread. This is left as an exercise.

Reminder: the livelock/starvation problems associated with completing the `while`-loops that are waiting for the conditions (in the case of multiple readers and writers) have not yet been addressed.

2.6 Wot, no chickens?

This example illustrates the danger of simply `wait`ing for a condition in a `while`-loop when there are multiple consumers of that condition. It is described in detail in [1] and outlined here.

A college consists of five philosophers, a chef and a canteen – see Figure 1. The chef and the philosophers are *active* objects. The canteen is a *passive* object through which the philosophers and the chef interact. The canteen is implemented in the style of the above `Buffer` class (and Sun's `CubbyHole`).

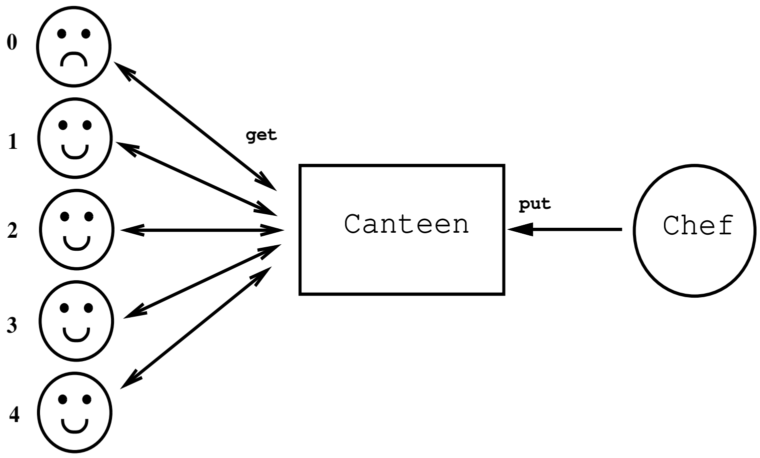

Figure 1: The greedy but starving philosopher

The philosophers think for a while and then go to the canteen for food. Except for one of them ... who is plain greedy, never thinks and just keeps going to the canteen.

The chef cooks in batches of four, replenishing the canteen when each batch is ready. The greedy philosopher always misses out! He gets there too early the first time (no food yet) and is put on hold. When released, he is put on the *back* of the canteen queue again ... behind his colleagues who arrived whilst he was on hold. His colleagues take the whole batch just arrived from the kitchen and he gets put on hold again. He never gets out of this cycle.

This infinite starvation and livelock does depend on bad luck with the timing. But that is typical of race hazards and we cannot rely on good luck in order to be safe. The timings are as follows:

- the philosophers and the chef start up around the same time;
- philosopher 0 thinks for no time at all;
- the other philosophers think for 3 seconds before going to eat;
- the chef cooks chickens in batches of 4 and takes 2 seconds per batch. Delivery to the canteen takes around 3 seconds (the chickens are hot!) and the chef helps to set them down. During this time, the canteen cannot accept further orders from philosophers (who will have to queue) because its lock is owned by the chef.

What happens is:

- *second 0:* the system starts up. Philosopher 0 calls `Canteen.get` and is made to `wait` since there are no chickens yet. The chef starts cooking and the other philosophers start thinking.

- *second 2:* the chef calls on `Canteen.put` with 4 chickens. Setting them down is going to lock up the canteen for the next 3 seconds.
- *second 3:* the other 4 philosophers try to call `Canteen.get`, but cannot acquire its monitor and have to get in line.
- *second 5:* the chef finally sets down the batch of 4 chickens and calls `notifyAll` to release anyone `waiting`. In this case, it's just philosopher 0 and (on many perfectly legal JVMs) he has to go to the back of the line.

  The chef exits the `Canteen.put` method which allows the philosophers to make their `Canteen.get` calls. Philosophers 1 through 4 go through in sequence, taking all the chickens. Finally, philosopher 0 regains the monitor – but (*"Wot, no chickens?"*) is put on hold again!

  The chef starts cooking again and the other philosophers start thinking. We are back to the state we were in at *second 0* and everything repeats – forever! Greedy philosopher 0, despite being first to the canteen, never gets served.

Here is the code for the `Canteen`:

```
class Canteen {

  private int nChickens = 0;

  public synchronized int get (int id) throws InterruptedException {
    while (nChickens == 0) {
      ...  complain ("Phil " + id + ": Wot, no chickens?")
      wait();
    }
    ...  place order ("Phil " + id + ": Those chickens look good ...")
    nChickens--;
    return 1;
  }

  public synchronized void put (int value) throws InterruptedException {
    ...  shout ("Chef : Make room ... this dish is very hot ...")
    Thread.sleep (3000);     // take 3 seconds to set down the dish
    nChickens += value;
    ...  announce ("Chef : More chickens ... NOTIFYING ...")
    notifyAll ();            // wake up any waiting philosophers
  }

}
```

where the doted lines indicate some statements for animating what's happening.

This code follows the same style as Sun's `CubbyHole` and gives little indication of the danger to which its callers are exposed. As recommended, we have used `notifyAll` rather than `notify`, but it makes no difference in the scenario we have set up – there is only ever *one* thread `waiting` for the chickens (philosopher 0).

What is the matter with this design? Clearly, the chef should not have to queue up with the philosophers to get into the canteen. The philosophers should have their own queue and the chef should only have to contend with one philosopher when dealing with the chickens in the canteen. However, we are modelling this `Canteen` on the `CubbyHole` class (to show the danger of starvation it contains) and `CubbyHole` has its queue shared by both readers and writers.

A more serious complaint – from the point of view of *object-orientation* – is this. All methods, apart from a `run` method, are executed as part of the thread of control of their calling objects. This is particularly clear here, where we see them speaking *as* those calling objects. For instance, the `Canteen.get` method is part of the life of the calling philosopher and `Canteen.put` is part of the life of the chef. So, this canteen *object* has bits of philosopher-algorithm and bits of chef-algorithm for its methods – a somewhat confused set of roles for something that's supposed to be a canteen. There's nothing in these methods that are oriented towards the life of the canteen.

These problems are not special to this `Canteen`, but are universal across all *passive* objects.

The canteen should, of course, be implemented as an *active* object connected to the philosophers and the chef via simple channels. Then, we wouldn't have the problems arising from this design. This comes next!

3 Synchronising parallel threads – the JavaPP primitives

This section introduces the basic JavaPP (i.e. occam/CSP) synchronisation primitives. These can be used exclusively to provide the *glue* to coordinate and exchange information between active threads – we never need to get involved again with `synchronized` methods or `wait` and `notify`.

Armed with them, multi-threaded code becomes **WYSIWYG** and system complexity can be ramped up with linear increase in effort. No run-time overheads are imposed that wouldn't be needed anyway to prevent race hazards. We can inherit the rich treasury of occam/CSP design and analysis methods/tools. Multi-threaded systems can be structured to reflect real world system hierarchies. Components become automatically thread-safe and reusable and the nasty accidents of deadlock, livelock and starvation can be ruled out by design.

But first we have to implement the primitives. These will be passive objects in the normal sense and we will have to do the hard work – *one last time* – of safely using the raw Java monitors.

3.1 Channels : review

Channels were described in Section 1.5. They provide synchronised (un-buffered) communication between a pair of threads – except that, from now on, we are going to refer to threads as *processes* (out of deference to CSP).

Recapping what was said earlier, a process may write to a channel at any time, but has to wait for some other process to read from that channel before it can continue. Similarly, a process may read from a channel at any time, but has to wait for another process to write before it can continue. Whoever gets to the channel first – the writing process or the reading one – is *blocked*. Blocking is entirely passive, the blocked process consuming no processor time.

3.2 Channels : first attempt

For the moment, we are going to restrict ourselves to a channel that only allows integers to be communicated – extending this to carry arbitrary objects is trivial. We also restrict the channel to support only a *single* reader and a *single* writer. It was *multiple* readers and writers that gave rise to the race hazard that caused infinite starvation when using the `Buffer` class above. We shall postpone solving that problem for a short while.

Here is a first attempt at the channel. The top-level structure is:

```
class Channel {

  private int channelHold;
  private boolean channelEmpty = true;     // synchronisation flag

  public synchronized int read () throws InterruptedException {...}

  public synchronized void write (int n) throws InterruptedException {...}

}
```

The `channelHold` attribute holds the value of the data being communicated through the channel. This is transient and the users of the channel cannot detect that it is buffered here temporarily. [*Aside:* for a channel carrying objects, `channelHold` will just contain a *reference* to the object being transmitted.]

The `channelEmpty` flag records whether the channel has a (reader or writer) process waiting on it.

The methods are short and sweet – but kind of tricky:

```
public synchronized int read () throws InterruptedException {
  if (channelEmpty) {
    channelEmpty = false;          // first to the rendezvous
    wait ();                       // wait for the writer process
  } else {
    channelEmpty = true;           // second to the rendezvous
    notify ();                     // schedule the waiting writer
  }
  return channelHold;
}
```

So, if the reader is first to the channel, it sets the flag and waits. It will be released later by the writer process (see below), which has first cleared the flag and written into `channelHold`. Once released, all the reader has to do is return this value.

If the reader is second to the channel, the writer has already written into `channelHold` and set the synchronisation flag. We clear the flag, notify the writer to be released and return the value that has been written.

Nearly-but-not-quite symmetrically, the writer is implemented:

```
public synchronized void write (int n) throws InterruptedException {
  channelHold = n;
  if (channelEmpty) {
    channelEmpty = false;          // first to the rendezvous
    wait ();                       // wait for the reader process
  } else {
    channelEmpty = true;           // second to the rendezvous
    notify ();                     // schedule the waiting reader
  }
}
```

The writer can always write, straightaway, into `channelHold` – it doesn't matter whether the reader got there first or not.

Next, if the writer is first to the channel, it sets the flag and waits. It will be released later by the reader process (see above), which first clears the synchronisation flag. Once released, all the writer has to do is exit.

If the writer is second to the channel, then the reader is waiting. We clear the synchronisation flag, notify the reader to be released and exit.

3.3 Channels : paranoia

The above seems to work for current implementations ... but we need to be more careful! There is a potential danger in the reader code. In the case that it was first to the rendezvous and waiting, what does the writer process do after notifying the reader? Well, it exits and could immediately have something else to write down this channel and issue another `write`. If by any chance that second write got in the queue to the channel monitor *before* the reader it just notified, the data from the first write – which the reader still has to return – will be lost! This is an unlikely, but possible, way that the underlying threads mechanism may sometimes schedule things. It is necessary to rule it out.

There is a potential for similar danger if the reader was second to the rendezvous and has to notify the waiting writer. Invoking `notify` does not hand over the monitor lock to the process being notified (so that it can exploit the condition that caused us to issue the notification). If it did, the notified writer would simply exit ... and we would be back with the scenario from the previous paragraph.

Let's be paranoid and guard against both the real and potential dangers. If the reader were second to the rendezvous, we can take another copy locally (within the `read` method) before notifying the waiting writer and return that local copy. If the reader were first to the rendezvous and waiting, let's make sure it regains the monitor *before* the notifying writer exits its `write` method – we can do this simply by forcing the writer to wait after it notifies the reader. Then, it is as though the reader had been *second* to the rendezvous (i.e. we take a local copy, notify the waiting writer and return the local copy). The reader and writer methods become:

```
public synchronized int read () throws InterruptedException {
  int local;
  if (channelEmpty) {
    channelEmpty = false;          // first to the rendezvous
    wait ();                       // wait for the writer process
    local = channelHold;           // take a copy of what was written
    notify ();                     // schedule the writer to finish
  } else {
    channelEmpty = true;           // second to the rendezvous
    local = channelHold;           // take a copy of what was written
    notify ();                     // schedule the waiting writer
  }
  return local;
}
```

and:

```
public synchronized void write (int n) throws InterruptedException {
  channelHold = n;
  if (channelEmpty) {
    channelEmpty = false;          // first to the rendezvous
    wait ();                       // wait for the reader process
  } else {
    channelEmpty = true;           // second to the rendezvous
    notify ();                     // schedule the waiting reader
    wait ();                       // let the reader regain the lock
  }
}
```

Phew!

3.4 Channels : multiple readers and writers

We now extend the channel so as to make it secure for use by multiple readers and writers. We still want full synchronisation between a reading and writing process. Any process may read or write on this channel. Readers and writers are queued separately. A reader only completes when it gets to the front of its queue and finds a writer. A writer only completes when it gets to the front of its queue and finds a reader. There is no logical buffering of data in the channel.

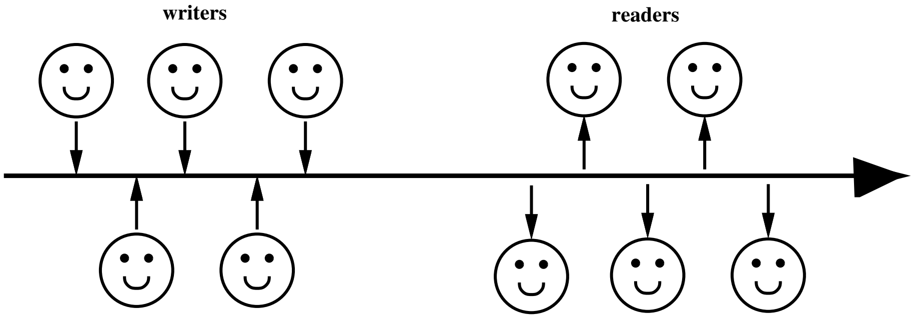

Figure 2: A shared channel

Figure 2 illustrates the connection: the smiling faces are *active* processes and the thick arrow (plus its stubs) represents the *passive* object that is the shared channel.

The previous version of `Channel` guarded its `waits` with a simple `if`-condition. We did not need to use a `while` – as recommended by many textbooks – because we only had two processes to worry about (one reader and one writer). We do not want to use a `while` now because of the lack of a guarantee that it will ever exit.

Instead, we make the readers acquire their own *reader-lock* before trying to access the channel. We make the writers do the same for their own *writer-lock*. That way, we ensure that there is still only a single reader and a single writer actually competing to use the channel itself – and we can use the previous algorithm.

Every object in Java contains a monitor on which we can park processes simply by making them `synchronize` on it. The simplest way to do this is to make use of another facility of Java – the synchronising *block* (rather than synchronising *methods*). To explain what this is, consider a Java monitor method:

```
public synchronized int f () {
  ...  body of the method
}
```

This is equivalent to an *unsynchronised* method whose body is guarded by a *synchronise* command on the monitor object:

```
public int f () {
  synchronized (this) {
    ...  body of the method
  }
}
```

Of course, the synchronised block need not be the whole of the body:

```
public int g () {
  ...  stuff
  synchronized (this) {
    ...  exclusive execution here
```

```
      }
      ...  more stuff
    }
```

where only one process at a time can execute the synchronised code. We can also synchronise on *any* Java object, so long as we know its name, and we make use of this to construct our extended channel:

```
  class Channel {

    private int channelHold;
    private boolean channelEmpty = true;

    private final Object readMonitor   // all readers multiplex
      = new Object ();                 // through this

    private final Object writeMonitor  // all writers multiplex
      = new Object ();                 // through this

    public int read () throws InterruptedException {
      synchronized (readMonitor) {     // compete with other readers
        synchronized (this) {          // compete with a single writer
          int local;
          if (channelEmpty) {
            channelEmpty = false;      // first to the rendezvous
            wait ();                   // wait for the writer thread
            local = channelHold;       // take a copy of what was written
            notify ();                 // schedule the writer to finish
          } else {
            channelEmpty = true;       // second to the rendezvous
            local = channelHold;       // take a copy of what was written
            notify ();                 // schedule waiting writer thread
          }
          return local;
        }
      }
    }

    public void write (int n) throws InterruptedException {
      synchronized (writeMonitor) {    // compete with other writers
        synchronized (this) {          // compete with a single reader
          channelHold = n;
          if (channelEmpty) {
            channelEmpty = false;      // first to the rendezvous
            wait ();                   // wait for the reader thread
          } else {
            channelEmpty = true;       // second to the rendezvous
            notify ();                 // schedule waiting reader thread
            wait ();                   // let reader regain this monitor
          }
        }
      }
    }

  }
```

The methods are the same as before *except* that, before synchronising on the channel monitor itself, we have to synchronise on our respective (reader or writer) monitor. It is crucial, therefore, that the methods are not `synchronized` themselves.

3.5 Channels : choosing between them

So far, we have only shown how to read and write on channels. Sometimes, we want to wait on a number of channels and choose the first one that becomes *ready* – i.e. has a process trying to use it at the other end. This implies making some declaration of intent that we are willing to use *any* of the channels. When we find *one* that is ready, we need to withdraw that declaration of intent from the others and commit to the one selected.

Allowing both sides of a channel communication to back off such a declaration is possible, but requires considerable overheads for its secure management. Although CSP allows this, we adopt the same compromise made by the occam multiprocessing language and only allow *readers* the luxury of backing off. This constraint significantly reduces overheads and, as found in over a decade of industrial and academic practice, does not significantly limit our freedom in design.

We also insist that when a process is waiting on a bunch of channels, no other process may be reading from any of them. In other words, such channels may only have a single (potential) reader. They can, of course, have multiple writers.

To make a choice between a set of channel reads, the channels have to be held in a Java array. If they are individual channels, this is still easy to arrange – for example:

```
final Channel[] c = {panic, supply, service};
final int PANIC = 0, SUPPLY = 1, SERVICE = 2;   // indices to c
```

We also need an instance of a class we have called `Alternative` – this object is bound to the set of channels (or, in general, *guards*) from which the selection is to be made:

```
final Alternative alt = new Alternative (c);
```

where we refer to [1] for a description of its implementation. The term `Alternative` comes from the occam `ALT` constructor. We will sometimes refer to this choice mechanism as *alting*. We make our choice as follows:

```
int i = alt.select ();  // blocks until one or more channels are ready
int x = c[i].read ();   // i is the index of the selected channel
...  some process (which makes use of the information i and x)
```

If there are one or more writers who have already written to one or more of the channels, the `alt.select` will return immediately with the index of one of them. The selection can be *prioritised* so that, if there is a choice, it will choose the lowest index - or made *fairly*.

If there are no writers pending, the `alt.select` will block *passively* – consuming no processor time – until one or more writers appear. It will then behave as above.

After receiving the chosen index, we *must* read from the channel selected – if we don't, bad things will happen. After reading from the selected channel, we may engage in actions appropriate to the processing of the message and the channel from which it was received.

Often, the channel array will be small (and, if the channels are carrying Java 'Objects', each element can be communicating a separate protocol). In such cases, a `switch` statement may be more convenient:

```
switch (alt.select ()) {
  case PANIC: {
    status  = panic.read ();     // note: panic == c[PANIC]
    ...  take evasive action
    break;
  }
```

```
      case SUPPLY: {
        goods = supply.read ();      // note: supply == c[SUPPLY]
        ...  update accounts
        break;
      }
      case SERVICE: {
        order = service.read ();     // note: service == c[SERVICE]
        ...  process the order
        break;
      }
    }
```

We can also impose run-time determined conditions that can be used to *mask* out some of the channels over which we are alting. We just set up an array of boolean guards with the same size as the channel array. Then:

```
    int i = alt.select (guard);
    x = c[i].read ();
    ...  some process (which makes use of the information i and x)
```

What happens now is that if `guard[j]` is *false*, channel `c[j]` will be ignored when making this choice – even if it has a writer pending. Note that the `guards` are evaluated just *once* at the start of the selection – it would, of course, be a bad program that indulged in the race hazard of having another thread alter the values of these `guards` whilst this operation was being conducted!

Reference [1] gives a much fuller account of the material in this section, including the setting of timeouts on waiting for channels and the simple polling of channels.

3.6 Wot, no chickens : WYSIWYG and safe

Let's do the *"Wot, no chickens?"* example safely and simply, using channels to connect everything together. Figure 3 is a diagram of the network.

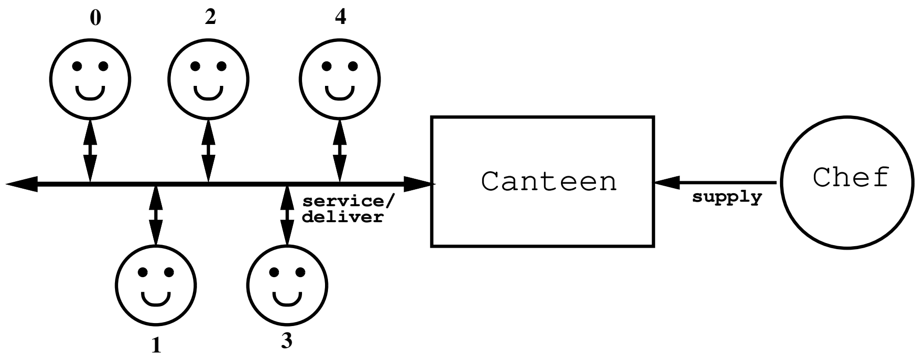

Figure 3: The greedy and non-starving philosopher and his mates

This time the canteen is an active object – a pure *server* process for its philosopher and chef *clients*. It listens on its `supply` and `service` channels, giving priority to the former.

The philosophers share a pair of channels – `service` and `deliver`. Philosophers eat chickens. They queue up at the canteen on the `service` channel to place their orders. The canteen refuses to accept any order when it has no chickens. This is a crucial difference from the previous version. The canteen is a live process and can make its own decisions and impose its will. Previously, the canteen was passive and the philosophers had to barge in, discover the absence of food (*"Wot, no chickens?"*) and then do something to sort out the resulting mess.

When (*and only when*) the canteen has some supplies, it will accept orders from `service` and dispense chickens down the `deliver` channel. It is always prepared to accept supplies from the `supply` channel.

The chef cooks chickens. When a batch is ready, she queues up at the canteen on its `supply` channel. Setting down the batch takes around 3 seconds and she is made to hang about until this has happened – this is to stay in line with the behaviour of the original system (that caused starvation). Here is the code for the chef:

```
class Chef extends Thread {

  private final Channel supply;

  public Chef (Channel supply) {
    this.supply = supply;
  }

  public void run () {
    try {
      ...  starting ("Hello from the chef ...")
      while (true) {
        int nChickens;
        ...  sing ("Frying tonite ...")
        ...  cook (takes around 2 seconds)
        nChickens = 4;
        ...  announce ("4 chickens, ready-to-go ...")
        supply.write (nChickens);    // supply the chickens
        supply.write (0);             // wait till they're set down
      }
    } catch (InterruptedException e) {}
  }

}
```

where we are obliged to put in an exception handler in case any of the channel methods fail.

This process repeatedly executes a typical *client* transaction: *it* decides when it wants to start the transaction (the first `supply.write`) and commits itself to complete the transaction (the second `supply.write`).

The philosopher processes also behave as *clients*:

```
class Phil extends Thread {

  private final int id;

  private final Channel service;
  private final Channel deliver;

  public Phil (int id, Channel service, Channel deliver) {
    this.id = id;
    this.service = service;
    this.deliver = deliver;
  }
```

```
    public void run () {
      try {
        ...  starting ("Hello from philosopher " + id + " ...")
        while (true) {
          int chicken;
          if (id > 0) {
            ...  think for around 3 seconds
          }
          ...  announce ("Phil " + id + " ... gotta eat ...")
          service.write (0);            // may have to wait here
          chicken = deliver.read ();    // never get blocked here
          ...  consume ("Phil " + id + " ... mmm ... that's good ...")
        }
      } catch (InterruptedException e) {}
    }

  }
```

The canteen process is a pure *server* – it never initiates anything external, but responds (depending on its state) to *client* requests:

```
  class Canteen extends Thread {

    private final Channel service;   // shared (any-1)
    private final Channel deliver;   // shared (but used 1-1)
    private final Channel supply;    // not shared (1-1)

    public Canteen (Channel service, Channel deliver, Channel supply) {
      this.service = service;
      this.deliver = deliver;
      this.supply = supply;
    }

    public void run () {

      try {

        final Channel[] c = {supply, service};         // alt channels
        final int SUPPLY = 0, SERVICE = 1;             // alt indices
        final Alternative alt = new Alternative (c);   // alt object
        final boolean[] guard = {true, false};         // alt guards

        int nChickens = 0;                             // variable

        ...  starting ("Hello from canteen ...")

        while (true) {

          guard[SERVICE] = (nChickens > 0);

          switch (alt.select (guard)) {

            case SUPPLY: {
              int value = supply.read ();        // new batch of chickens
              ...  ouch ("This dish is hot ...")
              ...  take 3 seconds to set down the dish
              nChickens += value;
              ...  announce (nChickens + " now available ...")
              supply.read ();                    // let the chef go
              break;
            }
```

```
          case SERVICE: {                      // ==> (nChickens > 0)
            service.read ();                   // philosopher waiting
            ...  announce ("One chicken coming down ...")
            deliver.write (1);                 // serve one chicken
            nChickens--;
            break;
          }
        }
      }
    } catch (InterruptedException e) {}
  }
}
```

Each `case` of the `switch` corresponds (roughly) with the synchronised `get` and `put` methods of the *passive* monitor in Section 2.6. However, this time there are no dangerous loops and *each case can be understood without reference to the other*.

Finally, the network itself is constructed with the following code (which could go inside the `main` Java method):

```
int nPhilosophers = 5;

Channel service = new Channel ();
Channel deliver = new Channel ();
Channel supply = new Channel ();

Canteen canteen = new Canteen (service, deliver, supply);
canteen.start ();

Chef chef = new Chef (supply);
chef.start ();

Phil[] phil = new Phil[nPhilosophers];

for (int i = 0; i < nPhilosophers; i++) {
  phil[i] = new Phil (i, service, deliver);
  phil[i].start ();
}
```

This time, even though philosopher 0 is just as greedy, nobody starves. Philosopher 0 arrives at the `service` channel first, but is blocked *there* since the canteen has no chickens. The chef delivers between seconds 2 and 5, locking up the canteen as it services the delivery. At second 3, the other 4 philosophers also arrive at the `service` channel and queue up *behind* philosopher 0[1] – in the previous version, the greedy philosopher was stuck in the stand-by area inside the canteen. At second 5, philosophers 0 through 3 get served (because only 4 chickens were delivered) and philosopher 4 remains in line. The next time more chickens arrive, philosopher 4 will be at the head of the line.

The system is *fair* and *deadlock/livelock/starvation*-free. The system conforms to well-defined *client-server* design rules for which there are CSP theorems [17, 18, 19, 20, 21, 22] that prove these somewhat essential properties.

[1] Such good behaviour depends on the underlying JVM *queuing* threads trying to acquire an already-locked monitor. For our implementation of a shared channel, this imposes a *queue* on processes trying to write (or read). Should the JVM *stack* such threads (which it is legally entitled to do), then the greedy philosopher will starve as before – but such starvation is always possible in such a regime. A good point about the channel mechanism is that queuing process access is *sufficient* for a base-level guarantee against starvation (i.e. so long as every communication on the channel succeeds, a queued process will eventually get to the front and communicate). As we saw earlier, queuing thread access to a Java monitor is *insufficient* to provide a similar guarantee! Even though that *monitor* version of the canteen was programmed using the approved design pattern, the greedy philsopher found himself relegated to the back of the queue each time he reached the front.

3.7 Buffers : WYSIWYG and safe

As our last example, we return to the `Buffer` from Section 2.5. Because of the restriction to alting only over channel reads, the consumer of this `Buffer` has to make a `request` before it can take anything. Notice the guards on selection that refuse consumer requests when the buffer is empty and refuse supplier inputs when the buffer is full.

```
class Buffer extends Thread {

  private final Channel in;         // could be shared (any-1)
  private final Channel request;    // could be shared (any-1)
  private final Channel out;        // could be shared (but used 1-1)

  private final int[] buffer;       // space to hold buffered data
  private final int max;            // size of buffer

  public Buffer (int max, Channel in, Channel request, Channel out) {
    this.max = max;
    this.in = in;
    this.request = request;
    this.out = out;
    buffer = new int[max];          // assume : (max > 0)
  }

  public void run () {

    try {

      int size = 0;          // number of items currently held
      int lo = 0;            // index of oldest item (when size > 0)
      int hi = 0;            // index of next free slot (when size < max)

      final Channel[] c = {in, request};             // alt channels
      final int IN = 0, REQUEST = 1;                 // alt indices
      final Alternative alt = new Alternative (c);   // alt object
      final boolean[] guard = {true, false};         // alt guards

      while (true) {
        guard[IN] = (size < max);                // only if there's room
        guard[REQUEST] = (size > 0);             // only if not empty
        switch (alt.select (guard)) {
          case IN: {                             // ==> (size < max)
            buffer[hi] = in.read ();             // into first free slot
            hi = (hi + 1) % max;                 // up the pointer
            size++;                              // maintain the size
            break;
          }
          case REQUEST: {                        // ==> (size > 0)
            request.read ();                     // must take the request
            out.write (buffer[lo]);              // send oldest data
            lo = (lo + 1) % max;                 // up the pointer
            size--;                              // maintain the size
            break;
          }
        }
      }
    } catch (InterruptedException e) {}
  }
}
```

This `Buffer` is safe for use by any number of supplier and consumer processes – all are fairly queued (assuming a *queueing* JVM) on their respective access channels. All suppliers are blocked when the buffer is full. All consumers are blocked when the buffer is empty. When the blocks are removed, processes are serviced in the order they were queued – nobody is exposed to the danger of being (forever) overtaken.

We claim that this code is significantly easier to write and understand than the previous monitor version *and* that it is safe (whereas the monitor version is not). The responses of `Buffer` to each signal it receives from the outside world are short and *independent of each other*. The guarding of the alt channels, made possible by the *liveness* of this process, stops any mess happening due to signals being accepted at inappropriate times – since there is no mess to clean up, the code is very simple.

4 Summary

We have outlined the raw mechanisms for threads in Java and for synchronising them via the monitor methods provided. We have shown the difficulties of reasoning with monitors (whose semantics require a consideration of all their methods *at the same time*) – difficulties which are compounded by an incomplete rendering of the ideas presented in Hoare's paper [14]. We have presented a channel model for synchronisation and communication that is directly based on Hoare's (later) algebra of CSP [8, 9] and its practical realisation in occam [13].

We suggest that multi-threaded systems can and should be designed without user involvement in the coding of monitors (i.e. without using the key-word `synchronized` or the `wait`/`notify` methods or, even, the `suspend`/`resume` ones). The occam/CSP approach to Java (which, for the moment, we have christened JavaPP [2]) provides the foundation for the creation of reliable and safe applications. The semantics of the system – whether treated formally or intuitively – are compositional, which means that the meaning of an individual thread fragment does not depend on what other threads may be doing (*What You See Is What You get*). Analysis of systems for deadlock/livelock/starvation properties can exploit the large body of knowledge and tools that have been developed over the last decade for CSP. Even better, higher level CSP design rules (with tool support) may be applied that provide automatic guarantees against the presence of such dengerous attributes.

Of course, without changing the Java language, we cannot match the full security rigour (or even the performance) achieved by a CSP-aware language as occam, but these are significant wins. The JavaPP class libraries are, currently, implemented via the Java monitor methods (and described in this paper). We are looking to see if it is possible to substitute something based on the occam multi-processing kernels (which are specifically geared to support the CSP primitives) for the underlying threads libraries supporting the JVM. These kernels manage process context-switches and process startup/shutdowns in well under a microsecond – around 250 nanoseconds on a fast SPARC Ultra, compared with around 50 microseconds for the equivalent Java on the same processor. The benefit of these very low overheads is that we feel no constraints in designing with lots and lots of threads. The benefit of that is that we get closer to the real-world system we are trying to model and which, ultimately, pays the bills.

References

[1] Peter Welch et al. *Java Threads Workshop – Post Workshop Discussion*. <URL:http://www.hensa.ac.uk/parallel/ groups/ wotug/ java/discussion/ index.html>, February 1997.

[2] JavaPP Team. *JavaPP Home Page*. <URL:http://www.cs.bris.ac.uk/~alan/javapp.html>, February 1997.

[3] J.M.R. Martin and S.A. Jassim. A Tool for Proving Deadlock Freedom. In A. Bakkers, editor, *Parallel Programming and Java, Proceedings of WoTUG 20*, volume 50 of *Concurrent Systems Engineering*, pages 1–16, University of Twente, Netherlands, April 1997. World occam and Transputer User Group (WoTUG), IOS Press, Netherlands.

[4] Gerald Hilderink, Jan Broenink, Wiek Vervoort, and Andre Bakkers. Communicating Java Threads. In A. Bakkers, editor, *Parallel Programming and Java, Proceedings of WoTUG 20*, volume 50 of *Concurrent Systems Engineering*, pages 48–76, University of Twente, Netherlands, April 1997. World occam and Transputer User Group (WoTUG), IOS Press, Netherlands.

[5] G.H. Hilderink. Communicating Java Threads Reference Manual. In A. Bakkers, editor, *Parallel Programming and Java, Proceedings of WoTUG 20*, volume 50 of *Concurrent Systems Engineering*, pages 283–325, University of Twente, Netherlands, April 1997. World occam and Transputer User Group (WoTUG), IOS Press, Netherlands.

[6] How to Design Deadlock-Free Networks Using CSP and Verification Tools A Tutorial Introduction. J.M.R. Martin and S.A. Jassim. In A. Bakkers, editor, *Parallel Programming and Java, Proceedings of WoTUG 20*, volume 50 of *Concurrent Systems Engineering*, pages 326–338, University of Twente, Netherlands, April 1997. World occam and Transputer User Group (WoTUG), IOS Press, Netherlands.

[7] JCSP Team. *JCSP Home Page*. <URL:http://www.cs.ukc.ac.uk/projects/ofa/jcsp>, November 1999.

[8] C.A. Hoare. Communication Sequential Processes. *CACM*, 21(8):666–677, August 1978.

[9] C.A. Hoare. *Communication Sequential Processes*. Prentice Hall, 1985.

[10] Oxford University Computer Laboratory. *The CSP Archive*. <URL: http:// www.comlab.ox.ac.uk/ archive/ csp.html>, 1997.

[11] Ian East. *Parallel Processing with Communication Process Architecture*. UCL press, 1995. ISBN 1-85728-239-6.

[12] John Galletly. *occam 2 – including occam 2.1*. UCL Press, 1996. ISBN 1-85728-362-7.

[13] occam-for-all Team. *occam-for-all Home Page*. <URL:http://www.hensa.ac.uk/parallel/ occam/occam-for-all/index.html>, February 1997.

[14] C.A. Hoare. Monitors: an operating system structuring concept. *CACM*, 17(10):549–557, October 1974.

[15] Ken Arold and James Gosling. *The Java Programming Language*. Addison Wesley Longman, 1996. ISBN 0-201-63455-4.

[16] JavaSoft. *The Java Tutorial: Monitors*. <URL:http://java.sun.com/ docs/ books/ tutorial/ java/ threads/ monitors.html>, 1997.

[17] J.M.R. Martin and P.H. Welch. A Design Strategy for Deadlock-Free Concurrent Systems. *Transputer Communications*, 3(4):215–232, October 1996. ISSN 1070-454X.

[18] J. Martin, I. East, and S. Jassim. Design Rules for Deadlock Freedom. *Transputer Communications*, 2(3):121–133, September 1994. ISSN 1070-454X.

[19] P.H. Welch, G.R.R. Justo, and C. Willcock. High-Level Paradigms for Deadlock-Free High-Performance Systems. In Grebe et al., editors, *Transputer Applications and Systems '93*, pages 981–1004, Amsterdam, 1993. IOS Press. ISBN 90-5199-140-1.

[20] A.W. Roscoe and N. Dathi. The Pursuit of Deadlock Freedom. Technical Report *Technical Monograph PRG-57*, Oxford University Computing Laboratory, 1986.

[21] D.J. Beckett and P.H. Welch. A Strict occam Design Tool. In *Proceedings of UK Parallel '96*, pages 53–69, London, July 1996. BCS PPSIG, Springer-Verlag. ISBN 3-540-76068-7.

[22] A.W. Roscoe. *A Classical Mind - Essays in Honour of C.A.R. Hoare, Model Checking CSP*. Prentice Hall, 1994. ISBN 0-13-294844-3.