FairThreads: mixing cooperative and preemptive threads in C

Frédéric Boussinot

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

FairThreads: mixing cooperative and preemptive threads in C

Frédéric Boussinot

N° 5039

Décembre 2003

THÈME 1

ISSN 0249-6399 ISRN INRIA/RR--5039--FR+ENG

INRIA
SOPHIA ANTIPOLIS

FairThreads: mixing cooperative and preemptive threads in C

Frédéric Boussinot

Thème 1 — Réseaux et systèmes
Projet Mimosa

Rapport de recherche n° 5039 — Décembre 2003 — 28 pages

Abstract: FairThreads offers a very simple framework for concurrent and parallel programming. Basically, it defines *schedulers* which are synchronization servers, to which *fair threads* are linked. All threads linked to the same scheduler are executed in a cooperative way, at the same pace, and they can synchronize and communicate using broadcast events. Threads which are not linked to any scheduler are executed by the OS in a preemptive way, at their own pace.

FairThreads 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 limitations of native threads.

The implementation in C is based on the pthreads library. Several fair schedulers, executed by distinct pthreads, can be used simultaneously in the same program. Using several schedulers and unlinked threads, programmers can take benefit of multiprocessors machines (basically, SMP architectures)

Key-words: Concurrent programming, Threads, Reactive programming, Parallelism

Unité de recherche INRIA Sophia Antipolis
2004, route des Lucioles, BP 93, 06902 Sophia Antipolis Cedex (France)
Téléphone : +33 4 92 38 77 77 — Télécopie : +33 4 92 38 77 65

FairThreads:
un mélange de threads coopératifs et préemptifs en C

Résumé : Les FairThreads proposent un moyen simple de programmation concurrente et parallèle en C. Ils introduisent la notion de scheduler qui peut être vu comme un serveur de synchronisation pour les fair threads qui lui sont liés. Les threads liés à un même scheduler sont exécutés avec une stratégie coopérative, au même rythme, en pouvant se synchroniser et communiquer à l'aide d'événements diffusés. Les threads qui ne sont liés à aucun scheduler sont sous le contrôle de l'OS uniquement et ils s'exécutent de manière préemptive, à un rythme qui leur est propre.

Des automates sont définis dans les FairThreads pour les tâches à courte durée de vie qui ne nécessitent pas toute la puissance des threads natifs. Les automates nécessitent peu de ressources et ne sont pas soumis à certaines limitations des threads natifs.

L'implémentation en C utilise la librairie Pthreads. Plusieurs schedulers, exécutés par des threads natifs distincts, peuvent cohabiter au sein de la même application. En utilisant plusieurs schedulers et threads non liés, le programmeur peut profiter du parallélisme fourni par les machines multi-processeurs (par exemple, les architectures SMP).

Mots-clés : Programmation concurrente, Programmation réactive, Threads, Parallélisme

1 Introduction

Threads are generally considered as having two main advantages: first, programs can be run by multiprocessor machines without any change. Thus, multithreaded systems immediately take benefit from *symetric multiprocessing* (SMP) architectures, which become now widely available. Second, blocking I/Os do not need special attention of any kind. Indeed, as the scheduler is preemptive, there is no risk that a thread blocked forever on an I/O operation will also block the rest of the system.

The benefit of using threads is however not so clear for systems made of tasks needing strong synchronizations or a lot of communications. Indeed, in a preemptive context, to communicate or to synchronize generally implies the need to protect some data involved in the communication or in the synchronization. Locks are often used for this purpose, but they have a cost and are error-prone (possibilities of deadlocks). Pure cooperative threads are more adapted for highly communicating tasks. Indeed, data protection is no more 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, as, 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. An other 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, specially when large numbers of short-lived components are needed. Among these are thread-pooling, to limit the number of created threads, and the use of small pieces of code, sometimes called *chores* or *chunks*, which can be executed in a simpler way than threads are.

1.1 The FairThreads Proposal

FairThreads proposes to overcome the difficulties of threads by giving users the possibility 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. 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 limitations of native threads.

This paper describes FairThreads in the context of C, implemented on top of the Pthreads library [22]. The structure is as follows: section 2 presents the rationale for the design of FairThreads. An overview of the 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 (hence the name FairThreads; note however that the word "fair" has a different meaning in the context of concurrency). 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[1]. Indeed, as the scheduling is cooperative, there is no risk for the thread to be

[1] In this respect, schedulers are quite close to standard monitors (see the classification of [12]) except that they need a dedicated thread of control.

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 in several threads and areas to which threads can link dynamically, following their needs. Moreover, a thread can be unlinked, that is 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 OS, and are thus just standard preemptive threads. Data shared by unlinked threads have to be protected by locks, in the standard way.

2.2 Cooperative Aspects

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. The scheduler to which the fair thread is linked gives it the possibility to get the processor. All threads linked to the scheduler get 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 which 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 an other thread linked to the scheduler.

A fair scheduler actually *broadcasts* 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.

Modularity

Events are a powerful synchronization and communication mean which simplifies concurrent programming while reducing risks 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). 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 always receive it during the same very instant it is generated; thus, a

thread leaving the control on a cooperation point does not risk to lose an event generated later in the same instant, as the scheduler will necessary resume it.

Determinism

Cooperative frameworks are less indeterministic than preemptive ones, as in cooperative frameworks preemption cannot occurs in an uncontrolled way. Actually, FairThreads puts the situation to an extreme point, when considering linked threads: linked threads are chosen for execution following a strict round-robin algorithm and the order is the one in which the threads have been linked to the scheduler. This leads to deterministic systems and can be a great help in programming and debugging.

Absence of Priorities

Priorities are meaningless for linked threads which always have equal rights to execute. Absence of priorities also contributes to simplify programming.

2.3 Automata

FairThreads proposes *automata* to deal with auxiliary tasks, such as waiting for an event to stop a thread, that do not need the full power of a dedicated native thread to execute. An automaton is a special fair thread which is always linked to a scheduler and which executes using its native thread. Thus, an automaton does not have its own execution stack to store its execution state or local variables. As a consequence, it can be implemented more efficiently than threads are but its expressive power is limited (for example, recursive functions that are not tail-recursive cannot be coded using automata).

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. Execution leaves the current state when an explicit jump to another state is executed. When the 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, only the coarse-grain sequentiality of states execution is.

2.4 Mapping to Native Threads

In FairThreads all fair threads, except automata, are mapped onto native threads. Fair threads which are linked to a scheduler are under the control of it, while unlinked threads behave as standard native preemptive threads. Unlinked threads are present 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/Os, programming would become problematic. Second, unlinked threads can be run by distinct processors. The use of unlinked threads is a plus in multiprocessor contexts.

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 (e.g. Runge-Kutta algorithms).

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 threads 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 example, considers the simulation on screen of moving particles. A fair thread should be quite naturally associated to 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 have to synchronize for collision processing. The needed synchronization is actually automatically provided by common instants shared by fair threads linked to the same scheduler. Collision processing should use a broadcast event generated by each particle and processed by all of them. 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, only processed by particles present in the region.

Using standard threads, the coding would be very different. First, a pool of processing threads should be defined, in order to benefit from multiprocessing. Then, particles should be placed in a common place (list or array), in which the processing threads pick them up. The shared common place should be protected for concurrent accesses. Particles should also be protected to avoid concurrent accesses of two threads processing simultaneously two colliding particles. Moreover, in order to avoid the quadratic processing of collisions, particles should actually be grouped into several areas corresponding to proximity considerations.

A new approach has been recently proposed for modelling and simulation of physical systems, based on reactive programming (considered in section 5). This approach is specially useful for modeling mixed continuous/discrete behaviors [8]. FairThreads can certainly be used with profit in this context.

3 Overview of the API

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

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 the 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`. A thread run by a dedicated native thread is created by `ft_thread_create(s,r,c,a)`; the thread is linked to the scheduler `s`. 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 to 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 assure 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 to continue execution during the current instant, as the stop become effective only at the beginning of the next instant. In the terminology of synchronous languages [16], the preemption resulting from `ft_thread_stop` is a "weak" one, not a "strong" one.

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 `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` needs not to be linked to the scheduler of the calling thread. With `ft_thread_join_n(t,i)` the suspension is limited to `i` instants.

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 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, one always has the possibility 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. `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` (case of one single event) or `ft_thread_select` (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 is for example 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;
   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 "was absent!" is printed only at next instant because to determine the absence of `event` takes the whole current instant. This is a major difference with Esterel [9]: 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 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 the 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 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:

```
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` can only be 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 control of the operating system. Note that in the case where it will later re-link to the scheduler, it does not keep its position and will be put, as every new incoming threads, 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 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 to the function `ft_thread_mutex_lock(p)`, where `p` is a mutex, suspends the calling thread until `p` becomes locked. The lock is released using the function `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` which are created using 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 annex. States are numbered, starting from 0, and the numbers must be consecutive, without any hole 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 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 by 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 an other 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 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 opposition to a creation by `ft_thread_create`, no new pthread is created when the fuction `ft_automaton_create` is used, 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

One considers several examples which show various aspects of FairThreads. The example of 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 interest to have precise semantics. Finally, several uses of automata are considered in section 4.4.

4.1 Determinism

The following code is a complete example, made of two threads linked to the same scheduler.

```
#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;
  }
```

The program outputs `Hello World!` cyclically. Note the call of `ft_exit` to prevent the program to terminate before executing the two threads. Execution of linked fair threads is round-robin and deterministic: the two messages `Hello` and `World!` are always printed in this order because the thread printing `Hello` is created and linked to `sched` before the one printing `World`.

4.2 Producer/Consumer

One implements a producer/consumer example. There are 2 files, `in` and `out` of type `file_t`, 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 in the following steps: first the thread links to `in_sched` to get a value; then, it unlinks to process the value and when this is terminated, it links to `out_shed` to deliver the result; finally, the thread unlinks. The code is the following:

```
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 implies that a value is available: it could happen that the value has already been consumed by another thread. This is the reason why `in` is tested again, in sequence of `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[2].

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:

```
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);
```

[2] Using as many events as processing threads, one could also design a different solution in which only one thread would be awaken at a time.

```
   ft_exit ();
   return 0;
}
```

Here are 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

One considers 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, in the same situation, 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 the function `ft_scheduler_stop` by `pthread_cancel`. The resulting program is deeply non-deterministic. Actually, one of the two threads could cancel the other and run its body up to completion. But the situation where both threads cancel the other one is also possible; in this case, both bodies can execute while the threads are not canceled, which produces unpredictable results.

4.4 Automata Examples

One considers three examples using automata. The first example is a recoding of the previous "Hello World!" program. The second example is a 3-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 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;
}
```

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

One considers 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 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 part of the simulation in which the ball is; `presence` is the event which signals the presence of the ball, used for collision processing; `i`, `area` 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->area = where_is (ball);
}
STATE (2) {
   if (ball->area == ball->current) { IMMEDIATE (4); }
   else { ball->current = ball->area; }
}
STATE_LINK (3,scheduler_array[ball->current]);
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) {
      ft_scheduler_broadcast_value (draw,ball);
      GOTO (1);
   }
   if (ball != ball->other) collision (ball,ball->other);
   ball->i++;
   IMMEDIATE (5);
}
END_AUTOMATON
}
```

One thus 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 as collisions between balls belonging to distinct schedulers are not considered.

5 Related Work

Thread Libraries in C

Several thread libraries exist for C. Among them, the Pthreads library [22] 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 [18] provides programmers with a minimal support for multithreading at user-space level. Basically, it implements context-switching in assembly code, and is thus a low-level solution to multi-threading.

Gnu Portable Threads [15] (GNU Pth) is a library of purely cooperative threads which has portability as main objective. The Next Generation POSIX Threading project [6] proposes to extend GNU Pth to the M:N model (M user threads, N native threads), with Linux SMP machines as target. The M:N model is also at 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 [25].

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 on the way threads are scheduled (cooperative or preemptive schedulings are both possible) which makes Java multi-threaded systems difficult to program and to port [17]. This difficulty is pointed out by the suppression from the recent versions of the language of the primitives to gain fine control over threads [4]. A first version of FairThreads has been proposed in the context of the Java language [11] in order to simplify concurrent programming in Java; this version was limited to cooperative threads.

Recently, a new standard, called *Real-Time Specification for Java* (RTSJ) has been proposed [23]. The aim of this standard is to extend Java to support real-time threads whose execution conforms to timing constraints. A central point adressed by RTSJ is the garbage collection.

Threads in Functional Languages

Threads are used in several ML-based languages such as CML [24]. CML is preemptively scheduled and threads communication is synchronous and based on channels. Threads are also introduced in CAML [2]; 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 [1] 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 out from the so-called *reactive approach* [7] which is itself a ramification of *synchronous languages* [16]. Instants and broadcast events are issued from Esterel [9], a synchronous language for the specification of harware and embedded systems. However, there are two main differences between reactive programming and FairThreads on one hand, and Esterel and the synchronous language 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. Second, 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 [10] language was the first proposal for reactive programming in C; in this respect, FairThreads can be considered as a descendant of it.

Chores and Filaments

Chores [14] and *filaments* [20] are small pieces of code that do not have private stack and are never preempted. Chores and filaments are designed for fine-grain parallelism programming on shared-memory machines. Chores and filaments are completely executed and cannot be suspended nor resumed. Generally, a pool of threads is devoted to execute them. Chores and chunk-based techniques are described in details in the context of the Java language in [13] and [17]. 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* [21] in which states capture the notion of a running mode in the context of the synchronous language Lustre [16].

Cohorts and Staged Computation

Cohort scheduling [19] dynamically reorganizes series of computations on items in an input stream, so that similar computations on different items execute consecutively. Staged computation intends 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 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 in the familly of data-flow models.

6 Conclusion

Multiprocessing

In FairThreads, users have control on 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 [3]). 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` [3] which must be used with the standard Pthreads library.

Acknowledgments

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

References

[1] **Bigloo Web Site** – `http://www.inria.fr/mimosa/fp/Bigloo`.

[2] **CAML Web Site** – `http://caml.inria.fr/ocaml/`.

[3] **FairThreads Web Site** – `http://www-sop.inria.fr/mimosa/rp/FairThreads`.

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

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

[6] **Next Generation POSIX Threading Web Site** – `http://oss.software.ibm.com/developerworks/opensource/pthreads`.

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

[8] **Simulations in Physics** – `http://www-sop.inria.fr/mimosa/rp/SimulationInPhysics`.

[9] Berry, G. and Gonthier G. – **The Esterel Synchronous Programming Language: Design, Semantics, Implementation** – *Science of Computer Programming*, 19(2), 1992, pp. 87-152.

[10] Boussinot, F. – **Reactive C: An Extension of C to Program Reactive Systems** – *Software-Practice and Experience*, 21(4), 1991.

[11] Boussinot, F. – **Java Fair Threads** – *Inria research report, RR-4139*, 2001.

[12] Buhr, P. A. and Fortier, M. and Coffin, M. H – **Monitor Classification** – *ACM Computing Surveys*, 27(1), 1995, pp. 63-107.

[13] Christopher, Thomas W. and Thiruvathukal, George K. – **High Performance Java Platform Computing: Multithreaded and Networked Programming** – *Sun Microsystems Press Java Series, Prentice Hall*, 2001.

[14] Eager, Derek L. and Zahorjan, John – **Chores: Enhanced run-time support for shared memory parallel computing** – *ACM Transaction on Computer Systems*, 11(1), 1993.

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

[16] Halbwachs, Nicolas – **Synchronous Programming of Reactive Systems** – *Kluwer Academic Publishers, New York*, 1993.

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

[18] Keppel, D. – **Tools and Techniques for Building Fast Portable Threads Packages** – *Technical Report UWCSE 93-05-06, University of Washington*, 1993.

[19] Larus, James R. and Parkes Michael. – **Using Cohort Scheduling to Enhance Server Performance** – *Proc. of USENIX Conference, Monterey Cal.*, 2002, pp. 103-114.

[20] Lowenthal, David K. and Freech, Vincent W. and Andrews, Gregory R. – **Efficient Support for Fine-Grain Parallelism on Shared-Memory Machines** – *TR 96-1, University of Arizona*, 1996.

[21] Maraninchi, F. and Remond, Y. – **Running-Modes of Real-Time Systems: A Case-Study with Mode-Automata** – *Proc. 12th Euromicro Conference on Real-Time Systems, Stockholm, Sweden*, 2000.

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

[23] Real-Time for Java Expert Group – **Real-Time Specification for Java** – *Addison-Wesley, ISBN 0-201-70323-8*, 2000.

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

[25] Zabatta, F. and Ying, K. – **A Threads Performance Comparison: Windows NT and Solaris on a Symetric Multiprocessor** – *Proc. 2nd USENIX Windows NT Symposium, Seattle*, 1998.

Annex: 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 and Broadcasting 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

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

Getting Values of Events	
`int ft_thread_get_value (ft_event_t event,` `int n,void **result)`	attempt to get the `n`th value associated to an event

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

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)`	re-links the automaton to `sched`

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

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

Miscelaneous	
`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

Unité de recherche INRIA Sophia Antipolis
2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex (France)

Unité de recherche INRIA Futurs : Parc Club Orsay Université - ZAC des Vignes
4, rue Jacques Monod - 91893 ORSAY Cedex (France)
Unité de recherche INRIA Lorraine : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex (France)
Unité de recherche INRIA Rennes : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)
Unité de recherche INRIA Rhône-Alpes : 655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier (France)
Unité de recherche INRIA Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399