# Java Fair Threads

F. Boussinot
EMP-CMA/INRIA - MIMOSA Project
2004 route des Lucioles - BP 93
F-06902 Sophia Antipolis, Cedex
Frederic.Boussinot@sophia.inria.fr

June 10, 2002

### Abstract

Fair threads are cooperative threads run by a fair scheduler which gives them equal access to the processor. Fair threads can communicate using broadcast events, and are fully portable as their semantics does not depends on the executing platform. Fine control over fair threads execution is possible allowing the programming of specific user-defined scheduling strategies. This paper presents fair threads in the context of the Java language, and describes the API to use them. Link with reactive programming, which is at the basis of the fair threads proposal, is also considered.

## 1 Introduction

Contrary to standard sequential programming where the processor executes a single program, in concurrent programming the processor is a shared resource which is dispatched to several programs. The term *concurrent* is appropriate because programs can be seen as concurrently competing to gain access to the processor, in order to execute.
Threads are a basic mean for concurrent programming, and are widely used in operating systems. They are also introduced in the Java language as first-class primitive. At language level, threads offer a way to structure programs by decomposing systems in several concurrent components; in this respect, threads are useful for modularity.
However, threads are generally considered as low-level primitives leading to over-complex programming. Moreover, threads generally have loose semantics, in particular depending on the underlying executing platform; to give them a precise semantics is a difficult task, and this is a clearly identified problem to get portable code.
This paper proposes a new framework with clear and simple semantics, and with an efficient implementation. In it, threads are called *fair*; basically, a fair thread is a cooperative thread executed in a context in which all threads always have equal access to the processor. Fair threads have a deterministic semantics, relying on previous work belonging to the so-called reactive approach.
The structure of the paper is as follows: section 2 introduces threads and scheduling strategies; the **FairThreads** framework is defined in section 3; section 4 contains the Java API for using **FairThreads**; three examples are given in section 5; finally, links with the reactive approach, which is at the basis of the fair thread proposal, are considered in section 6.

## 2 Threads

A thread is basically a program, that is a list of instructions and a program counter which indicates the current instruction to be executed, as shown on Figure Thread 1.

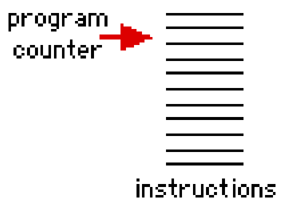


Figure 1: A Thread

Threads are sharing the same address space and are executed in turn, one after the other. The processor is allocated to threads by a scheduler which schedules threads executions. This gives the drawing of Figure Scheduler 2.

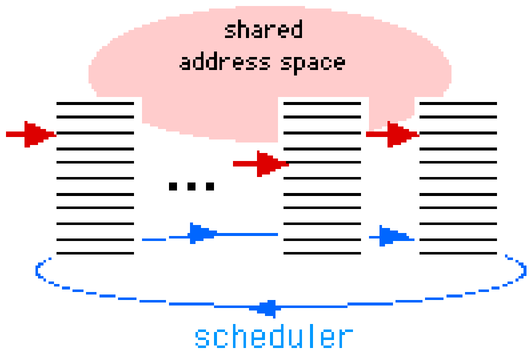


Figure 2: A Scheduler

### Context Switches

When the processor is released by a thread, its execution context has to be saved in order to be able to resume the thread later. This is called a *context switch*. Context switches basically means to save the program counter and the execution stack associated with the executing thread, then to restore the context of a new thread (usually, some others items must also be saved/restored; for simplicity, we do not consider these).

### Threads vs Processes

As opposite to threads, processes have their own private memory space. This implies that communication between processes is more complex than communication between threads, which can directly use shared memory. This also implies that context switches are simpler for threads, since there is no need to save the address space which always remains the same. Because threads context switches need less computing resource than process context switches, threads are sometimes called *lightweight processes*. Note also that less memory swaps are needed using threads than using processes, as threads do not have their own private memory.

### Nondeterminism

The way the scheduler chooses the thread to be executed is usually left unspecified; in this way, implementations are less constrained and, thus, can be more efficient. However, nondeterminism may occurs: with the same input, several distinct outputs are possible, depending on the scheduler choices. A major drawback of nondeterminism is that debugging becomes more complex (for example, faulty executions are more difficult to replay).

## 2.1 Scheduling Strategies

There are two basic scheduling strategies for chosing the next thread to execute, called cooperative and preemptive.

In the cooperative strategy, the scheduler has no way to force a thread to release the processor. Thus, if the executing thread never releases the processor, then other threads never get a chance to execute. This situation can occur when, for example, the executing thread enters in an infinite loop. It is certainly a bug; however, in a cooperative context, this kind of bug is dramatic as it freezes the whole system. In order to avoid such situations, threads must cooperate with other threads and always release the processor after some time.

In a preemptive strategy, the scheduler can force the executing thread to release the processor, when it decides to do so. Scheduler can use several criteria to withdraw the processor from the thread; one criteria is related to execution time: time slices are allocated to threads, and context switch occurs when the time slice given to an executing thread is expired. Other criteria exist, for example criteria related to priorities; for simplicity, one does not consider these.

Pros and cons for cooperative and preemptive strategies are the following:

#### Non-cooperative threads

There is no possibility, in the general case, to decide if a thread is cooperative or not (this is an undecidable problem). Thus, thread cooperativeness cannot be checked automatically and is of the programmer's responsability. This is a real problem for cooperative strategies, as presence of a non-cooperative thread can prevent the global system to work properly. On the contrary, presence of non-cooperative threads is not a problem for preemptive strategies, because they can be forced to release the processor.

#### Reasoning with threads

In a cooperative framework, execution of a thread cannot be interrupted by the scheduler at arbitrary moments. This makes reasonning on programs easier: actually in a cooperative framework, thread execution is atomic as long as the thread does not release the processor. This contrast with preemptive context, where, by default, the scheduler can freely interleave threads instructions.

#### Reusability

In a preemptive context, traditional sequential programs can be embbeded in threads and run without risk to prevent other threads to execute. Thus, preemptive schedulers are a good point for reusability. On the contrary, a "third-party" sequential program cannot be used just as it is in a cooperative framework, as it could prevent the whole system to progress.

#### Nondeterminism

Preemptive schedulers are less deterministic than cooperative ones. Actually, both kinds of schedulers have freedom to chose the next thread to execute; however, a preemptive scheduler has additional freedom to preempt or not the currently executing thread; this additional freedom is a supplementary source of nondeterminism.

#### Data protection

In preemptive frameworks, access to data in the shared address space must be protected because thread switches can occur in the middle of an access. Protection is usually based on use of boolean variables, called locks; a lock is set (taken) by the thread at the beginning of the data use and reset (released) by it at the end; while the lock is taken, other threads willing to access the protected data have to wait for the lock to be released. The test of a lock and its setting must be an atomic action, otherwise the scheduler could withdraw the processor to the thread just in between the test and the setting, leaving the data unprotected. Programmers are thus, in preemptive contexts, highly concerned with data protection. However, deadlocks can occur because of a bad locking strategy (a first thread locks a data needed by a second thread which similarly holds a lock on a data needed by the first). On the contrary, data are not to be protected in a cooperative framework, where threads run atomically. Note that atomicity of test and set actions is automatic; thus, simple boolean variables can be safely used for data protection.

#### Efficiency

Basically, threads unefficiency comes from context switches. In a preemptive strategy context switches are not under control of the programmer, but of the scheduler; in some cases, the scheduler thus introduces context switches which are not necessary but cannot be avoided by programmers. In a cooperative context, it is the programmer's responsability to introduce cooperation actions in order to obtain threads cooperation. One thus generally considers that cooperative strategies can lead to more efficient executions than preemptive ones.

### 2.2 Existing Threading Frameworks

Threading systems can be implemented in several ways, depending on the way kernel resources are used.

#### One-to-one Mapping

On one extreme, there are systems in which user threads are mapped to kernel threads in a one-to-one way. One get the picture of Figure One-one-mapping 3.

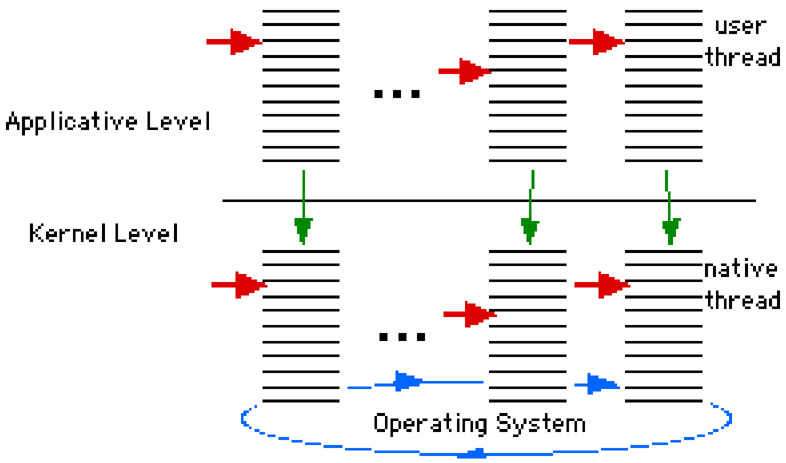


Figure 3: One-to-One Mapping

As modern operating systems are usually preemptive, this approach leads to preemptive threading systems. This is the case of Microsoft NT system. Note that the mapping to kernel threads can be rather inefficient; for example, all thread manipulations in NT systematically introduce an overhead of 600 machine cycles [11].

#### Many-to-one Mapping

On the other extreme, all user threads can be mapped on one unique kernel thread. This many-to-one mapping correspond to Figure Many-one-mapping 4.

In this approach, scheduler and user threads correspond to one process, mapped to one single kernel thread. This approach is more flexible than the previous one, as the choice is left open for the scheduler to implement either a preemptive or a cooperative strategy, independently of the operating system characteristics.

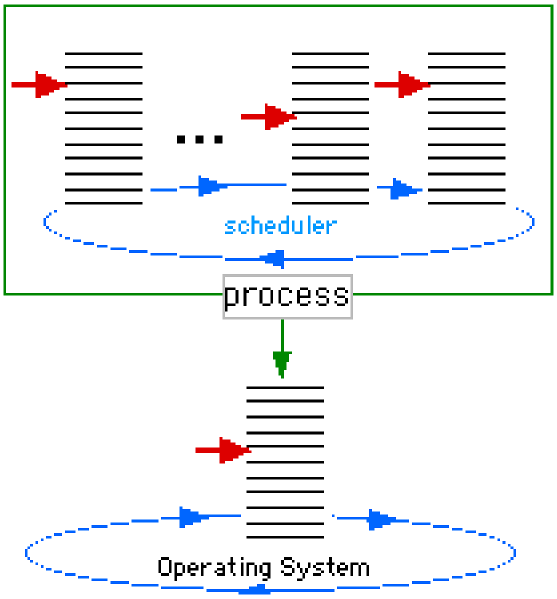


Figure 4: Many-to-One Mapping

### Many-to-many Mapping

There are also intermediate approaches, in which user threads are mapped to several kernel threads. This is the solution proposed by the Solaris system of Sun [4]. In Solaris terminology, user threads are mapped to *green thread* which are grouped in *light weight processes* (LWP). LWPs are the units scheduled by the operating system. This many-to-many approach is represented on Figure Many-many-mapping 5.

Note that threads are executed in a cooperative way inside a LWP, but LWPs are scheduled in a preemptive way by Solaris. Solaris is thus a mix of cooperative and preemptive strategies.

### Thread Libraries and Threads in Languages

A lot of thread libraries exist, for a large number of languages. Among these is the Pthread library [12] which implements in C the POSIX standard for threads.

Linux Threads [2] proposes an implementation of POSIX for Linux. Each Linux thread is a separate Unix process, sharing its address space with the other threads (using the system call `clone()`). Scheduling between threads is handled by the kernel scheduler, just like scheduling between Unix processes.

Recently, Gnu Portable Threads [8] have been proposed with portability as main objective; they are purely cooperative threads.

Very few languages introduce threads as first class primitives. The most well-known is of course Java [6]. Actually, however, Java basically introduces locks (with the `synchronized` keyword), and not directly threads which are available through an API.

The CML language [13] is an other language with threads; it introduces threads in the functional programming framework of ML. CML threads are run by a preemptive scheduler.

### Java Thread

It is very difficult to design threaded systems which run in the same way on both preemptive and cooperative frameworks. Most of the time, threaded systems do not fulfill this requirement, and are thus not portable. In the context of Java, task is even more difficult because no assumption can be made about the JVM scheduling strategy. No assumption can neither be made on the way the JVM is implemented (process, native thread, or something else). In this situation, programmers have to face a situation where:

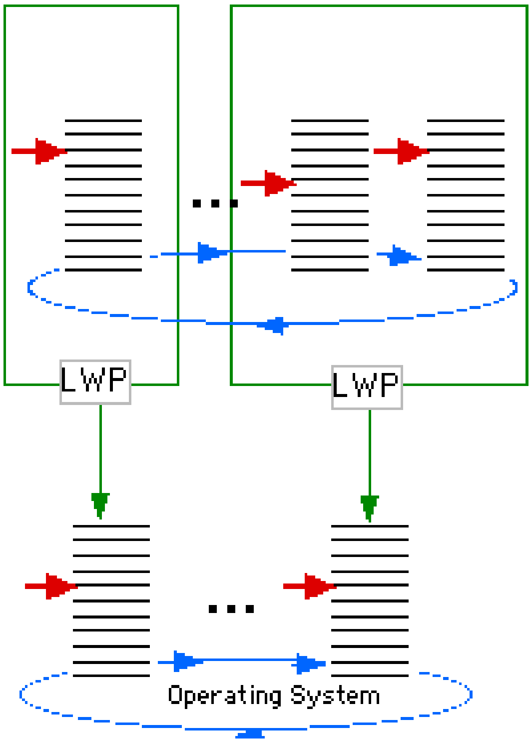


Figure 5: Many-to-Many Mapping

- Each thread can be preempted at any time, because scheduling can be preemptive. This implies that data have to be protected by synchronized code, as soon as they can be accessed by two distinct threads. These protections are unnecessary when a cooperative strategy is used.
- Each thread must periodically give up control in order to cooperate with other threads, because scheduling can be cooperative. Note that the basic `yield()` method for cooperation has a very loose semantics: it does not force a change of processor assignment, but only allows it to take place. Note also that `sleep(long)`, which forces the executing thread to release processor, is not portable as it depends on the machine execution speed.

To sum-up, programmers have to code for the worst case for getting portable concurrent code (one does not consider priorities which do not at all solve the problem; see [11] for a discussion on this point).

Java programmers are faced with a second problem concerning threads: indeed, ways to get fine control over thread, that is ability to stop, suspend, or resume threads, have been suppressed from recent versions of the language [1]. Reason is that using these means is error prone (for example, stopping a thread can damage data that where locked by it). This certainly weaken the language as, without fine control over threads, it becomes very difficult to program user specific scheduling strategies.

## 2.3 Conclusion

We are faced to the (seemingly contradictory) following needs:

- A preemption mechanism is absolutely necessary for code reuse.
- Thread systems should be as deterministic as possible (in particular, for debugging concerns).
- There should be a way to limit to the minimum the use of deadlock prone constructs such as locks.

- One should have possibility to get fine control over threads execution, for being able to program user-defined scheduling strategies.
- Thread context switchings, even if they are cheaper than processes ones, are rather expensive and memory consuming. It should exist a way to limit them to the strict minimum.

The **FairThreads** framework we are introducing now is a proposal solution to fulfill these needs.

# 3 Fair Thread Framework

One considers a new framework made of fair threads executed by fair schedulers. It is presented via a list of questions/answers.

### What are Fair Threads?

A fair thread is basically a cooperative thread which must never forgets to cooperate with other threads, by calling the `cooperate()` method. Fair threads are run by fair schedulers; scheduler fairness is twofold:

- Fair schedulers give each thread an equal possibility to get the processor. Thus, all threads get an equal right to execute. More precisely, fair schedulers define execution phases where all started threads run up to their next cooperation point, as shown on Figure Fair-scheduler 6.

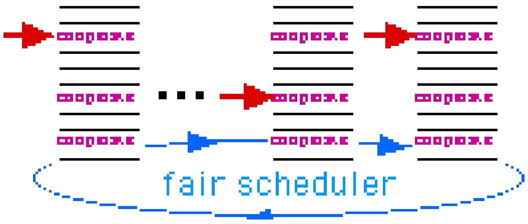


Figure 6: Fair Scheduler

- Fair scheduler always dispatch the same information to all threads. More precisely, a fair scheduler broadcasts events to all threads started in it. Thus, all threads see events in exactly the same way, because they are broadcast. Moreover, events can have values associated to their generations, and these values are also broadcast.

### Why Fair Threads?

The **FairThreads** framework is basically cooperative; it is thus simpler than preemptive ones. Indeed, as preemption cannot occurs in an uncontroled way, cooperative frameworks are less undeterministic. Actually, **FairThreads** puts the situation to an extreme point, as it is fully deterministic; threads are chosen for execution following a strict round-robin algorithm. This can be a great help in programming and debugging. **FairThreads** provides users with a powerful communication mean that is event broadcasting. This simplifies concurrent programming while reducing risks of deadlocks.

### Why Broadcast Events?

Events are used when one wants one or more threads to wait for a condition, without need for them to poll a variable to determine when the condition is fulfilled. Broadcast is a mean to get modularity, as the thread which generates an event has nothing to know about potentially receivers of it. Fairness in event processing means that all threads waiting for an event receive it during the same phase where it is generated; thus, a thread leaving control to cooperate with other threads does not risk to loose an event generated later in the same phase. Note that scheduler phases actually define time scopes of events.

### How is it Implemented?

Fair threads are implemented in the Java programming language and usable through an API. Fair thread implementation is based on standard Java threads, but it is independent of the actual JVM and OS, and is thus fully portable. Fair schedulers are actually at the level of the Java Virtual Machines; one thus have the situation shown on Figure Implementation 7.

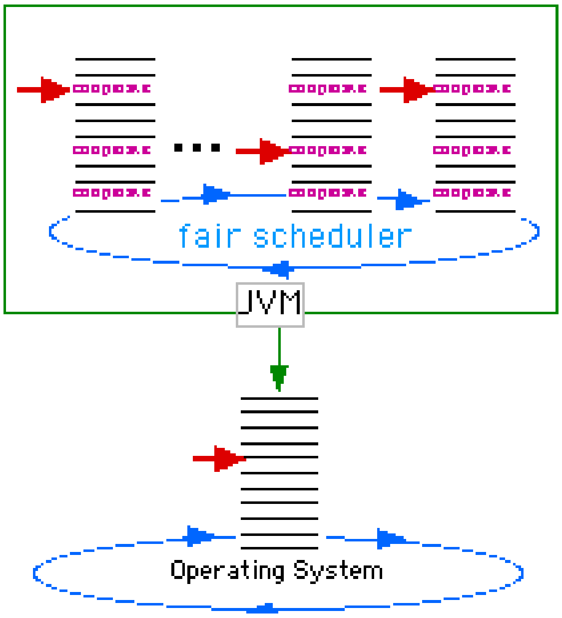


Figure 7: Implementation

### What about Locks?

As fair threads are basically cooperative, no lock is needed when accessing a shared object. While executing, a fair thread cannot be interrupted by another fair thread; thus, execution is atomic and there is no need of synchronized code. This contributes to minimize deadlock situations which are the plague of concurrent programming.

### What about Priorities?

Priorities are meaningless in a fair context, where threads always have equal rights to execute. Absence of priorities also contributes to simplify programming. Note that the effect of priorities in Java is rather unclear (see [11] for a discussion on this matter).

### What about Preemption?

A preemptive strategy is sometimes needed, for example to reuse a piece of code which was not designed to be run concurrently. In the context of fair threads, preemption is possible through the notion of a fair process, assuming that the operating system is preemptive. A fair process gives life to a standard process which is executed by the operating system concurently with the JVM running the fair scheduler. This gives the drawing of Figure Fair-process 8, where the fair process is represented as a black box.

Note that one gets an instance of the many-to-many approach presented in section 2.2.

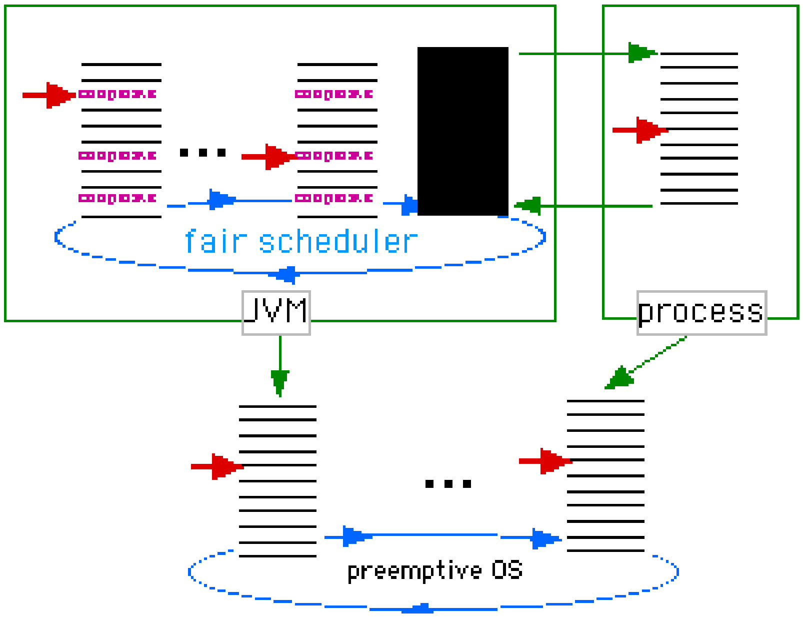


Figure 8: Fair Process

**What about Parallelism?**

Up to now, one has only considered uniprocessor machines. When several processors are available, several threads can be simultaneously executed; this situation is often called parallelism. As threads are sharing the same address space, data protection becomes mandatory. Actually, this is very similar to preemptive scheduling; in both cases, shared data have to be protected against concurrent accesses, and it is the programmer's responsability to avoid deadlocks.

Fair threads are designed for uniprocessor machines, and it is left for future work to adapt them to multiprocessor ones.

**What about Signals and Interrupts?**

In operating systems, signals can occur at any moment during execution, and are to be processed without delay. Signals are useful to implement interrupts, for example asynchronous IO interrupts. In the context of fair threads, signals are quite naturally represented by events, which can be generated at any moment. Fair threads also offer the possibility for generated values to be immediately processed: generated values associated to events are broadcast to all components, and are received by them without delay, that is during their generation phase.

# 4 Java FairThreads API

The FairThreads framework is contained in a Java package named `fairthread`. Three main interfaces are defined in it: `Event` for events, `FairThreadInterface` for fair threads, and `FairSchedulerInterface` for fair schedulers.

## 4.1 Event Interface

The interface `Event` basically defines two methods: one for hashcoding events, and one for comparing them; these two methods are the minimum needed for implementing events.

```
public interface Event
{
```

```
    boolean equals(Object object);
    public int hashCode();
}
```

A possible implementation of events is the class `StringEvent` in which they are considered as strings:

```
public class StringEvent implements Event
{
    public String identity;
    public StringEvent(String s){ identity = s; }
    public int hashCode(){ return identity.hashCode(); }
    public boolean equals(Object object){
       if (object instanceof StringEvent)
           return identity.equals(((StringEvent)object).identity);
       return false;
    }
}
```

## 4.2 Fair Thread Interface

The interface of fair threads is the following:

```
public interface FairThreadInterface
{
   void run(FairScheduler scheduler);
   void start(FairScheduler scheduler);
   void stop(FairScheduler scheduler);
   void suspend(FairScheduler scheduler);
   void resume(FairScheduler scheduler);

   void cooperate();
   void cooperate(long delay);

   void join(FairThread thread);
   void join(FairThread thread,long delay);

   void generate(Event event);
   void generate(String event);
   void generate(Event event,Object val);
   void generate(String event,Object val);

   void await(Event event);
   void await(String event);
   void await(Event event,long delay);
   void await(String event,long delay);

   final static public Object NULL = new Object();
   Object nextValue(Event event);
   Object nextValue(String event);
}
```

### Control over Threads

- Method `run(FairScheduler s)` is the basic method executed by the fair thread; `s` is the fair scheduler that runs the thread. By default, the method does nothing. This method corresponds to the `run()` method of standard Java threads.
- Method `start(FairScheduler s)` starts execution of the fair thread by `s`. It corresponds to the `start()` method of standard threads.

- Method `stop(FairScheduler s)` stops the fair thread run by `s`. It corresponds to the `stop()` method of Java version 1, which is now deprecated.
- Method `suspend(FairScheduler s)` is used to suspend the fair thread run by `s`; execution can be resumed by calling the `resume(FairScheduler)` method. It corresponds to the `suspend()` method of Java version 1 which is now deprecated.
- Method `resume(FairScheduler s)` resumes the thread, previously suspended by a call to `suspend(FairScheduler)`. It corresponds to the `suspend()` method of Java version 1 which is now deprecated.

### Cooperation

- Method `cooperate()` is the method to be called by the fair thread to cooperate with other threads; it is the basic mechanism of **FairThreads**. It corresponds to `yield()` of standard threads.
- By calling method `cooperate(long n)` the fair thread falls asleep during n phases. The method corresponds to the `sleep(long)` method of standard threads. Actually, `cooperate(n)` is a loop that runs `cooperate()` `n` times.

### Joining Threads

- Method `join(FairThread t)` waits for the termination of the fair thread `t`. It corresponds to method `join()` of standard threads.
- Method `join(FairThread t,long n)` waits for termination of the fair thread `t` during at most `n` phases. It corresponds to the method `join(long)` of standard threads.

### Generation of Events

- Method `generate(Event e)` generates the event `e`. All fair threads waiting for `e` automatically receive it, because it is broadcast by the fair scheduler running the thread. Actually, events correspond more or less to condition variables of Pthreads [12], and `generate(Event)` is the counterpart of `pthread_cond_broadcast`. Method `generate(String s)` is similar, `s` being converted to an event.
- Method `generate(Event e,Object o)` generates `e` with `o` as generated value. Method `generate(String s,Object o)` is similar, `s` being converted to an event.

### Awaiting Events

- Method `await(Event e)` waits for `e`. It corresponds to the function called `pthread_cond_wait` of Pthreads. Method `await(String s)` is similar, `s` being converted to an event.
- Method `await(Event e,long n)` waits for event `e` during at most `n` phases. It corresponds to the function called `pthread_cond_timedwait` of Pthreads. Method `await(String s,long n)` is similar, `s` being converted to an event.

### Generated Values

- Method `nextValue(Event e)` returns the next value of `e` generated during the current phase. Value `NULL` is returned if the value generated was `null`. Value `null` is returned if no new generated value is available. In this last case, `nextValue` remains blocked up to the end of the current phase, and returns during the next phase (indeed, absence of a value is impossible to decide before the end of the current phase, as otherwise it could be produced by a thread scheduled later in the same phase). Method `nextValue(String s)` is similar, `s` being converted to an event.

## 4.3 Fair Scheduler Interface

Interface of fair schedulers is the following:

```
public interface FairSchedulerInterface
{
   void broadcast(Event event);
   void broadcast(String event);
   void broadcast(Event event,Object val);
   void broadcast(String event,Object val);

   void start(FairThread thread);
   void stop(FairThread thread);
   void suspend(FairThread thread);
   void resume(FairThread thread);
}
```

- Method `broadcast(Event e)` broadcasts `e` to all executing fair threads. Method `broadcast(String s)` is similar, `s` being converted to an event. Method `broadcast(Event e, Object o)` broadcasts `e`, with `o` as generated value. Method `broadcast(String s,Object o)` is similar, `s` being converted to an event.
- Method `start(FairThread t)` starts `t`. The call `scheduler.start(thread)` is equivalent to `thread.start(scheduler`
- Method `stop(FairThread t)` stops `t`. The call `scheduler.stop(thread)` is equivalent to `thread.stop(scheduler)`.
- Method `suspend(FairThread t)` suspends `t`. The call `scheduler.suspend (thread)` is equivalent to the call `thread.suspend(scheduler)`.
- Method `resume(FairThread t)` resumes thread `t`. The call `scheduler.resume (thread)` is equivalent to `thread.resume(scheduler)`.

## 4.4 Fair Thread Class

Fair threads are instance of the `FairThread` class which extends the standard class `Thread`. Class `FairThread` implements interface `FairThreadInterface` and has two constructors:

```
public FairThread()
public FairThread(Fair fair)
```

- The first constructor is used to create a fair thread with the default empty `run (FairScheduler)` method. It corresponds to the constructor without parameter of the class `Thread`.
- The second constructor creates a fair thread from a fair object (see section "ref–Fair Interface") given as argument. It corresponds to the constructor with a `Runnable` parameter of the class `Thread`.

## 4.5 Fair Scheduler Class

Fair schedulers are instances of the class `FairScheduler` which has one single constructor:

```
public FairScheduler()
```

Usually, a fair scheduler is created at starting of the main application method, and is used later to start fair threads created during execution.

### 4.6 Fair Interface

Interface `Fair` is to be implemented by objects given to the fair thread constructor. It corresponds to the `Runnable` interface of standard Java threads.

```
public interface Fair
{
   void run(FairScheduler scheduler,FairThread thread);
}
```

- The `run(FairScheduler s,FairThread t)` method is the basic method executed by fair thread `t`, run by the fair scheduler `s`.

### 4.7 Fair Process

A fair process corresponds to a standard process created by the call `exec(String)` of the class `Runtime`, with the difference that it can be safely immerged in the **FairThreads** framework. Fair processes are instances of the class `FairProcess` which extends `FairThread`; it has one single constructor:

```
FairProcess(String command)
```

The string in argument is interpreted as a command and it is parsed exactly in the same way the method `Runtime.exec` does.

## 5 Examples

Three examples are described. The first one is a very simple but complete example in which two fair threads are run. In the second example, one defines a preemption mechanism to stop a fair thread when an event becomes present. The third example considers generated values and defines a way to process them.

### 5.1 Hello World Example

Here is a small example of a complete program that runs two fair threads. The first one cyclically prints `hello, `, and the second one prints `world`. The main method of the program first creates a fair scheduler, and then starts the two threads in it.

```
import fairthread.*;

class Say extends FairThread
{
   String msg;
   public Say(String msg){ this.msg = msg; }

   public void run(FairScheduler scheduler){
      while(true){
         System.out.print(msg);
         cooperate();
      }
   }
}

public class HelloWorld
{
    public static void main(String[] args){
       FairScheduler scheduler = new FairScheduler();
       scheduler.start(new Say("hello, "));
       scheduler.start(new Say("world!\n"));
   }
}
```

Several points are important:

- Output is an infinite list of `hello, world` messages. This is because the first thread, after printing the first part of the message, always leaves the processor and thus lets the second thread print the last part of it.
- Output is always the same; the program is totally deterministic.
- Without the `cooperate()` call in `Say`, the program would enter in a loop, printing `hello` forever; `cooperate()` is mandatory because **FairThreads** is basically cooperative.

Using the `Fair` interface, the previous program could be equivalently written as:

```
import fairthread.*;

class Say implements Fair
{
   String msg;
   public Say(String msg){ this.msg = msg; }

   public void run(FairScheduler scheduler,FairThread thread){
      while(true){
         System.out.print(msg);
         thread.cooperate();
      }
   }
}

public class HelloWorld
{
    public static void main(String[] args){
       FairScheduler scheduler = new FairScheduler();
       scheduler.start(new FairThread(new Say("hello, ")));
       scheduler.start(new FairThread(new Say("world!\n")));
   }
}
```

The main point is that, in this way, class `Say` can extend an other class; this is not possible with the first program, as multiple inheritance of classes is forbidden in Java.

## 5.2 Stopping Threads with Events

One defines a class `Until` which extends `FairThread`, with an associated class `Controller`. `Until` runs a method, called `controlled`, and starts an instance of `Controller` which stops the `Until` thread when an event becomes present. Here is definition of `Until`:

```
import fairthread.*;

public class Until extends FairThread
{
   String event;

   public Until(String event){ this.event = event; }

   public void controlled(FairScheduler scheduler){}

   public final void run(FairScheduler scheduler){
       FairThread controller = new Controller(event,this);
       scheduler.start(controller);
```

```
        controlled(scheduler);
        scheduler.stop(controller);
   }
}

class Controller extends FairThread
{
   FairThread controlled;
   String event;

   public Controller(String event,FairThread controlled){
      this.event = event; this.controlled = controlled;
   }

   public void run(FairScheduler scheduler){
      await(event);
      scheduler.stop(controlled);
   }
}
```

This example gives a hint of how one can program sophisticated user-defined execution strategies for fair threads, using broadcast events.

### 5.3 Generated Values

A *scanner* is a thread which is associated to one event and which runs a special callback method each time a value is generated for the event. The notion of a scanner comes from the new version of the SugarCubes [5] framework. Scanners extends the following `Scanner` class, redefining the `callback` method:

```
public class Scanner extends FairThread
{
   String event;
   public Scanner(String event){ this.event = event; }

   void callback(Object obj){}

   public void run(FairScheduler scheduler){
     while(true){
          await(event);
          while(true){
             Object obj = nextValue(event);
             if (obj == null) break;
             callback(obj);
          }
      }
   }
}
```

At each phase, all generated values are processed during it; moreover, after the processing of the last value, the call of `nextValue` blocks, waiting for the next phase to return the `null` value.

## 6 Links with Reactive Programming

When considering the `Until` class of section 5.2, one can see that there is no real need for using two distinct threads. Indeed, it would be possible to merge the instructions of the controler with the ones of the controlled method; in this way, the `Until` thread would also test for the event presence, at fixed moments, say, just before execution of the `cooperate` method. Thus, unnecessary context switchings would be saved, which would leads to a more efficient execution. Such a solution is of course more difficult to program and less modular as the code of the controlled method must be transformed.

### Reactive Programming

The fair framework offers a way to conciliate efficiency and programming ease through the use of reactive programs. Reactive programs are basic elements of the Junior framework [10] which is a set of Java classes for reactive programming. Junior programs are usable through an API called Jr [9]. Comparison of threads and of SugarCubes, a framework closely related to Junior, can be found in [7]. The Web site [3] contains references to reactive programming.

Reactive programming is based on the notion of an instant which is shared by all concurrent components. Reactive programs are made of basic instructions, with semantics defined in term of instants. For example, the instruction `Stop()` of Junior stops execution for the current instant, and execution at the next instant restarts in sequence from it. Event based programming is possible in Junior; events are instantaneously broadcast: an event generated during one instant is received by all components waiting for it during the same instant.
Correspondance between fair threads and Junior is straigtforward: a Junior instant corresponds to a fair scheduler phase, where all threads are executed once; events have exactly the same semantics in the two contexts; the `cooperate()` method of fair threads corresponds to the `Stop()` instruction of Junior.
Junior programs can be used in the context of fair threads, by the intermediate of class `JrProgWrapper` which extends `FairThread`. For example, consider the class:

```
public class Kill extends Until
{
   String msg;
   public Kill(String s){ super("E"); msg = s; }
   public void controled(FairScheduler scheduler){
      for(int i = 0; i < 100; i++){
         System.out.print(msg+"init ");
         cooperate(5);
         System.out.print(msg+"end ");
      }
   }
}
```

The equivalent class, made of a reactive program of type `Program`, is the following:

```
public class Kill extends JrProgWrapper
{
    Program prog =
      Jr.Until("E",
        Jr.Repeat(100,
          Jr.Seq(Jr.Atom(new Print(msg+"init ")),
          Jr.Seq(Jr.Repeat(5,Jr.Stop()),
                 Jr.Atom(new Print(msg+"end "))))));

    public Kill(){ super(); setProgram(prog); }
}
```

### Efficiency

Actually, instructions of concurrent programs in Junior are interleaved (by the `Par` operator) during each instant, in a way that allows broadcast of events. This approach is different from the one of threads, as there is no need of context switching in Junior; actually, context switches are replaced in reactive programming by interleavings, as shown on Figure Merge 9.


Figure 9: Merge of Reactive Instructions

The absence of context switching is a good point for efficiency. Let us return to the `Kill` example of previous section. We consider N threads and measure time T (in milliseconds) taken to run them (on a MacIntosh G3, without printing). For N=100, one has T=6533ms. Now, running the equivalent `JrProgWrapper` gives T= 1577ms. For N=500, T=31189ms with `FairThread`, and T=3685ms with `JrProgWrapper`.

Note that reactive programs consume less memory than threads; last experiment, with N=500, needs 8.7MB with `JrProgWrapper`, and 79.3MB using threads. The overhead of memory used by threads is a well-known problem; for example, with N=1000, it has not been possible to run the previous example using threads, while no problem arise using a reactive program.

# 7 Conclusion

One has defined a new framework for concurrent programming in Java, based on the notion of a fair thread. Fair threads are run by fair schedulers which give threads equal rights to get the processor and equal rights to receive broadcast events.
The **FairThreads** framework makes a clear separation between the cooperative world and the preemptive one. Fair threads are basically cooperative, and it is the programmer's responsability to program cooperation with other threads. However, there exists a way to embed non-cooperative code in **FairThreads**, through the notion of a fair process.
**FairThreads** has a clear and simple semantics, relying on the reactive approach. As a consequence, **FairThreads** programs are basically deterministic and fully portable.
Fair threads are of valuable help with uniprocessor machines to get portable, simple, and efficient code.
The API of the **FairThreads** framework and its implementation are available on the Web [3].

### Acknowledgments

Thanks to Julien Demaria, Loic Henry-Greard, Xavier Leroy, Fabrice Peix, and Jean-Ferdy Susini for their valuable comments and suggestions on a previous version of the paper.

# References

[1] **Java Web Site** – `http://java.sun.com`.

[2] **LinuxThreads Web Site** – `http://pauillac.inria.fr/~xleroy/linuxthreads/`.

[3] **Reactive Programming Web Site** – `http://www.inria.fr/mimosa/rp`.

[4] **Solaris Web Site** – `http://www.sun.com/solaris`.

[5] **SugarCubes Web Site** – `http://www.inria.fr/mimosa/rp/SugarCubes`.

[6] Arnold, Ken and Goslin, James – **The Java Programming Language** – *Addison-Wesley*, 1996.

[7] Boussinot, F. and Susini, J-F. – **Java threads and SugarCubes** – *Software Practice and Experience*, 30(5), 2000, pp. 545-566.

[8] Engelschall, Ralf S. – **Portable Multithreading** – *Proc. USENIX Annual Technical Conference*, San Diego, California, 2000.

[9] Hazard, L. and Susini, J-F. and Boussinot, F. – **Programming with Junior** – *Inria Research Report*, (4027), 2000.

[10] Hazard, L. and Susini, J-F. and Boussinot, F. – **The Junior reactive kernel** – *Inria Research Report*, (3732), 1999.

[11] Hollub, A. – **Taming Java Threads** – *Apress*, 2000.

[12] Nichols, B. and Buttlar, D. and Proulx Farrell J. – **Pthreads Programming** – *O'Reilly*, 1996.

[13] Reppy, John H. – **Concurrent Programming in ML** – *Cambridge University Press*, 1999.