CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. 2006; **18**:445–469
Published online 13 September 2005 in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/cpe.919

FairThreads: mixing cooperative and preemptive threads in C

Frédéric Boussinot*,†

EMP-CMA/INRIA—MIMOSA Project, 2004 route des Lucioles—BP 93, F-06902 Sophia Antipolis, Cedex, France

SUMMARY

FairThreads introduces fair threads which are executed in a cooperative way when linked to a scheduler, and in a preemptive way otherwise. Constructs exist for programming the dynamic linking/unlinking of threads during execution. Users can profit from the cooperative scheduling when threads are linked. For example, data only accessed by the threads linked to the same scheduler does not need to be protected by locks. Users can also profit from the preemptive scheduling provided by the operating system (OS) when threads are unlinked, for example to deal with blocking I/Os. In the cooperative context, for the threads linked to the same scheduler, FairThreads make it possible to use broadcast events. Broadcasting is a powerful, abstract, and modular means of communication. Basically, event broadcasting is made possible by the specific way threads are scheduled by the scheduler to which they are linked (the 'fair' strategy). FairThreads give a way to deal with some limitations of the OS. Automata are special threads, coded as state machines, which do not need the allocation of a native thread and which have efficient execution. Automata also give a means to deal with the limited number of native threads available when large numbers of concurrent tasks are needed, for example in simulations. Copyright © 2005 John Wiley & Sons, Ltd.

KEY WORDS: concurrency; parallelism; thread; C language; multiprocessor; reactive programming

1. INTRODUCTION

Threads give users access to concurrency, a technique which is widely recognized as central in programming. They are generally considered as having two major advantages: first, multi-threaded programs can benefit from multiprocessor machines, in particular those based on *symmetric multiprocessing* (SMP) architectures, which are now widely available. Secondly, blocking I/Os do not need special attention because, as the scheduler is preemptive, there is no risk that a thread blocked on an I/O operation also blocks the rest of the system.

*Correspondence to: Frédéric Boussinot, EMP-CMA/INRIA—MIMOSA Project, 2004 route des Lucioles—BP 93, F-06902 Sophia Antipolis, Cedex, France.
†E-mail: frederic.boussinot@sophia.inria.fr

Received 9 January 2004
Revised 29 April 2004
Accepted 14 July 2004

The benefit of threads is, however, not so clear for systems made of tasks needing frequent synchronization or communication actions. Indeed, in a preemptive context, to communicate or to synchronize generally implies the need to protect some of the data involved in the communication or in the synchronization. Locks are basically used for this purpose, but they have a cost and are error-prone (introducing the possibility of deadlocks).

Pure cooperative threads are actually more adapted for highly communicating tasks. Indeed, data protection is no longer needed and one can avoid the use of locks. Moreover, cooperative threads have clear and simple semantics, and are thus easier to program and to port. However, while cooperative threads can be efficiently implemented at user level, they cannot benefit from multiprocessor machines and they need special means to deal with blocking I/Os.

Actually, programming with threads is difficult because threads generally have very 'loose' semantics. This is particularly true with preemptive threads because their semantics strongly relies on the scheduling policy. The semantics of threads also depends on other aspects, for example the way threads' priorities are mapped at the kernel level. Moreover, threads raise efficiency problems. For example, threads take time to create, and need a rather large amount of memory to execute. Another issue is related to the limitation of the number of native threads than can be created at system level. Several techniques exist to bypass these problems, especially when large numbers of short-lived components are needed. Among these techniques is thread-pooling, to limit the number of created threads, and the use of small code fragments, sometimes called *chores* or *chunks*.

1.1. The FairThreads proposal

FairThreads proposes to overcome the difficulties of threads by giving users the opportunity to choose the context, cooperative or preemptive, in which threads are executed.

More precisely, FairThreads defines *schedulers* which are cooperative contexts to which threads can dynamically link or unlink. A thread can be linked to at most one scheduler at a time. All threads linked to the same scheduler are executed in a cooperative way, and at the same pace. Threads which are not linked to any scheduler are executed by the OS in a preemptive way, at their own pace. An important point is that FairThreads offers programming constructs to dynamically link and unlink threads.

FairThreads has the following main characteristics.

- Programs can take advantage of multiprocessor machines. Indeed, schedulers and unlinked threads can be run in real parallelism, on distinct processors.
- It allows users to stay in a purely cooperative context by linking all the threads to the same scheduler. In this case, systems are completely deterministic and have a simple and clear semantics.
- Blocking I/Os can be implemented in a very simple way, using unlinked threads.
- It defines *instants* shared by all the threads which are linked to the same scheduler. Thus, all threads linked to the same scheduler execute at the same pace, and there is an automatic synchronization at the end of each instant.
- It introduces *events* which are instantaneously broadcast to all the threads linked to a scheduler; events are a modular and powerful mean for threads to synchronize and communicate.
- It defines *automata* to deal with small, short-lived tasks, which do not need the full power of native threads. Automata have lightweight implementation and are not subject to some of the limitations of native threads.

This paper describes FairThreads in the context of C, implemented on top of the Pthreads library [1]. The structure is as follows. Section 2 presents the rationale for the design of FairThreads. An overview of the application programmer interface (API) of FairThreads is given in Section 3. Several examples showing various aspects of FairThreads are described in Section 4. Related work is considered in Section 5. Finally, Section 6 concludes the paper.

2. RATIONALE

In FairThreads, schedulers can be seen as *synchronization servers*, in which linked threads automatically synchronize at the end of each instant. However, in order to synchronize, linked threads must behave fairly[‡] and cooperate with the other threads by returning the control to the scheduler. Thus, linked threads are basically *cooperative* threads. Schedulers can also be seen as *event servers* as they are in charge of broadcasting generated events to all the linked threads. In this way, a scheduler defines a kind of *synchronized area* made of cooperative threads running at the same pace and communicating through broadcast events.

2.1. Synchronized areas

A synchronized area can, quite naturally, be defined to manage some shared data that has to be accessed by several threads. In order to get access to the data, a thread first has to link to the area, and then it becomes scheduled by the area and can thus get safe access to the data[§]. Indeed, as the scheduling is cooperative, there is no risk to the thread of being preempted during an access to the data. The use of a synchronized area is, in this case, an alternative to the use of locks. A synchronized area can also play the role of a location that threads can join when some kind of communication or synchronization is needed.

FairThreads allows programmers to decompose complex systems into several threads and areas to which threads can link dynamically, following their needs. Moreover, a thread can be unlinked, that is it can be totally free from any synchronization provided by any schedulers. Of course, unlinked threads cannot benefit from broadcast events. Unlinked threads are run in the preemptive context of the operating system (OS), and are thus just standard preemptive threads. Data shared by unlinked threads have to be protected by locks, in the standard way.

The computing model of FairThreads leads to systems made of synchronous areas, as shown in Figure 1. In Figure 1, threads are represented as vertical stacks of small segments (the instructions) with associated arrows (the program counters). The shaded areas are the synchronized areas defined by the schedulers. Threads linked to them are executed in a cooperative way.

2.2. Linked threads

Basically, a linked fair thread is a cooperative thread which can synchronize with other fair threads using events, and which can communicate with them through values associated to these events.

[‡]Hence the name FairThreads; note, however, that this differs from the standard meaning of *fair* in the context of concurrency.

[§]In this respect, schedulers are quite close to standard monitors (see the classification of Buhr *et al.* [2]), except that they need a dedicated thread of control.

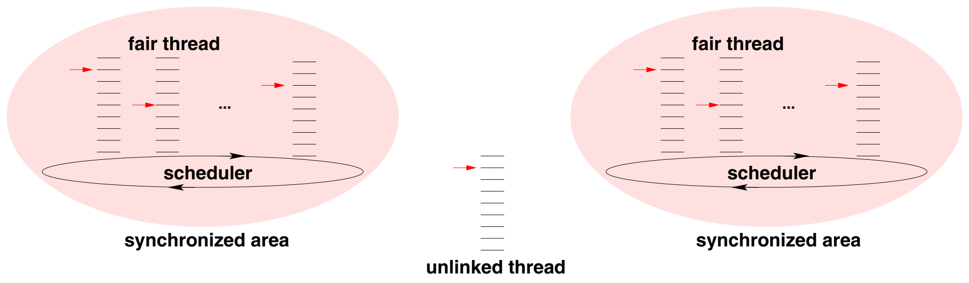

Figure 1. Two synchronized areas and one unlinked thread.

Intuitively, all threads linked to the scheduler get an equal right to execute. More precisely, a scheduler defines *instants* during which all threads linked to it run up to their next cooperation point. There are two kinds of cooperation points.

- Explicit ones, which are calls to the `cooperate` function, used when the thread has finished its execution for the current instant. In this case, the thread will only regain the control at the next instant (except of course if it is suspended or stopped). The `cooperate` function can thus be seen as a kind of 'yield', the primitive that is central in co-routine based formalisms.
- Implicit points, where threads are waiting for events. Note that in this case, the execution can return to the thread during the same instant, if the awaited event is generated later, by any other thread linked to the scheduler.

A fair scheduler actually *broadcasts* generated events to all the fair threads linked to it. Thus, all the threads linked to the same scheduler 'see' the presence and the absence of events in exactly the same way. Moreover, values associated to events are also broadcast. Actually, events are local to the scheduler in which they are created, and are non-persistent data which are reset at the beginning of each new instant.

Fair scheduling

To show how the fair scheduling works, consider three fair threads informally represented by:

Thread A	**Thread B**	**Thread C**
1: await evt1	1: generate evt1	1: await evt1
2: await evt2	2: cooperate	2: generate evt2
3: cooperate	3: generate evt3	3: await evt3
4: await evt1		

Let us detail the scheduling of these threads and describe, for each instant, exactly *how* and *when* each thread executes.

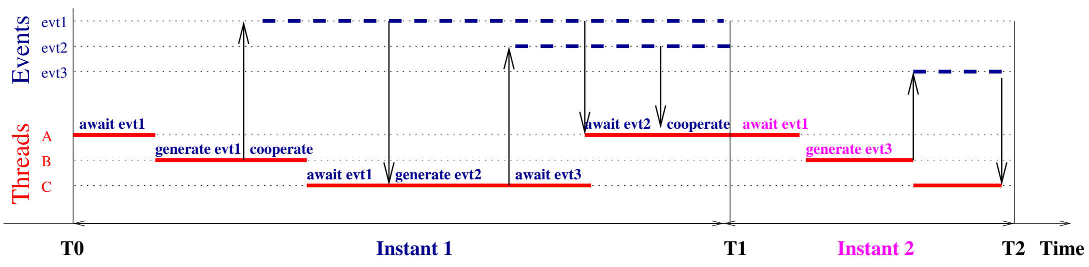

Figure 2. Executing fair threads A, B, and C.

- Instant 1: Thread *A* gets blocked (line 1) because it is waiting for event `evt1` which is not already generated during the instant.
  Thread *B* generates `evt1` (line 1), then it cooperates (line 2), which means that it has completed its execution for the instant. It is explicitly blocked.
  Thread *C* does not block on event `evt1` as it has been generated (by *B*) during the instant. It generates `evt2` and it blocks, waiting for event `evt3`.
  At this point all threads have executed but there exists a thread (*A*) that is blocked on an event which is present. Thus the scheduler re-elects *A* in the instant. This thread does not block on event `evt2` because it is present (generated by *C*). Then it explicitly cooperates (line 3). At this point all threads are blocked and no new event has been generated. This marks the end of instant 1. Events are reset before proceeding to the next instant.
- Instant 2: Thread *A* gets blocked instantly waiting for `evt1` (line 4), which has not been generated during the instant. (Remember that events are reset at the beginning of each instant.)
  Thread *B* generates `evt3` and then it terminates its execution.
  Thread *C* awakes for `evt3` (line 3) and it terminates. At this point all threads are either blocked or terminated. Instant 2 ends.
- Instant 3: Thread *A* blocks for the instant and for the next instants until `evt1` is generated.

The execution of the three threads is summarized in Figure 2.

Modularity

Events are a powerful synchronization and communication mechanism which simplifies concurrent programming while reducing the risk of deadlocks. Events are used when one wants one or more threads to wait for a condition, without polling a variable to determine when the condition is fulfilled (from this point of view, events correspond to the condition variables of Pthreads). Broadcasting is a way to get modularity, as the thread which generates an event requires no knowledge about potential receivers. Fairness in event processing means that all threads waiting for an event always receive it the same instant it is generated; thus a thread which waits for an event and returns the control to the scheduler does not risk losing the event if it is generated later in the same instant. Indeed, in this case, the scheduler will necessarily resume the thread during the instant.

Determinism

Cooperative frameworks are more deterministic than preemptive ones, as in cooperative frameworks preemption cannot occur in an uncontrolled way. Actually, FairThreads takes this to an extreme point when considering linked threads: at each instant, the order in which a scheduler starts to run threads is always the same, and the scheduler iterates until every thread gets blocked using this very same order. More precisely, the order is actually the one in which threads have been linked to the scheduler. This strategy leads to completely deterministic systems which can be a great help in programming and debugging.

Note that, using events, one can get behaviors that are actually independent of the order in which threads are executed. For example, returning to the example described in Figure 2, the reader can verify that executing the three threads in a different order would produce the same results (the same events would be generated at the same instants). Of course, the presence of side effects, as print instructions, makes things more complicated, and, in the general case, the result depends on the order in which threads are chosen for execution.

Absence of priorities

Priorities are meaningless for linked threads which always have an equal right to execute. Absence of priorities also contributes to simpler programming.

2.3. Automata

Automata are special fair threads which are coded as state machines and are always linked to a scheduler. An automaton can dynamically change its linking to a scheduler using an atomic operation (this is different with standard fair threads, which must first unlink and then re-link). As with standard fair threads, events can be awaited and generated in automata with the same semantics. The point is that an automaton does not need the full power of a native thread to execute. An automaton is actually run by the thread of the scheduler to which it is linked. This is possible because an automaton is basically a state-machine which does not need a dedicated stack to store its execution state or its local variables, and because it is never unlinked. As a consequence, an automaton can be implemented more efficiently than a standard fair thread but its expressive power is more limited (for example, recursive functions that are not tail-recursive cannot be coded with automata).

Automata are specially useful in two cases. The first one is for coding auxiliary or short-lived small tasks (for example, waiting for an event to stop a thread) which do not need the allocation of a native thread. The second case is when large numbers of tasks are needed. Indeed, in this case, the use of a standard fair thread, mapped to a native thread, would possibly exceed the number of native threads allowed by the system.

Basically, automata are lists of *states* which are elementary pieces of sequential code. The current state is stored by the automaton and execution starts from it at the beginning of the instant. Specific states are defined for dealing with events; for example, there exists a wait state in which the automaton stays until an event is present. Explicit jump operations are provided to leave states. When a state terminates without any explicit jump, execution automatically proceeds to the next state. Execution of the automaton terminates when the last state is exited. Thus, the fine-grain sequentiality of execution inside states is not memorized by automata, which only capture the coarse-grain sequentiality of states.

2.4. Mapping to native threads

In FairThreads, all fair threads, except automata, are mapped onto native threads, which in the implementation are standard pthreads of the Pthreads library; in the rest of the paper, native threads and pthreads will be considered as synonymous. Fair threads which are linked to a scheduler are under the scheduler's control, while unlinked threads behave as standard native preemptive threads, under the control of the OS. Actually, unlinked threads are introduced in FairThreads for two main reasons. First, using unlinked threads, users can program non-blocking I/Os in a very simple way. Without this kind of I/O, programming would become problematic. Secondly, unlinked threads can be run by distinct processors. The use of unlinked threads is a plus in multiprocessor contexts.

Schedulers are basically mapped to native threads and run autonomously, defining distinct synchronized areas, with their own instants. However, FairThreads also gives a way to control the execution of schedulers, instant by instant, which allows users to program complex scheduling strategies, involving several schedulers run in a synchronized way.

2.5. Use for simulations

Simulation of physical entities is used in many distinct areas, ranging from surgery training to games. The standard approach consists in discretization of time, and then integration using some stepwise method.

The use of threads to simulate separate and independent objects of the real world appears quite natural when the focus is put on objects, behaviors and interactions between them. However, using threads in this context is not so easy: for example, complex interactions between objects may demand complex thread synchronizations, and the number of components to simulate may exceed the number of available threads.

FairThreads can be helpful in several aspects.

- Simulation of large numbers of components is possible using automata. Automata do not need private stacks and the consumption of memory can thus stay low.
- Interactions can be expressed with broadcast events, which gives a very modular way to deal with them.
- Instants provide a common discrete time that can be used by the simulation.
- Interacting components can be naturally grouped into synchronized areas. The presence of several synchronized areas can be a plus for multiprocessing.

As an example, consider the simulation on screen of moving particles. A fair thread should be quite naturally associated with each particle for executing its behavior. An example of behavior could be to call at each instant two functions, one for inertia and one for bouncing on the borders of the screen. As the number of particles can be large, each particle should actually be implemented as an automaton. Particles that are close enough have to synchronize for collision processing. The needed synchronization is actually automatically provided by common instants shared by the fair threads linked to the same scheduler. Collision processing should use a broadcast event generated by each particle and processed by the others. To avoid considering distant particles during collision processing, the global simulation should be divided into several sub-regions which can be quite naturally mapped to distinct schedulers to which particles dynamically link according to their moves. In this way, each region gets its own collision event, and only particles present in the region are processed by it (of course, the collision of two particles belonging to different regions needs a special treatment, not considered here). Moreover, the schedulers can execute in real parallelism, on a multiprocessor machine. One thus gets a natural and efficient programming of the simulation, based on the synchronizations provided by instants. More generally, simulations appear as a domain which could certainly profit from the techniques proposed by FairThreads.

3. OVERVIEW OF THE API

An overview of the API of FairThreads is given in this section. All functions are presented, but, for simplicity, some details such as error codes are not considered here. The API is summarized in Appendix A.

3.1. Schedulers and threads

FairThreads explicitly introduces schedulers, of type `ft_scheduler_t`, which are created with the function `ft_scheduler_create`. Once started by a call to `ft_scheduler_start`, a scheduler is run by a dedicated native thread which cyclically gives the control in turn to the threads linked to it. Several schedulers can be used simultaneously in the same program. Using the `ft_scheduler_react` function, it is possible to execute only one instant of a scheduler. With this function, users can get control over schedulers' execution and, for example, synchronize several of them according to their needs.

Fair threads are of type `ft_thread_t` and are created with one of the two functions `ft_thread_create` or `ft_thread_create_unlinked`. The call `ft_thread_create (s,r,c,a)` creates in the scheduler `s` a thread run by a dedicated native thread. The creation is not immediate but becomes actual at the beginning of the next instant of `s`. The thread is automatically started and it executes the function `r` with `a` as parameter. If stopped (by `ft_scheduler_stop`), the thread switches execution to the function `c` to which `a` is also transmitted.

The call `ft_thread_create_unlinked(r,c,a)` creates an unlinked thread which executes the function `r` with `a` as parameter. As previously, the thread switches execution to `c` if it is stopped (which supposes that the thread has been linked to a scheduler by `ft_thread_link`, described later).

Here is a typical program main function which creates a scheduler and a fair thread in it, and then starts the scheduler (the call to `ft_exit` prevents the immediate termination of the whole program; it is considered later):

```
int main (void)
{
   ft_scheduler_t sched = ft_scheduler_create ();
   ft_thread_create (sched,t,NULL,NULL);
   ft_scheduler_start (sched);
   ft_exit ();
   return 0;
}
```

Orders can be given to a scheduler to stop, suspend, or resume a thread linked to it (an error is returned if the thread is actually unlinked). For example, the call `ft_scheduler_stop(t)` gives the scheduler `s` (which executes the thread `t`) the order to stop it. The stop will become actual at the beginning of the next instant of the scheduler, in order to ensure that `t` is in a stable state when stopped. In a similar way, a thread can be suspended and resumed with the functions `ft_scheduler_suspend` and `ft_scheduler_resume`.

The executing thread is returned by `ft_thread_self()` and the scheduler of the executing thread is returned by `ft_thread_scheduler()` (an error code is returned if the thread is unlinked).

For example, the following call is a way for a thread to stop itself:

```
ft_thread_stop (ft_thread_self ());
```

Note that this call does not prevent the executing thread from continuing execution during the current instant, as the stop becomes effective only at the beginning of the next instant. In the terminology of synchronous languages [3], the preemption resulting from `ft_thread_stop` is 'weak', not 'strong'.

3.2. Cooperation and termination

The call `ft_thread_cooperate()` is the explicit way for the calling thread to return control to the scheduler running it. An error code is returned if the executing thread is unlinked.

For example, the following function gives a way to trace the instants of a scheduler:

```
void trace_instants (void *n)
{
   int i = 0;
   while (1) {
      printf ("\ninstant %d: ", i++);
      ft_thread_cooperate();
   }
}
```

The call `ft_thread_cooperate_n(i)` is equivalent to `i` calls to `ft_thread_cooperate()`. Actually, `ft_thread_cooperate_n` is present in the API only for optimization purposes.

The call `ft_thread_join(t)` suspends the execution of the executing thread until the thread `t` terminates (either normally or because it is stopped). Note that `t` does not need to be linked to the scheduler of the calling thread. With `ft_thread_join_n(t,i)` the suspension is limited to `i` instants in the scheduler of the executing thread.

The following loop, for example, waits for the termination of all the components of an array of threads:

```
for (i = 0; i < MAX; i++)
   ft_thread_join (thread_array [i]);
```

3.3. Events

An event has type `ft_event_t` and is created with the function `ft_event_create`, which receives as a parameter the scheduler `s` in charge of it. Only threads linked to `s` will be able to generate the event, to await it, or to get its associated values. Nevertheless, it is always possible to generate the event from outside `s`, with `ft_scheduler_broadcast`.

The call `ft_thread_generate(e)` immediately generates the event `e` in the scheduler `s` in charge of it. An error code is returned if the executing thread is not linked to `s`. The call `ft_thread_generate_value(e,v)` adds `v` to the list of values associated to `e` during the current instant (these values can be read using `ft_thread_get_value`, considered later).

For example, the following instruction generates the event `presence` and associates the executing thread to it:

```
ft_thread_generate_value (presence,ft_thread_self ());
```

The call `ft_scheduler_broadcast(e)` gives to the scheduler `s` of the event `e` the order to broadcast it to all the linked threads. In this case it is not mandatory that the executing thread is linked to `s`, and it can even be unlinked. The call `ft_scheduler_broadcast_value(e,v)` associates the value `v` to `e` (as previously, `v` can be read using `ft_thread_get_value`).

Awaiting events

Events can be awaited using `ft_thread_await` (in the case of one single event) or `ft_thread_select` (in the case of several events). In all cases, the executing thread must be linked to the scheduler of awaited events, in order to get safe information about their presence or absence. Thus an error code is returned if the executing thread is not linked to the scheduler of awaited events.

The call `ft_thread_await(e)` suspends the execution of the calling thread until the event `e` becomes generated. Execution resumes as soon as `e` is generated.

Here, for example, is a function that waits for an event to be present and then stops a thread (`preempt_t` is a pointer type on a structure made of an event and a thread):

```
void killer (void *p)
{
   preempt_t p = p;
   ft_thread_await (p->event);
   stop (p->thread);
}
```

With `ft_thread_await_n(e,i)`, the waiting is limited to at most `i` instants: the executing thread is automatically resumed at the beginning of the `i`th next instant if `e` was not previously generated.

For example, the following code tests if `event` is present during the current instant (the executing thread is supposed to be correctly linked to the scheduler of the event):

```
if (OK == ft_thread_await_n (event,1)) printf ("present!");
else printf ("was absent!");
```

Note that message ‘was absent!’ is printed only at the next instant because to determine the absence of `event` takes the whole current instant. This is a major difference from Esterel [4]: in FairThreads instantaneous reaction to the absence of an event is impossible; only delayed reaction to absence is possible.

The call `ft_thread_select(k,array,mask)` suspends the execution of the calling thread until the generation of at least one element of `array`, which is an array of `k` events. Then, `mask`, which is an array of `k` Boolean values, is set accordingly. With `ft_thread_select_n(k,array,mask,i)`, the waiting is limited to `i` instants.

Getting event values

The call `ft_thread_get_value(e,i,r)` is an attempt to get the `i`*th* value associated to event `e` during the current instant (as previously, the executing thread must be linked to the scheduler of `e`). If such a value exists, it is assigned to the location pointed to by `r` and the call terminates instantly. Otherwise, the special code `ENEXT` is returned at the next instant.

For example, the following instruction waits for `event` and then gets all the values associated to it during the current instant:

```
ft_thread_await (event);
i = 0;
while (OK == ft_thread_get_value (event,i++,res)) {
     ...
}
```

An important point is that the loop does not terminate at the instant in which `event` is generated, but at the next one. Indeed, the fact that all values have been considered can only be known at the end of the current instant. Thus `ENEXT` is only returned at the next instant.

3.4. Linking, unlinking, and Pthreads

The call `ft_thread_unlink()` unlinks the executing thread `t` from the scheduler `s` in which it is running (an error code is returned if the executing thread is already unlinked). Then `t` is completely removed from `s` and it will no longer synchronize, instant after instant, with the other threads linked to `s`. Actually, after unlinking, `t` behaves as a standard native thread, only under the control of the OS. Note that if it later re-links to the scheduler, it does not keep its position and is put, as every new incoming thread, is at the end of the list of linked threads.

The call `ft_thread_link(s)` links the calling thread to the scheduler `s`. The calling thread must be unlinked when executing the call. The linkage becomes actual at the beginning of the next instant of `s`.

For example, the following function implements a cooperative reading I/O using the standard blocking `read` function. The thread first unlinks from the scheduler, then performs the read, and finally re-links to the scheduler:

```
ssize_t ft_thread_read (int fd,void *buf,size_t count)
{
   ft_scheduler_t sched = ft_thread_scheduler ();
   ssize_t res;
   ft_thread_unlink ();
   res = read (fd,buf,count);
   ft_thread_link (sched);
   return res;
}
```

In the presence of unlinked threads, locks can be needed to protect data shared between unlinked and linked threads. Standard mutexes are used for this purpose. The call `ft_thread_mutex_lock(p)`, where `p` is a mutex, suspends the calling thread until `p` becomes locked. The lock is released using `ft_thread_mutex_unlock`. Locks owned by a thread are automatically released when the thread terminates definitively or when it is stopped.

The call `ft_pthread(t)` returns the native pthread which executes the fair thread `t`. This function gives direct access to the Pthreads implementation of FairThreads.

The function `ft_exit` is equivalent to `pthread_exit`. The basic use of `ft_exit` is to terminate the pthread which is running the function `main`, without exiting from the process running the whole program.

3.5. Automata

Automata are fair threads of the type `ft_thread_t`, created with the function `ft_automaton_create`. The thread returned by `ft_automaton_create(s,r,c,a)` is executed as an automaton by the scheduler `s`, which means that it is run by the native thread of the scheduler and not by a dedicated native thread.

The automaton `r` is described as a list of numbered states coded using a set of macros described in Appendix A. States are numbered, starting from zero, and the numbers must be consecutive, without any gap in the numbering.

For example, here is an automaton equivalent to the function `killer`, previously defined:

```
DEFINE_AUTOMATON (killer)
{
   preempt_p p  = ARGS;
   BEGIN_AUTOMATON
      STATE_AWAIT (0,p->event)
      STATE (1) {
         ft_scheduler_stop (p->thread);
      }
   END_AUTOMATON
}
```

The automaton is introduced by the macro `DEFINE_AUTOMATON` with the automaton name as a parameter. The macro `ARGS` gives access to the argument given at creation. The list of states starts with `BEGIN_AUTOMATON` and ends with `END_AUTOMATON`, and the state numbered 0 is always the initial state. States are either standard states (introduced by `STATE`) or special states corresponding to some API functions. For example, the special state 0 of the previous automaton corresponds to a call of `ft_thread_await`. Actually, the control will stay in this state while the event is not present, and it will flow to the next state as soon as the event is generated. Passing from one state to another one can be made explicit using the macros `GOTO`, `GOTO_NEXT`, or `IMMEDIATE`. With `GOTO` and `GOTO_NEXT`, the execution of the target state will occur only at the next instant, while it is immediate when `IMMEDIATE` is used.

A fair thread instance of `killer` is created by:

```
ft_thread_t a = ft_automaton_create (sched,killer,NULL,args);
```

In contrast to a creation by `ft_thread_create`, no new pthread is created using `ft_automaton_create`, and the automaton is simply run by the pthread of the scheduler to which it is linked. Thus no supplementary thread context switch appears, which is a good point for efficiency. Moreover, limitations on the number of native threads that can be simultaneously running do not apply to automata.

4. EXAMPLES

Several examples are given which show various aspects of FairThreads. The example in Section 4.1 illustrates the determinism of linked threads. A producer/consumer example which can benefit from multiprocessor machines is described in Section 4.2. Section 4.3 shows the benefit of having precise semantics. Finally, several uses of automata are considered in Section 4.4.

4.1. Determinism

The following code is made of two threads linked to the same scheduler, and it outputs `Hello World!` cyclically. The whole code is given here for sake of completeness:

```
#include "fthread.h"
#include <stdio.h>

void print (void *txt)
{
   while (1) {
      printf ("%s", (char*)txt);
      ft_thread_cooperate ();
   }
}

int main (void)
{
  ft_scheduler_t sched = ft_scheduler_create ();
  ft_thread_create (sched,print,NULL,"Hello");
  ft_thread_create (sched,print,NULL," World!\n");
  ft_scheduler_start (sched);
  ft_exit ();
  return 0;
}
```

Note the call of `ft_exit` to prevent the program terminating before executing the two threads. Execution of linked fair threads is deterministic: the two messages `Hello` and `World!` are always printed in this order because the thread which prints `Hello` is created and linked to `sched` before the one which prints `World`.

4.2. Producer/consumer

A producer/consumer example follows. There are two files, `in` and `out` of type `file_t` (not detailed here), and a pool of threads that take data from `in`, process them, and then put results in `out`.

Processing a value is supposed to be time-consuming. A scheduler and an event are associated to each file; the event is generated to indicate that a new value is produced in the associated file:

```
file_t in = NULL, out = NULL;
ft_scheduler_t in_sched, out_sched;
ft_event_t new_input, new_output;
```

Processing values

Each cycle of the processing thread consists of the following steps. First the thread links to `in_sched` to get a value. Then it unlinks to process the value. When this is finished, it links to `out_shed` to deliver the result. Finally, the thread unlinks. The code is

```
void process (void *args)
{
  int v;
  while (1) {
     ft_thread_link (in_sched);
     while (size(in) == 0) {
        ft_thread_await (new_input);
        if (size (in) == 0) ft_thread_cooperate ();
     }
     v = get (&in);
     ft_thread_unlink ();
     < time consuming processing of v >
     ft_thread_link (out_sched);
     put (v,&out);
     ft_thread_generate (new_output);
     ft_thread_unlink ();
  }
}
```

The event `new_input` is used to prevent polling when no value is available from `in`. However, to test it as present does not imply that a value is available: it could happen that the value has already been consumed by another thread. This is the reason why file `in` is tested again, in sequence with `ft_thread_await`. Note the call to `ft_thread_cooperate` to avoid an infinite loop during the same instant if `new_input` is tested as present while no value is actually available¶.

Main function

Two schedulers are created: one for values to be processed, and the other for results. Then several unlinked processing threads are created. The main function is the following:

¶Using as many events as processing threads, one could also design a different solution in which only one thread would be awakened at a time.

```
int main (void)
{
   int i;
   in_sched  = ft_scheduler_create ();
   out_sched = ft_scheduler_create ();
   new_input  = ft_event_create (in_sched);
   new_output = ft_event_create (out_sched);
   for (i = 0; i < MAX_THREADS; i++)
         ft_thread_create_unlinked (process,NULL,NULL);
   ft_thread_create (in_sched,produce,NULL,NULL);
   ft_thread_create (out_sched,consume,NULL,NULL);
   ft_scheduler_start (in_sched);
   ft_scheduler_start (out_sched);
   ft_exit ();
   return 0;
}
```

Below we give some important points.

- While processing values, the processing threads are unlinked and can thus be run by distinct processors; the producer/consumer system can, in this way, benefit from multiprocessor machines.
- The use of two synchronized areas defined by the two schedulers is an alternative to the use of locks: no explicit lock is indeed needed despite the fact that all the processing threads share the two files `in` and `out`.
- It is possible, for processing values, to use a non-cooperative procedure provided it is thread-safe (and thus, reentrant). As the executing thread is unlinked, calling the procedure does not penalize the other threads which do not have to wait for its termination to start running.

4.3. Using events

Consider two threads `t1` and `t2`, and two events `e1` and `e2`. The thread `t1` awaits `e1` and `t2` awaits `e2`. When a thread receives the event it is waiting for, it stops the other thread and starts running:

```
ft_scheduler_t sched;
ft_thread_t t1,t2;
ft_event_t  e1,e2;
   ....

void run1 (void *args)
{
   ft_thread_await (e1);
   ft_scheduler_stop (t2);
   < body1 >
}

void run2 (void *args)
{
   ft_thread_await (e2);
   ft_scheduler_stop (t1);
   < body2 >
}
  ....
  sched = ft_scheduler_create ();
  t1 = ft_thread_create (sched,run1,NULL,NULL);
  t2 = ft_thread_create (sched,run2,NULL,NULL);
  e1 = ft_event_create (sched);
  e2 = ft_event_create (sched);
   ....
```

The question is: what happens when `e1` and `e2` are simultaneously present (perhaps, because `e1` and `e2` are the same event)? The answer is clear and precise, according to the semantics of FairThreads: `body1` and `body2` are executed during only one instant, and then `t1` and `t2` both terminate at the next instant. Note that, if one prefers `body1` and `body2` not to be executed at all, it is sufficient to insert a call to `ft_thread_cooperate` just after the call to `ft_scheduler_stop`, in both `run1` and `run2`.

Now, suppose that the same example is coded using standard pthreads instead of fair threads, replacing events by condition variables and `ft_scheduler_stop` by `pthread_cancel`. The resulting program is deeply non-deterministic. Actually, one of the two threads could prevent the other from execution and run its own body up to completion. However, the situation where both threads cancel each other is also possible; in this case, both bodies execute for a while, with an unpredictable result.

4.4. Automata examples

Consider three examples using automata. The first example is a recoding of the previous 'Hello World!' program. The second example is a three-state automaton which runs two threads in turn. The context of simulations, as presented in Section 2.5, is considered in the third example.

Hello World with automata

Three native threads are actually run by the program of Section 4.1: one for the scheduler and two instances of `print`. Using automata, one gets an equivalent program which needs only one native thread (the one of the scheduler). The use of automata, which clearly improves efficiency, is possible because the threads are never unlinked. The program becomes:

```
#include "fthread.h"
#include <stdio.h>

DEFINE_AUTOMATON (print)
{
  BEGIN_AUTOMATON
     STATE (0) {
        printf ("%s", (char*)ARGS);
        GOTO(0);
     }
  END_AUTOMATON
}
```

```
int main (void)
{
  ft_scheduler_t sched = ft_scheduler_create ();
  ft_automaton_create (sched,print,NULL,"Hello");
  ft_automaton_create (sched,print,NULL," World!\n");
  ft_scheduler_start (sched);
  ft_exit ();
  return 0;
}
```

Note the replacement of `ft_thread_create` by `ft_automaton_create` in the function `main`.

Two threads run in turn

The following automaton switches control between two threads, according to the presence of an event. The automaton `switch_aut` has three states. State 0 resumes the first thread (initially, one assumes that both threads are suspended). The switching event is awaited in the state 1, and the threads are switched when the event becomes present. State 2 is similar to state 1, except that the threads are exchanged:

```
DEFINE_AUTOMATON (switch_aut)
{
   void       **args    = ARGS;
   ft_event_t   event   = args[0];
   ft_thread_t  thread1 = args[1];
   ft_thread_t  thread2 = args[2];
   BEGIN_AUTOMATON
     STATE (0) {ft_scheduler_resume (thread1);}
     STATE_AWAIT (1,event) {
        ft_scheduler_suspend (thread1);
        ft_scheduler_resume  (thread2);
        GOTO(2);
     }
     STATE_AWAIT (2,event) {
        ft_scheduler_suspend (thread2);
        ft_scheduler_resume  (thread1);
        GOTO(1);
     }
   END_AUTOMATON
}
```

If a standard thread were used instead of an automaton, one supplementary pthread would be needed to perform the same task.

Simulation

Consider a simulation of colliding balls based on a matrix of schedulers to which the balls are linked. Each scheduler is in charge of a part of the global simulation and the balls dynamically link to the schedulers according to their coordinates. A special event is defined in each scheduler, which is generated by balls linked to it in order to signal their presence for collision processing.

Each ball is implemented as an automaton which, at each instant, moves and performs collisions with the other balls linked to the same scheduler. A specific scheduler is dedicated to graphics, and each ball broadcasts the `draw` event for being drawn on screen.

Balls have local variables of pointer type `ball_locals` with the following fields: `current` is the current area of the simulation in which the ball is; `presence` is the event which signals the presence of the ball, used for collision processing; `i`, `here` and `other` are auxiliary variables:

```
DEFINE_AUTOMATON(ball_fun)
{
   ball_locals ball = (ball_locals)ARGS;
   BEGIN_AUTOMATON
      STATE (0) {initialize (ball);}
      STATE (1) {
         move (ball);
         ball->here = where_is (ball);
         ft_scheduler_broadcast_value (draw,ball);
      }
      STATE (2) {
         if (ball->here == ball->current) IMMEDIATE (4);
         ball->current = ball->here;
      }
      STATE_LINK (3,scheduler_array[ball->here]);
      STATE (4) {
         ball->presence = collide_event_array[ball->current];
         ft_thread_generate_value (ball->presence,ball);
         ball->i = 0;
      }
      STATE_GET_VALUE (5, ball->presence, ball->i, (void**)&ball->other) {
         if (RETURN_CODE != OK) IMMEDIATE (1);
         if (ball != ball->other) collision (ball,ball->other);
         ball->i++;
         IMMEDIATE (5);
      }
   END_AUTOMATON
}
```

Below we give the description of the automaton states.

- State 0: the initialization of the ball.
- State 1: the ball is moved and the area in which it falls is stored in the auxiliary variable `here`. Moreover, the event `draw` is broadcast to the scheduler in charge of the graphics.
- State 2: if the ball stays in the same scheduler, then the control immediately goes to state 4. Otherwise, the current area is updated, and the control immediately flows to state 3.
- State 3: the special state in which the control stays until the automaton gets linked to the target scheduler.
- State 4: the event for signaling the presence of the ball in the current scheduler is generated. The ball is given as a value to the event.
- State 5: the special state to get and process the balls associated to the presence event. A possible collision is considered for all the other balls linked to the same scheduler. At the next instant, when all the balls have been considered (return code different from OK), the execution immediately returns to state 1.

One gets a simulation which can benefit from the presence of several processors, as schedulers can then run in parallel. Note, however, that the simulation described is only partial because collisions between balls belonging to distinct schedulers are not processed.

5. RELATED WORK

Thread libraries in C

Several thread libraries exist for C. Among them, the Pthreads Library [1] implements the POSIX standard for preemptive threads. LinuxThreads [5] is an implementation of Pthreads for Linux; it is based on native (kernel-level) threads. Quick Threads [6] provides programmers with minimal support for multithreading at user-space level. Basically, it implements context-switching in assembly code, and is thus a low-level solution to multithreading.

Gnu Portable Threads [7] (GNU Pth) is a library of purely cooperative threads which has portability as the main objective. The Next Generation POSIX Threading project [8] proposes to extend GNU Pth to the M:N model (M user threads, N native threads), with Linux SMP machines as the target. The M:N model is also the basis of the Solaris OS of Sun, where kernel objects of execution are called *light weight processes*. In Windows NT, threads are used at kernel level, but the unit of concurrency at user level is not the thread but the *fiber*; a comparison of Solaris and NT in the context of SMP is described in Zabatta and Ying [9].

Java threads

Java introduce threads at language level. Actually, threads are generally heavily used in Java, for example when graphics or networking is involved. No assumption is made of the way threads are scheduled (cooperative or preemptive schedulings are both possible), which makes Java multi-threaded systems difficult to program and to port [10]. This difficulty is pointed out by the suppression from the recent versions of the language of the primitives to gain fine control over threads [11]. A first version of FairThreads has been proposed in the context of the Java Language [12] in order to simplify concurrent programming in Java; this version was limited to cooperative threads.

Recently a new standard, called the *Real-Time Specification for Java* (RTSJ), has been proposed [13]. The aim of this standard is to extend Java to support real-time threads whose execution conforms to timing constraints. A central point addressed by RTSJ is the garbage collection (more precisely, RTSJ proposes constructs allowing certain real-time threads to circumvent the garbage collection).

Threads in functional languages

Threads are used in several ML-based languages such as CML [14]. CML is preemptively scheduled and threads, communication is synchronous and based on channels. Threads are also introduced in CAML [15]; they are implemented by time-sharing on a single processor, and thus cannot benefit from multiprocessor machines.

FairThreads has been recently introduced in the Bigloo [16] implementation of Scheme. The present version only supports linked threads, and special 'service threads' are introduced to deal with non-blocking cooperative I/Os.

Reactive approach

FairThreads actually comes from the so-called *reactive approach* [17], which is, itself, a ramification of *synchronous languages* [3]. Instants and broadcast events are issued from Esterel [4], a synchronous language for the specification of hardware and embedded systems. However, there are two main differences between reactive programming and FairThreads on one hand, and Esterel and the synchronous languages on the other hand. First, in the reactive approach, the absence of an event during one instant cannot be decided before the end of this very instant. As a consequence, reaction to absence is delayed to the next instant. This is a way to solve the so-called 'causality problems' which are raised by synchronous languages and are obstacles to modularity. Secondly, dynamic creation of concurrent components (of threads in the case of FairThreads) and of events is possible, while it is forbidden by synchronous languages in which the structure of programs is always static.

The Reactive-C [18] language was the first proposal for reactive programming in C; in this respect, FairThreads can be considered as a descendant of it.

A new approach has been recently proposed for the modeling and the simulation of physical systems, based on reactive programming. This approach is specially useful for modeling mixed continuous/discrete behaviors [19]. FairThreads can certainly be used with profit in this context.

Chores and filaments

Chores [20] and *filaments* [21] are small pieces of code that do not have a private stack and are never preempted. Chores and filaments are designed for fine-grained parallelism programming on shared-memory machines. Chores and filaments are completely executed and cannot be suspended or resumed. Generally, a pool of threads is devoted to execute them. Chores and chunk-based techniques are described in detail in the context of the Java language in Christopher and Thiruvathukal [22] and Hollub [10]. Automata in FairThreads are close to chores and filaments, but give programmers more freedom for direct coding of states-based algorithms. Automata are also related to *mode automata* [23] in which states capture the notion of a running mode in the context of the synchronous language Lustre [3].

Cohorts and staged computation

Cohort scheduling [24] dynamically reorganizes a series of computations on items in an input stream, so that similar computations on different items execute consecutively. Staged computation is intended to replace threads. In the staged model, a program is constructed from a collection of stages, and each stage has scheduling autonomy to control the order in which operations are executed. Stages are thus very close to instants of FairThreads, and cohort scheduling looks very much like cooperative scheduling. In the staged model, emphasis is put on the way in which to exploit program locality by grouping similar operations in cohorts that are executed at the same stage; in this way, cohorts and staged computations fall into the family of data-flow models.

6. CONCLUSION

Multiprocessing

In FairThreads, users have control over the way threads are scheduled. Fair threads which are linked to a scheduler are scheduled in a cooperative way by it. When a fair thread unlinks from a scheduler, it becomes an autonomous native thread which can be run in real parallelism, on a distinct processor. An important point is that FairThreads provides users with *programming primitives* allowing threads to dynamically link to schedulers and to dynamically unlink from them.

Precise semantics

Linked threads have a precise and clear semantics (the formal semantics of the cooperative part of FairThreads is given in [25]). The point is that systems exclusively made of threads linked to one unique scheduler are completely *deterministic*.

Simplicity

FairThreads offers a very simple framework for concurrent and parallel programming. Simple cooperative systems can be coded without the need of locks to protect data. Instants give automatic synchronizations that can also simplify programming in certain situations.

Compatibility with Pthreads

FairThreads is fully compatible with the standard Pthreads library. Indeed, unlinked fair threads are actually just pthreads. In this respect, FairThreads is basically an extension of Pthreads, which allows users to define cooperative contexts, with a clear and simple semantics, in which threads execute at the same pace and events are instantaneously broadcast.

Automata

Auxiliary tasks can be implemented using automata instead of standard fair threads. Implementation of an automaton is lightweight and does not require a dedicated native thread. Automata are useful for short-lived small tasks or when a large number of tasks is needed. Automata are an alternative to techniques such as 'chunks' or 'chores', sometimes used in thread-based programming.

Implementation

A first implementation of FairThreads in C is available (under the *Gnu General Public License*) as a library called `fthread` [25], which must be used with the standard Pthreads library.

Future work

A language (named *Language Over Fair Threads*) is under development which should provide users with a real syntax for programming with fair threads. In particular, some difficulties of automata coding (mainly the use of macros) should disappear and the use of automata should become transparent to the programmer.

An implementation of cellular automata based on FairThreads is also under development. Related to this work, experiments are made with nondeterministic schedulers which do not preserve the order in which threads are selected for execution (while, of course, preserving the existence of instants and the broadcasting of events). When large numbers of threads are considered, nondeterministic schedulers are more efficient than standard deterministic ones. Nondeterministic schedulers actually introduce an intermediate level between the complete determinism of standard schedulers and the total nondeterminism of the OS.

APPENDIX A. API SUMMARY

Creation of schedulers, threads, and events

`ft_scheduler_t ft_scheduler_create (void)`	Creation of a scheduler
`ft_thread_t ft_thread_create ( ft_scheduler_t, void (*runnable)(void*), void (*cleanup)(void*), void *args)`	Creation of a linked fair thread run by a native thread
`ft_thread_t ft_thread_create_unlinked ( void (*runnable)(void*), void (*cleanup)(void*), void *args)`	Creation of an unlinked fair thread
`ft_thread_t ft_automaton_create ( ft_scheduler_t, void (*automaton)(ft_thread_t), void (*cleanup)(void*), void *args)`	Creation of a fair thread run as an automaton
`ft_event_t ft_event_create (ft_scheduler_t)`	Creation of an event

Control over schedulers

`int ft_scheduler_start (ft_scheduler_t)`	The scheduler is cyclically executed by a native thread
`void ft_scheduler_react (ft_scheduler_t)`	Only one instant of the scheduler is executed

Control over threads

`int ft_scheduler_stop (ft_thread_t)`	Stops a thread linked to a scheduler
`int ft_scheduler_suspend (ft_thread_t)`	Suspends a thread linked to a scheduler
`int ft_scheduler_resume (ft_thread_t)`	Resumes a thread linked to a scheduler

Cooperation and termination

`int ft_thread_cooperate (void)`	Cooperation
`int ft_thread_cooperate_n (int num)`	Cooperation during exactly `num` instants
`int ft_thread_join (ft_thread_t)`	Joining a thread
`int ft_thread_join_n (ft_thread_t,int timeout)`	Limited join

Link and unlink

`int ft_thread_link (ft_scheduler_t)`	Thread linking to a scheduler
`int ft_thread_unlink (void)`	Thread unlinking

Generating, broadcasting and getting values of events

`int ft_thread_generate (ft_event_t)`	Generation of an event
`int ft_thread_generate_value ( ft_event_t,void *value)`	Generation of an event with an associated value
`int ft_scheduler_broadcast (ft_event_t)`	Order to broadcast an event
`int ft_scheduler_broadcast_value (ft_event_t, void *value)`	Order to broadcast an event with an associated value
`int ft_thread_get_value (ft_event_t event, int n,void **result)`	Attempt to get the `n`th value associated to an event

Awaiting events

`int ft_thread_await (ft_event_t)`	Waiting for an event
`int ft_thread_await_n (ft_event_t,int timeout)`	Limited waiting for an event
`int ft_thread_select (int len, ft_event_t *array,int *mask)`	Waiting for several events
`int ft_thread_select_n ( int len,ft_event_t *array, int *mask,int timeout)`	Limited waiting for several events

Automaton structure

`AUTOMATON(aut)`	Declares the automaton `aut`
`DEFINE_AUTOMATON(aut)`	Starts definition of the automaton `aut`
`BEGIN_AUTOMATON`	Starts the list of states
`END_AUTOMATON`	Ends the list of states

Explicit control

`GOTO(num)`	Blocks execution for current instant; next state is state `num`
`GOTO_NEXT`	Blocks execution for current instant and sets the next state to be the successor of the current state
`IMMEDIATE(num)`	Execution jumps to state `num` which is immediately executed
`RETURN`	Immediately terminates the automaton

States

`STATE(num)`	Standard state
`STATE_AWAIT(num,event)`	State to await `event`
`STATE_AWAIT_N(num,event,delay)`	States to await `event` during at most `delay` instants
`STATE_JOIN(num,thread)`	State to join `thread`
`STATE_JOIN_N(num,thread,delay)`	State to join `thread` during at most `delay` instants
`STATE_STAY(num,n)`	State to sleep for `n` instants
`STATE_GET_VALUE(num,event,n,result)`	State to get the `n`th value associated to `event`
`STATE_SELECT(num,n,array,mask)`	Generalizes `STATE_AWAIT` to an array of `n` events
`STATE_SELECT_N(num,n,array,mask,delay)`	Generalizes `STATE_AWAIT_N`
`STATE_LINK(num,sched)`	Atomically re-links the automaton to `sched` (nothing done if `sched` is actually the current scheduler)

Special automaton variables

`SELF`	The automaton
`LOCAL`	Local data of the automaton
`SET_LOCAL(data)`	Sets the local data of the automaton
`ARGS`	Argument which is passed at creation to the automaton
`RETURN_CODE`	Error code set by macros during automaton execution

Miscellaneous

`ft_thread_t ft_thread_self (void)`	The executing fair thread
`ft_scheduler_t ft_thread_scheduler (void);`	The scheduler of the executing fair thread
`void ft_exit (void)`	The actual pthread is exited
`int ft_thread_mutex_lock ( pthread_mutex_t *mutex)`	Mutex lock
`int ft_thread_mutex_unlock ( pthread_mutex_t *mutex)`	Mutex unlock
`pthread_t ft_pthread (ft_thread_t thread)`	The underlying pthread

ACKNOWLEDGEMENTS

Many thanks to Christian Brunette, Fabrice Peix, Jean-Ferdy Susini, and Olivier Tardieu for their valuable comments and suggestions.

REFERENCES

1. Nichols B, Buttlar D, Proulx FJ. *Pthreads Programming*. O'Reilly: Sebastopol, CA, 1996.
2. Buhr PA, Fortier M, Coffin MH. Monitor classification. *ACM Computing Surveys* 1995; **27**(1):63–107.
3. Halbwachs N. *Synchronous Programming of Reactive Systems*. Kluwer Academic: New York, 1993.
4. Berry G, Gonthier G. The Esterel synchronous programming language: Design, semantics, implementation. *Science of Computer Programming* 1992; **19**(2):87–152.
5. LinuxThreads. http://pauillac.inria.fr/∼xleroy/linuxthreads/.
6. Keppel D. Tools and techniques for building fast portable threads packages. *Technical Report UWCSE 93-05-06*, University of Washington, 1993.
7. Engelschall RS. Portable multithreading. *Proceedings of the USENIX Annual Technical Conference*, San Diego, CA, 2000. USENIX Association: Berkeley, CA, 2000.
8. Next Generation POSIX Threading. http://oss.software.ibm.com/developerworks/opensource/pthreads.
9. Zabatta F, Ying K. A threads performance comparison: Windows NT and Solaris on a symmetric multiprocessor. *Proceedings of the 2nd USENIX Windows NT Symposium*, Seattle, WA, 1998. USENIX Association: Berkeley, CA, 1998.
10. Holub A. *Taming Java Threads*. Apress, 2000.
11. Java. http://java.sun.com.
12. Boussinot F. Java Fair Threads. *INRIA Research Report RR-4139*, INRIA, 2001.
13. Real-time for Java expert group. *Real-Time Specification for Java*. Addison-Wesley: Reading, MA, 2000.
14. Reppy JH. *Concurrent Programming in ML*. Cambridge University Press: Cambridge, 1999.
15. CAML. http://caml.inria.fr/ocaml/.
16. Bigloo. http://www.inria.fr/mimosa/fp/Bigloo.
17. Reactive Programming. http://www-sop.inria.fr/mimosa/rp.
18. Boussinot F. Reactive C: An extension of C to program reactive systems. *Software—Practice and Experience* 1991; **21**(4):401–428.
19. Simulations in Physics. http://www-sop.inria.fr/mimosa/rp/SimulationInPhysics.
20. Eager DL, Zahorjan J. Chores: Enhanced run-time support for shared memory parallel computing. *ACM Transactions on Computer Systems* 1993; **11**(1):1–32.
21. Lowenthal DK, Freech VW, Andrews GR. Efficient support for fine-grain parallelism on shared-memory machines. *Technical Report 96-1*, University of Arizona, 1996.
22. Christopher TW, Thiruvathukal GK. *High Performance Java Platform Computing: Multithreaded and Networked Programming* (*Sun Microsystems Press Java Series*). Prentice-Hall: Englewood Cliffs, NJ, 2001.
23. Maraninchi F, Remond Y. Running-modes of real-time systems: A case-study with mode-automata. *Proceedings of the 12th Euromicro Conference on Real-Time Systems*, Stockholm, Sweden, 2000.
24. Larus JR, Parkes M. Using cohort scheduling to enhance server performance. *Proceedings of the USENIX Conference*, Monterey, CA, 2002. USENIX Association: Berkeley, CA, 2002; 103–114.
25. FairThreads. http://www-sop.inria.fr/mimosa/rp/FairThreads.