Fair Threads in C

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

June 4, 2002

Abstract

FairThreads offers a very simple framework for concurrent and parallel programming. Basically, it defines *schedulers* which are synchronization servers, to which threads can dynamically link or unlink. 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** offers programming constructs for linking and unlinking threads. **FairThreads** is fully compatible with the standard **Pthreads** library and has a precise and clear semantics for its cooperative part; in particular, systems exclusively made of threads linked to one unique scheduler are actually completely deterministic. Special threads, called automata, are provided for short-lived small tasks or when a large number of tasks is needed. Automata do not need the full power of a native thread to execute and thus consume less resources.

1 Introduction

Threads are generally considered to be well adapted for systems made of heavy-computing tasks run in a preemptive context and needing few communications and synchronizations. This is typically the case of Web servers, in which a thread (usually picked out from a pool of available threads) is associated to each new request. In such contexts, advantages of threads are clear:

- Modularity is increased, as threads can be naturally used for coding independant sub-parts of the system.
- Programs can be run by multiprocessors machines without any change. Thus, multithreaded systems immediately take benefit from SMP architectures, which become now widely available.
- 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.

Difficulties of Threads

The benefit of using threads is less 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 (sometimes called *green-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. Moreover, they need special means to deal with blocking I/O.
Actually, programming with threads is difficult because threads generally have very "lose" semantics. This is particularly true with preemptive threads because their semantics strongly relies on the scheduling policy. The semantics of threads also depends on others aspects, as, for example, the way threads priorities are mapped at the kernel level.
Threads take time to create, and need a rather large amount of memory to execute. Moreover, the number of native threads than can be created is often limited by the system. Several techniques can be used to get round these problems, specially when large numbers of short-lived components are needed. Among these techniques 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.

The Fair Threads Proposal

FairThreads proposes to overcome the difficulties of threads by giving users the possibility to chose 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 for linking and unlinking threads.

FairThreads has the following main characteristics:

- It allows programs to benefit from multiprocessors 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 submitted to some limitations that native threads have.

This paper describes **FairThreads** in the context of C, implemented on top of the **Pthreads** library.
The rest of the paper is organized as follows: section 2 presents rationale for the design of **FairThreads**. An overwiew of the API of **FairThreads** is given in section 3. Section 4 contains the full API. Some examples are described in section 5. Related work is considered in section 6. Finally, section 7 concludes the paper. Man pages of **FairThreads** are given in annex.

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 fair scheduler defines a kind of *synchronized area* made of cooperative threads running at the same pace, and communicating through broadcast events.

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 has first 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 for the thread to be 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 defined in the system. 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.

Cooperative Scheduling

Basically, a linked fair thread is a cooperative thread which can synchronize with other fair threads using events and 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, fair schedulers define *instants* during which all threads linked to it run up to their next cooperation point. There are only two kinds of cooperation points: explicit ones which are calls to the `cooperate()` function, and implicit ones where threads are waiting for events. A fair scheduler broadcasts events to all fair threads linked to it. Thus, all 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 synchronisation 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. 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 instant it is generated; thus, a thread leaving control on a cooperation point does not risk to loose an event generated later in the same instant.

Determinism

Cooperative frameworks are less undeterministic 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 which leads to deterministic systems. This can be a great help in programming and debugging.

No Priorities

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

Preemptive Scheduling

Basically, unlinked threads are standard native preemptive threads. They 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/Os, programming would become problematic. Second, unlinked threads can be run by distinct processors. The use of unlinked threads is a plus in multiprocessors contexts.

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 linked fair thread which executes using the native thread of the scheduler to which it is linked. Thus, an automaton does not have its own execution stack that it could use to store its execution state. As a consequence, it can be implemented more efficiently than threads are.
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 begining 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.
Events can be used without restriction in automata. There is a special state to await an event: execution stays in this state until the event is generated.

3 API Overview

3.1 Creation

Schedulers

FairThreads explicitely introduces schedulers, of type `ft_scheduler_t`. Before being used, a scheduler must be created by calling the function `ft_scheduler_create`.
In order to be executed, a scheduler must be started by a call to the function `ft_scheduler_start`. Note that several schedulers can be used without problem simultaneously in the same program.

Threads

Fair threads are of type `ft_thread_t`. The call `ft_thread_create(s,r,c,a)` creates a thread in the scheduler `s`. The thread is automatically started as soon as it is created. The function `r` is executed by the thread, and the parameter `a` is transmitted to it.

The function `c` is executed by the thread if it is stopped (by `ft_scheduler_stop`). The parameter `a` is also transmitted to it, if this happens.

Events

Events are of the type `ft_event_t`. An event is created by calling the function `ft_event_create` which takes as parameter the scheduler in charge of it.

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`. The function `r` executed by the automaton must be defined with the macro `DEFINE_AUTOMATON`.

3.2 Orders

Control of Threads

The call `ft_scheduler_stop(t)` gives to the scheduler which executes the thread `t` the order to stop it. The stop will become actual at the begining of the next instant of the scheduler, in order to assure that `t` is in a stable state when stopped.

The call `ft_scheduler_suspend(t)` gives to the scheduler which executes `t` the order to suspend `t`. The suspension will become actual at the begining of the next instant of the scheduler. The function `ft_scheduler_resume` is used to resume execution of suspended threads.

Broadcast of Events

The function `ft_scheduler_broadcast(e)` gives to the scheduler of the event `e` the order to broadcast it to all threads running in the scheduler. The event will be actually generated at the begining of the next instant of the scheduler. The call `ft_scheduler_broadcast_value(e,v)` associates the value `v` to `e` (`v` can be read using `ft_thread_get_value`).

3.3 Basic Primitives

Cooperation

The call `ft_thread_cooperate()` is the explicit way for the calling thread to return control to the scheduler running it. The call `ft_thread_cooperate_n(i)` is equivalent to a sequence of `i` calls `ft_thread_cooperate()`.

Termination

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

3.4 Managing Events

Generating Events

The call `ft_thread_generate(e)` generates the event `e` in the scheduler which was associated to it, when created. 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`).

Awaiting Events

The call `ft_thread_await(e)` suspends the execution of the calling thread until the generation of the event `e`. Execution is resumed as soon as the event is generated. With `ft_thread_await_n(e,i)`, the waiting takes at most `i` instants.

Selecting Events

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 takes at most `i` instants.

Getting Events 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. If such a value exists, it is returned in `r` and the call immediately terminates. Otherwise, the value **NULL** is returned at the next instant. The return code of the call indicates if the call was sucessful or not.

3.5 Linking

Link and Unlink

The call `ft_thread_unlink()` unlinks the calling thread `t` from the scheduler in which it was previously running. Then, `t` will no longer synchronize, instant after instant, with other threads linked to the scheduler. Actually, after unlinking, `t` behaves as a standard native thread.

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 begining of the next instant of `s`.

Locks

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 `ft_thread_mutex_lock(p)`, where `p` is a mutex, suspends the calling thread until the moment where `p` can be locked. The lock is released using `ft_thread_mutex_unlock`. Locks owned by a thread are automatically released when the thread terminates definitively or is stopped.

3.6 Automata

Automata are coded using macros. Here are the macros to define the automaton structure:

- `AUTOMATON(aut)` declares the automaton `aut`.
- `DEFINE_AUTOMATON(aut)` starts definition of the automaton `aut`.
- `BEGIN_AUTOMATON` starts the state list.
- `END_AUTOMATON` ends the state list.

The following macros start the state whose number is `num`:

- `STATE(num)` introduces a standard state.
- `STATE_AWAIT(num,event)` and `STATE_AWAIT_N(num,event,delay)` are states to await `event`.
- `STATE_JOIN(num,thread)` and `STATE_JOIN_N(num,thread,delay)` are states to join `thread`.
- `STATE_STAY(num,n)` is a state which keeps execution in it for `n` instants.
- `STATE_GET_VALUE(num,event,n,result)` is a state to get the nth value associated to `event`.
- **STATE_SELECT**(num,n,array,mask) and **STATE_SELECT_N**(num,n,array,mask,delay) generalise `STATE_AWAIT` and `STATE_AWAIT_N` to an array of events of length `n`.

Going from state to state is possible with:

- `GOTO(num)` blocks execution for the current instant and sets the state for the next instant to be state `num`.
- `GOTO_NEXT` blocks execution for the current instant and sets the state for the next instant to be the next state.
- `IMMEDIATE(num)` forces execution to jump to state `num` which is immediately executed.
- `RETURN` immediately terminates the automaton.

Finally, the following macros define some special variables:

- `SELF` is the automaton.
- `SET_LOCAL(data)` sets the local data of the automaton.

- `LOCAL` is the local data of the automaton.
- `ARGS` is the argument that is passed at creation to the automaton.
- `RETURN_CODE` is the error code set by macros run during automaton execution.

3.7 Miscelaneous

Current Thread

The calling thread is returned by `ft_thread_self()`.

Current Scheduler

The scheduler of the calling thread is returned by `ft_thread_scheduler()`.

Pthread

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**. In the rest of the paper, native thread and pthread will be considered as synonymous.

Exiting

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 whole process.

4 API

Constructors

```
ft_scheduler_t ft_scheduler_create (void);

ft_thread_t    ft_thread_create    (ft_scheduler_t scheduler,
                                    void (*runnable)(void*),
                                    void (*cleanup)(void*),
                                    void *args);

ft_thread_t    ft_automaton_create (ft_scheduler_t,
                                    void (*automaton)(ft_thread_t),
                                    void (*cleanup)(void*),
                                    void *args);

ft_event_t     ft_event_create     (ft_scheduler_t scheduler);
```

Starting a Scheduler

```
int ft_scheduler_start  (ft_scheduler_t scheduler);
```

Control of Threads

```
int ft_scheduler_stop      (ft_thread_t thread);
int ft_scheduler_suspend   (ft_thread_t thread);
int ft_scheduler_resume    (ft_thread_t thread);
```

Broadcast of Events

```
int ft_scheduler_broadcast        (ft_event_t event);
int ft_scheduler_broadcast_value  (ft_event_t event,void *value);
```

Cooperation

```
int ft_thread_cooperate    (void);
int ft_thread_cooperate_n  (int num);
```

Termination

```
int ft_thread_join    (ft_thread_t thread);
int ft_thread_join_n  (ft_thread_t thread,int timeout);
```

Generating Events

```
int ft_thread_generate        (ft_event_t event);
int ft_thread_generate_value  (ft_event_t event,void *value);
```

Waiting Events

```
int ft_thread_await    (ft_event_t event);
int ft_thread_await_n  (ft_event_t event,int timeout);
```

Selecting Events

```
int ft_thread_select    (int len,ft_event_t *array,int *mask);
int ft_thread_select_n  (int len,ft_event_t *array,int *mask,int timeout);
```

Getting Generated Values

```
int ft_thread_get_value  (ft_event_t event,int num,void **result);
```

Link and Unlink

```
int ft_thread_link    (ft_scheduler_t scheduler);
int ft_thread_unlink  (void);
```

Current Thread and Scheduler

```
ft_thread_t     ft_thread_self       (void);
ft_scheduler_t  ft_thread_scheduler  (void);
```

Exit

```
void ft_exit  (void);
```

Locks

```
int ft_thread_mutex_lock    (pthread_mutex_t *mutex);
int ft_thread_mutex_unlock  (pthread_mutex_t *mutex);
```

Pthreads

```
pthread_t ft_pthread  (ft_thread_t thread);
```

Macros for Automata

```
AUTOMATON(name)
DEFINE_AUTOMATON(name)
BEGIN_AUTOMATON
END_AUTOMATON

STATE(num)
STATE_AWAIT(num,event)
STATE_AWAIT_N(num,event,delay)
STATE_GET_VALUE(num,event,n,result)
STATE_STAY(num,delay)
STATE_JOIN(num,thread)
STATE_JOIN_N(num,thread,delay)
STATE_SELECT(num,n,array,mask)
STATE_SELECT_N(num,n,array,mask,delay)

GOTO(num)
GOTO_NEXT
IMMEDIATE(num)
RETURN

SELF
SET_LOCAL(data)
LOCAL
ARGS
RETURN_CODE
```

5 Examples

5.1 Hello World!

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

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

void h (void *id)
{
   while (1) {
      fprintf (stderr,"Hello ");
      ft_thread_cooperate ();
   }
}
```

```
void w (void *id)
{
   while (1) {
      fprintf (stderr,"World!\n");
      ft_thread_cooperate ();
   }
}

int main (void)
{
  ft_scheduler_t sched = ft_scheduler_create ();

  ft_thread_create (sched,h,NULL,NULL);
  ft_thread_create (sched,w,NULL,NULL);

  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: messages `Hello` and `World!` are always printed in this order.
Here is the typical way to produce executable code:

```
gcc -D_REENTRANT -o test test.c -lfthread -lpthread
```

5.2 Blocking I/O

The following function `ft_thread_read` implements a non-blocking read I/O, using the standard blocking `read` function. The calling 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;
}
```

5.3 Producer/Consumer

One implements a producer/consumer example. There are 2 files, `in` and `out`, and a pool of threads that take data from `in`, process them, and then put results in `out`. A scheduler and an event are associated to each file; the event is generated to indicate that a new value is produded in the associated file.

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

Processing Values

In order to process a value `v`, the calling thread first unlinks from `in_sched`. After processing, it links to `out_sched` in order to put the result in `out`, and finally, it re-links to `in_sched`. The procedure for processing a value is the following (for simplicity, values are of type `int`):

```
void process_value (int v)
{
  ft_thread_unlink ();
  < process v >
  ft_thread_link (out_sched);
  put (v,&out);
  ft_thread_generate (new_output);
  ft_thread_unlink ();
  ft_thread_link (in_sched);
}
```

The function run by the processing threads is:

```
void process (void *args)
{
  while (1) {
    if (size(in) > 0) {
      process_value (get (&in));
    } else {
      ft_thread_await (new_input);
      if (size (in) == 0) ft_thread_cooperate ()
    }
  }
}
```

The event `new_input` is used to prevent polling when no value is available from `in`. However, to test it as present does not necessary implies that a value is available: it could have been consumed by another thread. Thus, a call to `ft_thread_cooperate` is needed 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 threads are added to the system: one for producing new values, and the other for consuming results. The main function is the following:

```
int main (void)
{
   int i;
   ft_thread_t thread_array [MAX_THREADS]

   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++) {
      thread_array[i] = ft_thread_create (in_sched,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;
}
```

5.4 Automata

Preemption by an Event

Here is the example of a one-state automaton, named `killer`, that preempts a thread when an event is present. The thread and the event are accessible with the macro `ARGS`.

```
DEFINE_AUTOMATON (killer)
{
   void **args = ARGS;
   ft_event_t event   = args[0]
   ft_thread_t thread = args[1]

   BEGIN_AUTOMATON

     STATE_AWAIT (0,event)
     {
        ft_scheduler_stop (thread);
     }

   END_AUTOMATON
}
```

A fair thread is created by:

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

The difference with a standard thread created by `ft_thread_create` is that no new pthread is actually created by `ft_automaton_create`. The automaton is simply run by the scheduler's pthread. Thus, no pthread context switch is needed and execution is more efficient.

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. The first state resumes the first thread to run (initially, both threads are suspended). The switching event is awaited in the second state, and then the threads are switched. The third state is similar to the second, 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.

6 Related Work

Thread Libraries in C

Several thread libraries exist for C. Among them, the **Pthreads** library [17] implements the POSIX standard for preemptive threads. LinuxThreads [4] is an implementation of **Pthreads** for Linux; it is based on kernel-level threads. Quick Threads [14] 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 multithreading.

Gnu Portable Threads [11] (GNU Pth) is a library of purely cooperative threads which has portability as main objective. The Next Generation POSIX Threading project [5] proposes to extend GNU Pth to the M:N model, with Linux SMP machines as target.

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 scheduling) wich makes Java multithreaded systems difficult to program and to port [13]. This difficulty is pointed out by the suppression from the recent versions of the language of the threads primitives to gain fine control over threads [3]. A first version of **FairThreads** has been proposed in the context of the Java language [7] in order to simplify concurrent programming in Java.

Threads in Functional Languages

Threads are used in several ML-based languages such as CML [18]. 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 multiprocessors machines.

FairThreads has been recently introduced in the Bigloo [1] implementation of Scheme. The present version only supports linked threads, and special constructs are introduced to deal with non-blocking I/Os.

Reactive Approach

FairThreads actually comes out from the so-called *reactive approach* [6]. In this approach, one basically has instants and broadcast events. As opposite to *synchronous languages* [12] such as Esterel, the absence of an event during one instant cannot be decided before the end of this very instant. As a consequence, the

reaction to the absence of one event is delayed to the next instant. This is a way to solve so-called "causality problems" which are raised with synchronous languages, and which are obstacles to modularity. The Reactive-C [8] 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 [10] and *filaments* [15] 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 [9] and [13]. 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* [16] in which states capture the notion of a running mode in the context of the synchronous language Lustre.

7 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 (which can be formally given). 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 in a simple way, 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 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 standard techniques such as "chunks" or "chores", sometimes used in thread-based programming.

Implementation

A first implementation of **FairThreads** is available (under the GNU General Public License (`http://www.gnu.org`) as a library called `fthread` [6] which must be used with the standard **Pthreads** library.

References

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[15] 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.

[16] 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.

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

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

8 Man Pages

8.1 ft_scheduler_create

SYNOPSIS

```
#include <fthread.h>

ft_scheduler_t ft_scheduler_create (void);

int ft_scheduler_start (ft_scheduler_t sched);
```

DESCRIPTION

ft_scheduler_create returns a new scheduler that will run the threads created in it, using **ft_thread_create**.

The new scheduler **sched** starts running when the function **ft_scheduler_start** is called.

RETURN VALUES

On success **ft_scheduler_create** returns the new scheduler; **NULL** is returned otherwise. On success the value 0 is returned by **ft_scheduler_start** and a non-zero error code is returned otherwise.

ERRORS

NULL The scheduler cannot be created.

BADCREATE The scheduler **sched** is not correctly created when started.

SEE ALSO

ft_thread_create (3).

8.2 ft_thread_create

SYNOPSIS

```
#include <fthread.h>

ft_thread_t ft_thread_create (ft_scheduler_t sched,
                              void (*runnable)(void*),
                              void (*cleanup)(void*),
                              void *args);
```

DESCRIPTION

ft_thread_create returns a new thread of control and links it to the scheduler **sched**. While linked in **sched**, the new thread will execute concurrently with the other threads linked in it.

Actual starting of the new thread is asynchronous with the creation.

The new thread applies the function **runnable** passing it **args** as first argument. The new thread terminates when it executes **ft_exit** or when it returns from the **runnable** function.
When stopped (by **ft_scheduler_stop**), the new thread applies the function **cleanup**, if it is not **NULL**, passing it **args** as first argument.
A pthread is created with each fair thread. This pthread is initially attached. It can be detached using **ft_pthread** and **pthread_detach**.

RETURN VALUES

On success, **ft_thread_create** returns a new thread; **NULL** is returned otherwise.

ERRORS

NULL The thread cannot be created, or the scheduler **sched** is not correctly created.

SEE ALSO

ft_exit (3), **ft_scheduler_create** (3), **ft_scheduler_stop** (3), **ft_pthread** (3).

8.3 ft_event_create

SYNOPSIS

```
#include <fthread.h>

ft_event_t ft_event_create (ft_scheduler_t sched);
```

DESCRIPTION

ft_event_create returns a new event which is created in the scheduler **sched**. The event can be generated by **ft_event_generate** or **ft_scheduler_broadcast**, and it can awaited by **ft_event_await**.

RETURN VALUES

On success, the new event is returned and set to absent; **NULL** is returned otherwise.

ERRORS

NULL The event cannot be created or the scheduler **sched** is not correctly created.

SEE ALSO

ft_event_generate (3), **ft_scheduler_broadcast** (3), **ft_event_await** (3).

8.4 ft_scheduler_stop

SYNOPSIS

```
#include <fthread.h>

int ft_scheduler_stop    (ft_thread_t th);

int ft_scheduler_suspend (ft_thread_t th);

int ft_scheduler_resume  (ft_thread_t th);
```

DESCRIPTION

ft_scheduler_stop asks the scheduler running the thread **th** to force termination of it. Nothing special happens if the thread is already terminated. Otherwise, at the begining of the next instant, **th** executes the function **cleanup** if it exists, or otherwise terminates immediately.
ft_scheduler_suspend asks the scheduler running the thread **th** to suspend execution of it. The suspension will become actual at the beginning of the next instant of the scheduler.
ft_scheduler_resume asks the scheduler running the thread **th** to resume execution of it. The resume will become actual at the beginning of the next instant of the scheduler. Suspension has higher priority than resume: if a thread is suspended and resumed during the same instant, then the thread will be suspended. A suspended thread which is stopped is first resumed.

RETURN VALUES

On success, the value 0 is returned. On error, a non-zero error code is returned.

ERRORS

BADCREATE The thread **th** is not correctly created.

BADLINK The thread **th** is unlinked.

BADMEM Not enough memory (the order cannot be stored by the scheduler).

SEE ALSO

ft_thread_create (3), **ft_scheduler_create** (3).

8.5 ft_scheduler_broadcast

SYNOPSIS

```
#include <fthread.h>

int ft_scheduler_broadcast       (ft_event_t evt);

int ft_scheduler_broadcast_value (ft_event_t evt,void *val);
```

DESCRIPTION

ft_scheduler_broadcast asks the scheduler of the event **evt** to broadcast it. The event will be generated during the next instant of the scheduler. The value **val** is associated to **evt** when the function **ft_scheduler_broadcast_value** is used.

RETURN VALUES

On success, the value 0 is returned. On error, a non-zero error code is returned.

ERRORS

BADCREATE The event **evt** is not correctly created.

BADMEM Not enough memory (the scheduler cannot store the broadcast order).

SEE ALSO

ft_event_create (3).

8.6 ft_thread_cooperate

SYNOPSIS

```
#include <fthread.h>

int ft_thread_cooperate   (void);

int ft_thread_cooperate_n (int n);
```

DESCRIPTION

ft_thread_cooperate makes the calling thread cooperate by returning the control to the scheduler in which it is running.

The call **ft_thread_cooperate_n (k)** is equivalent to **for (i=0;i<k;i++) ft_thread_cooperate ()**.

RETURN VALUES

On success, the value 0 is returned. On error, a non-zero error code is returned.

ERRORS

BADLINK The calling thread is unlinked.

8.7 ft_thread_join

SYNOPSIS

```
#include <fthread.h>

int ft_thread_join   (ft_thread_t th);

int ft_thread_join_n (ft_thread_t th,int n);
```

DESCRIPTION

ft_thread_join suspends the execution of the calling thread until the thread **th** terminates (either by reaching the end of the function it run, or by executing `ft_exit`) or is stopped (by `ft_scheduler_stop`). If **th** is already terminated, the call immediately terminates.

ft_thread_join_n (th,i) waits for at most **i** instants for termination of **th**.

RETURN VALUES

On success, the value 0 is returned. On error, a non-zero error code is returned.

ERRORS

BADCREATE The thread **th** is not correctly created.

BADLINK The calling thread is unlinked.

TIMEOUT The timeout is reached before the thread is joined.

SEE ALSO

ft_thread_create (3), **ft_exit** (3), **ft_scheduler_stop** (3).

8.8 ft_thread_generate

SYNOPSIS

```
#include <fthread.h>

int ft_thread_generate       (ft_event_t evt);

int ft_thread_generate_value (ft_event_t evt,void *val);

int ft_thread_await          (ft_event_t evt);

int ft_thread_await_n        (ft_event_t evt,int n);

int ft_thread_select         (int len,ft_event_t *array,int *mask);

int ft_thread_select_n       (int len,ft_event_t *array,int *mask,int timeout);
```

DESCRIPTION

ft_thread_generate generates the event **evt** for the current instant of the scheduler in which the calling thread is running. The event is thus present for this instant; it will be automatically reset to absent at the begining of the next instant.

The value **val** is associated to **evt** when **ft_thread_generate_value** is used.

ft_thread_await suspends the calling thread until **evt** becomes generated in the scheduler in which it is running. The waiting takes as many instants as the generation of **evt** takes.

ft_thread_await_n (evt,k) is similar to **ft_thread_await (evt)** except that the waiting of **evt** lasts at most **k** instants.

ft_thread_select suspends the calling thread until one element of **array** becomes generated in the scheduler in which the thread is running; **array** should be of length **k**. On resumption, **mask** which is an array of **k** integers, is set accordingly: **mask**[**i**] is 1 if **array**[**i**] was generated; **mask**[**i**] is 0, otherwise.

ft_thread_select_n (k,array,mask,p) is similar to **ft_thread_select (k,array,mask)** except that the waiting lasts at most **p** instants.

RETURN VALUES

On success the value 0 is returned and a non-zero error code is returned on error.

ERRORS

BADCREATE The exist an event (either **evt** or an element of **array**) which is not correctly created.

BADLINK Either the calling thread is unlinked, or the scheduler of the calling thread and the one of a considered event (**evt** or an element of **array**) are different.

BADMEM Not enough memory (can only occur with **ft_thread_generate_value**).

TIMEOUT The timeout is reached.

SEE ALSO

ft_event_create (3), **ft_thread_get_value** (3).

8.9 ft_thread_get_value

SYNOPSIS

```
#include <fthread.h>

int ft_thread_get_value (ft_event_t evt,int n,void **result);
```

DESCRIPTION

ft_thread_get_value returns the **n**th value associated during the current instant to the event **evt** through calls of **ft_event_generate_value** or **ft_scheduler_broadcast_value**. If such a value exists, **ft_thread_get_value** sets **result** with a reference to it and terminates immediately (that is, during the current instant). Otherwise, it terminates at the next instant (returning **NEXT**) and **result** is then set to **NULL**.

RETURN VALUES

On success, the value 0 is returned (during the current instant). Otherwise, a non-zero error code is returned.

ERRORS

BADCREATE The event **evt** is not correctly created.

BADLINK Either the calling thread is unlinked, or the scheduler of the calling thread and the one of **evt** are different.

NEXT Less than **n** values where actually associated to generations of **evt** during the previous instant.

SEE ALSO

ft_thread_generate_value (3), **ft_scheduler_broadcast_value** (3).

8.10 ft_thread_link

SYNOPSIS

```
#include <fthread.h>

int ft_thread_unlink (void);

int ft_thread_link   (ft_scheduler_t sched);
```

DESCRIPTION

ft_thread_unlink unlinks the calling thread from the scheduler which is running it. Execution of the thread suspends until the begining of the next instant of the scheduler. At that point, the thread turns into a standard thread, not linked to any scheduler, and it resumes execution autonomously.

Initialy, a fair thread is automatically linked to the scheduler in which it is created (by **ft_thread_create**).

ft_thread_link links the calling thread to the scheduler **sched**. The thread must be unlinked. Execution suspends until **sched** gives the control to the thread; then, the thread resumes execution, being scheduled by **sched**.

RETURN VALUES

On success, the value 0 is returned. On error, a non-zero error code is returned.

ERRORS

BADCREATE The scheduler **sched** is not correctly created.

BADLINK The calling thread is already linked while running **ft_thread_link**, or it is unlinked while running **ft_thread_unlink**.

BADMEM Not enough memory (the scheduler cannot store the link/unlink order).

SEE ALSO

ft_thread_create (3).

8.11 ft_thread_self

SYNOPSIS

```
#include <fthread.h>

ft_thread_t    ft_thread_self      (void);

ft_scheduler_t ft_thread_scheduler (void);
```

DESCRIPTION

ft_thread_self returns the calling thread. **ft_thread_scheduler** returns the scheduler of the calling thread.

ERRORS

The value **NULL** is returned by **ft_thread_self** when the calling thread is not correctly created, or by **ft_thread_scheduler** when the calling thread is not correctly created or is unlinked.

8.12 ft_thread_mutex_lock

SYNOPSIS

```
#include <fthread.h>

int ft_thread_mutex_lock   (pthread_mutex_t *mutex);

int ft_thread_mutex_unlock (pthread_mutex_t *mutex);
```

DESCRIPTION

For unlinked threads, **ft_thread_mutex_lock** is like **pthread_mutex_lock** and **ft_thread_mutex_unlock** is like **pthread_mutex_unlock**.

For linked threads, **ft_thread_mutex_lock** suspends the calling thread until **mutex** can be locked. Thus, while **mutex** is unavailable, other threads in the scheduler can continue to run (this would not be the case if **pthread_mutex_lock** where used instead of **ft_thread_mutex_lock**). All locks owned by a thread are automatically released when it terminates or when it is stopped.

RETURN VALUES

On success **ft_thread_mutex_lock** and **ft_thread_mutex_unlock** both return the value 0. On error, a non-zero error code is returned.

ERRORS

Errors returned are the ones returned by **pthread_mutex_lock** and **pthread_mutex_unlock**.

SEE ALSO

ft_thread_link (3), **ft_thread_unlink** (3).

8.13 ft_exit

SYNOPSIS

```
#include <fthread.h>

void ft_exit (void);
```

DESCRIPTION

ft_exit forces the calling thread to terminate.

RETURN VALUES

The function **ft_exit** never returns.

8.14 ft_pthread

SYNOPSIS

```
#include <fthread.h>

pthread_t ft_pthread (ft_thread_t thread);
```

DESCRIPTION

The function **ft_pthread** returns the pthread on which the fair thread **thread** is built.

8.15 ft_automaton_create

SYNOPSIS

```
#include <fthread.h>

ft_thread_t ft_automaton_create (ft_scheduler_t sched,
                                 void (*automaton)(ft_thread_t),
                                 void (*cleanup)(void*),
                                 void *args);

AUTOMATON(name)
DEFINE_AUTOMATON(name)
BEGIN_AUTOMATON
END_AUTOMATON

STATE(num)
STATE_AWAIT(num,event)
STATE_AWAIT_N(num,event,delay)
STATE_GET_VALUE(num,event,n,result)
STATE_STAY(num,delay)
STATE_JOIN(num,thread)
STATE_JOIN_N(num,thread,delay)
STATE_SELECT(num,n,array,mask)
STATE_SELECT_N(num,n,array,mask,delay)

GOTO(num)
GOTO_NEXT
IMMEDIATE(num)
RETURN

SELF
SET_LOCAL(data)
LOCAL
ARGS
RETURN_CODE
```

DESCRIPTION

Automata Creation

ft_automaton_create is very similar to **ft_thread_create** except that a new automaton is returned. The automaton does not have its own pthread to execute, but it is run by the one of the scheduler.

The automaton applies the function **automaton**. Argument **args** can be accessed in the automaton definition with the macro **ARGS**.

Macros

- `AUTOMATON(name)` declares the automaton `name`.
- `DEFINE_AUTOMATON(name)` starts definion of the automaton `name`.
- `BEGIN_AUTOMATON` starts the state list.
- `END_AUTOMATON` ends the state list.
- `STATE(num)` starts state `num` description. States must be numbered consecutively, starting from 0. State 0 is the initial state.
- `STATE_AWAIT(num,event)` awaits `event`. It is the counterpart of `ft_thread_await` for automata. Execution stays in this state until `event` is generated.
- `STATE_AWAIT_N(num,event,n)` awaits `event` during at most **n** instant. It is the counterpart of `ft_thread_await_n`.
- `STATE_GET_VALUE(num,event,n,result)` is used to get the `n`th value generated with `event`. It is the counterpart of `ft_thread_get_value`.
- `STATE_JOIN(num,thread)` is used to join `thread`. It is the counterpart of `ft_thread_join`.
- `STATE_JOIN_N(num,thread,n)` is an attempt to join `thread` during at most **n** instants. It is the counterpart of `ft_thread_join_n`.
- `STATE_STAY(num,n)` let the automaton stay for `n` instants in the state.
- `STATE_SELECT(num,k,array,mask)` awaits elements of `array` which is an array of events of length `k`. It is the counterpart of `ft_thread_select` for automata. Execution stays in this state until at least one element of `array` is generated; the presence of events is recorded in `mask` which is an array of integers of length `k`.
- `STATE_SELECT_N(num,k,array,mask,n)` awaits elements of `array` during at most **n** instant. It is the counterpart of `ft_thread_select_n`.
- `GOTO(num)` blocks execution for the current instant and sets the state to be executed at the next instant to be state `num`.
- `GOTO_NEXT` blocks execution for the current instant and sets the state for the next instant to be the next state.
- `IMMEDIATE(num)` forces execution to jump to state `n` which is immediately (that is, during the same instant) executed.
- `RETURN` forces immediate termination of the automaton.
- `SELF` is the automaton. It is of type **ft_thread_t**.
- `LOCAL` is the local data of the automaton. The local data is of type `void*`.
- `SET_LOCAL(data)` sets the local data of the automaton.
- `ARGS` is the argument that is passed at creation to the automaton. It is of type `void*`.
- `RETURN_CODE` is the error code set by macros run by the automaton. As usual, 0 means success.

Note that there is no counterpart of **ft_thread_link** and **ft_thread_unlink** for automata, as an automaton always remains linked to the scheduler in which it was created.

RETURN VALUES

On success, **ft_automaton_create** returns a new thread; **NULL** is returned otherwise.

When an error is encountered during execution of a macro, **RETURN_CODE** is set accordingly, with one of the error values **BADMEM**, **TIMEOUT**, **NEXT**, **BADLINK**, or **BADCREATE**.

ERRORS

NULL The automaton cannot be created, or the scheduler **sched** is not correctly created.

EXAMPLE

The following automaton switches control between two threads, according to the presence of an event.

```
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
}
```

SEE ALSO

ft_thread_create (3), **ft_thread_await** (3), **ft_thread_get_value** (3), **ft_thread_join** (3).