Free Jazz: An User-Level Real-Time Threads Package Designed for Flexibility

Thorsten Kramp
kramp@informatik.uni-kl.de

Report 9/98

Sonderforschungsbereich 501

System Software Group
Computer Science Department
University of Kaiserslautern
Postfach 3049
67653 Kaiserslautern
Germany

FREE JAZZ: An User-Level Real-Time Threads Package Designed For Flexibility

Thorsten Kramp*

University of Kaiserslautern, Department of Computer Science
P. O. Box 3049, 67653 Kaiserslautern, Germany

ABSTRACT. Up to now designers of multi-threaded run-time systems traditionally disdain readily available user-level threads packages due to their inherent lack of flexibility, resultant from prevalently adhering to the black-box approach. FREE JAZZ, in contrast, is a user-level threads package developed in the spirit of the open implementation design methodology, providing a well-devised meta interface particularly with respect to soft real-time scheduling, inter-thread communication, and synchronisation. Any scheduling code has been systematically factored out to allow for easy customisation and adaption even at run time. The general viability of this approach is demonstrated by the results of a number of performance measurements carried out.

1 ON GRIEF AND REASON

FREE JAZZ is a user-level real-time threads package.[1] "Uh oh," I hear you say with a wry smile, "yet another user-level threads package. So what?" And yes, you are right — partially. However, no single user-level threads package has gained wide popularity among designers of multi-threaded run-time systems or middleware components at the time of writing. In fact, only a few, if any, seem to use a readily available user-level threads package at all, yet develop their own. Performance reasons and portability problems aside, the main obstacle still remaining usually is an inherent lack of flexibility, ultimately caused by adhering to the black-box principle. To be more specific, each developer of a threads package must face and find a solution for a number of critical design decisions, so-called *design dilemmas* or just *dilemmas* for short, namely in the areas of thread states, thread lists and scheduling, context switching modes, stack management, as well as timing and profiling. The developer of a threads package, however, cannot anticipate the "best" way to solve these dilemmas,

*kramp@acm.org

[1] The name FREE JAZZ was chosen because threads are similar to the members of a free jazz band, at least to a certain extend. While each musician often seems to follow a highly individual theme (and more often really does), everyone perfectly contributes to the common tune in real time.

whereas the designer of the multi-threaded run-time system is assumed to be targeting a specific application domain and therefore hopefully can. Trying to be clairvoyant only will force "coding between the lines," that is, playing hider-and-seeker with the threads package to get it out of the way.

The argument itself is not new and merely a variant of the famous end-to-end argument discussed by Saltzer et al. in the context of communication protocols [16]. Knowing that there are no silver bullets for dilemmas, research in the realm of object-oriented system-level software came up with the *open implementation* design methodology [9, 10]. Basically, the functional interface of a black box is enriched by an interface that describes the behaviour of its components, the so-called *meta interface*. This way critical design decisions are systematically exposed, sometimes with a fall-back policy that might be good enough for those who do not care, while the sensitive user is free to jump in and provide a better alternative whenever he or she is not satisfied with the default. Such an approach is particularly promising for user-level threads packages that might serve as starting points for building multi-threaded run-time systems. Yet, while configurable user-level threads packages such as PRESTOhave been around since 1988 at least [3, 4], the notion of an open implementation has been only recently exploited *systematically* for threads by Haines, whose OPENTHREADS were intended as a proof of concept [7]. Neither OPENTHREADS nor any other configurable user-level threads package we know about, however, were developed with real-time environments in mind.

At the heart of each open implementation is agreeing on which design decisions actually are critical and which are not; simply turning everything into an option is not an option. In fact, the meta interface should be kept as small as possible, yet cover all dilemmas. "If a user-level threads package is not useful to a system-level programmer, lack of control over scheduling is commonly at the root of the cause." [7] Black-box implementations allow the user to choose from a number of pre-defined schedulers at best, which is necessarily rather restrictive in view of the vast amount of quite different policies developed during the last three decades. In addition, all sorts of internal queueing and any synchronisation primitives may be affected by a scheduling policy, specifically in the context of real-time systems, where priority inversion usually becomes an issue and fairness is sacrificed light-heartedly. Rather than implementing one or even many schedulers at the core of a threads package, it therefore seems far more promising to transform thread transitions into events and let the user decide what to do next. All thread lists and, in the case of inter-thread communication, all message queues must be made explicit in such a way that their policies and internal structure may be overridden on demand. As minor aspects, stack management and context switching modes may also be opened, mainly with stack allocation and saving/restoring of registers in mind, respectively.

In the following sections we present FREE JAZZ, a user-level threads package to be used with C/C++ and developed in the spirit of the open implementation design methodology. Specifically and in contrast to OPENTHREADS, the particularities of soft real-time requirements are explicitely considered. While the underlying programming model and most of its internal structure are inherited from an undocumented non-real-time black-box threads package originally developed by Peter Buhler here at the Distributed Systems Group, scheduling has been systematically untangled from the inner core and the ground prepared for soft real-time threading.

The rest of this report is structured as follows. At first, an overview of related work is given in Section 2. Then, Section 3 pinpoints the pivotal concepts of the underlying programming model, before Section 4 gives an impression of what the programmer's interface actually looks like. Section 5, finally, summarises the results of some performance measurements carried out, with conclusions following in Section 6.

2 RELATED WORK

One of the first extensible user-level threads packages is PRESTO, written in C++ and designed with the philosophy that users should be able to replace components such as the scheduler [3, 4]. PRESTO targets multiprocessor machines such as the Sequent Balance 21000 and particularly allows differential extensions (due to the use of C++) as well as lateral extensions, that is, to change the behaviour dynamically at run time. However, a re-implementation in C and Assembler called FASTTHREADS, in which all policies were hard-coded, is about an order of magnitude faster [2].

QUICKTHREADS, in contrast, refrains from being a complete threads package on its own, yet provids a threads toolkit for uniprocessor architectures with a portable interface to machine-dependent code that performs thread initialisation and context switching [8]. That is, it plainly avoids all dilemmas right from the beginning by not including functionality such as scheduling and synchronisation primitives. This way QUICKTHREADS attempts to achieve most of the flexibility of PRESTO while retaining a performance similar to FASTTHREADS.

OPENTHREADS, finally, is similar to FREE JAZZ in that it systematically exploits the notion of an open implementation and reports thread state transitions to a user-defined scheduler [7]. Neither OPENTHREADS nor any other of the aforementioned threads packages, however, has been developed specifically with real-time environments in mind, while real-time user-level threads packages, in contrast, were black-box solutions up to now.

RT THREADS [5], for instance, is such a real-time user-level threads package, which even provides mechanisms for inter-thread communication between different address spaces, possibly on different machines. Its synchronisation primitives, however, do not care about priority inversion and the scheduling policy — a priority-driven approach featuring EDF (earliest deadline first) within each priority — is buried. While the RT THREADS package is operating system independent, the user-level real-time threads of the ARTS operating system [15] are so-called first-class threads. That is, the process scheduler of the kernel and the threads schedulers of all processes interact via scheduler activations as proposed by Anderson et al. [1] and Marsh et al. [14], for instance.

Yet, the need for more flexibility specifically with respect to real-time architectures has also been identified by Ann Lo et al. [13] as well as Goyal et al. [6]. Their work on real-time kernels, extending the idea of scheduler activations, led to a hierarchy of kernel-level and user-level schedulers. To this end, our open user-level threads packages naturally adds to these architectures.

3 THE RULES OF THE GAME

Prior to getting real with function calls and parameter lists, this section is meant to introduce the basic abstractions and the underlying design philosophy of FREE JAZZ.

Basic Abstractions & Inter-thread Communication.

Programming in FREE JAZZ is foremost a game on the grounds of processing in response to messages, that is, reactivity is at the core of what this is all about. A thread only jumps into action after it has received a message and equally may send messages to other threads, effectively requesting some processing on its behalf. The whole process is initially set into motion by a *root thread* created during system startup, which usually spawns a first set of user threads, sends some go-ahead messages to a subset of these, and finally awaits termination. For all practical matters, messages can be assumed to originate from another thread; any hardware I/O including timer signals or user interaction already is or may be transformed by dedicated threads into appropriate messages. These messages will re-animate the web of interacting threads if there was temporarily nothing left to do and the system went idle.

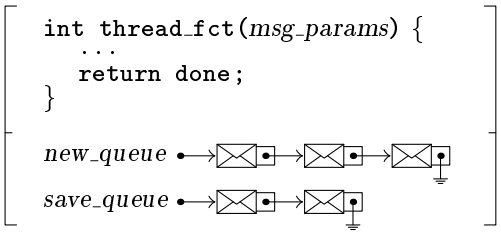

Fig. 1: A thread as seen from a user's point of view.

A thread merely consists of a C function with a fixed parameter list that describes the next message to process and is associated with a certain priority. Further, due to the intrinsic importance of inter-thread communication based on messages, it seems natural to introduce a queue with each thread to hold incoming messages, the so-called *new queue*. With each invocation of a thread the associated function is called, which performs a series of steps in response to the first message found in the new queue. These steps possibly include interaction with other threads and reading further messages from the new queue, before finally the thread's function is left with a return value. The return value indicates whether the thread awaits further messages and should remain in the system, or has terminated eventually and should be disposed of. If the thread is kept around, its function will be called again as soon as no thread of higher priority is requesting processing time, provided not all messages have been processed during the thread's last invocation or new ones have arrived in the meantime. Additionally, each thread is equipped with a *save queue* to temporarily store messages that cannot be processed immediately and require further preparation, due to an outstanding response from another thread, for example.

At each instant, a thread is either preemptable and may be interrupted in favour of a higher-priority thread, or non-preemptable, processing undisturbed until it blocks, allows a reassignment of the processor at its free will, or returns from its associated function. Preemption can be dynamically disabled and re-enabled as needed. Note that the property of being preemptable is assigned on a per-thread basis, not globally.

Messages may either be sent synchronously if there remains nothing to do for a thread until an answer is given, or asynchronously whenever the reply is not necessarily needed right now to continue or no response is required at all. Synchronous messages are termed as *calls* (bearing resemblance to telephone calls), whereas asynchronous messages are *sent* more similar to a letter. If not explicitely stated otherwise, the term *send* is used to indicate that both calls and sends are valid within a particular context. Moreover, because FREE JAZZ was designed with soft real time in mind, a message may be accompanied by a user-defined *constraint* field to describe scheduling properties such as deadlines.

Up to this point all threads are equal, but some threads are more equal than others. What sets a thread apart from an ordinary C function is the notion of a context, comprising the registers of the CPU including the process counter, the stack pointer, and some additional fields describing the thread's state or priority, for example. In FREE JAZZ, however, some threads do not own a context of their own yet have to "borrow" one from a thread that sent a message recently. To distinguish both species of a thread we call those with a context of their own *processes* (not to be confused with heavy-weight UNIX processes) and those of the context-sharing party *handlers*. The term *thread* will only be used from now on when process or handler could have been used interchangeably, that is, when it does not matter to whom the context actually belongs.

Anyway, the policy of context sharing is as follows. If a message is sent to a handler that is currently busy with another message, the behaviour is the same as if the message would have been sent to another process — the sending thread is blocked on calls and continues on sends. Otherwise the handler is invoked immediately and processes the message, borrowing the context of the sending thread. As soon as the handler returns, the borrowed context is given back provided the handler was not preempted by some higher-priority thread and has received further messages in the meantime. If its new queue is not empty, however, any pending messages are also processed within the borrowed context. Although a handler is free to save an incoming message for later processing, possibly blocking a calling process, a handler on its part must not block due to some technical subtleties. Specifically, a handler is not allowed to call another thread. Invoking a handler, however, is significantly faster than invoking a process (cf. Table 2 in Section 5).

Of Monitors and Semaphores.

FREE JAZZ does not multiplex its user-level threads onto a small number of kernel-level threads (which would be particularly reasonable on an SMP) and, thus, only provides multi-*programming*, not multi-*processing* at the time of writing. The pitfalls of concurrently accessed shared data, however, remain nevertheless. Threads still must synchronise prior to accessing shared data; otherwise bewildering inconsistencies inevitably will cause complete havoc in the end.

A simple, yet perfectly reasonable approach to ease the pain of synchronisation for user-level threads is going non-preemptive. Generally, the programmer may be assumed to be able to organise the threads of an application or middleware component in such a way that these are working cooperatively. The scheduler is only called in whenever a thread blocks, explicitely grants a reassignment of the CPU, or is done with a particular message and returns from its function. This way a thread's function cushily becomes the unit of mutual exclusion with respect to shared data.

Sometimes, however, preemption is handy at least. The programmer is relieved from manually taking care of higher-priority threads during long-running calculations and real-time system designers, even those in the business of soft deadlines only, usually try hard to keep the periods of non-preemption at a minimum to tame priority inversion, for instance. Consequently, fine-grained synchronisation primitives are reasonable by all means.

Interesting enough, handlers already can be used for such a fine-grained synchronisation.[2] Although not explicitely mentioned before, the properties of handlers are related to those of a monitor, particularly if invoked synchronously by a call. First of all, a handler is non-reentrant by definition and therefore may be used to access shared data mutually exclusively. Instead of a set of different functions as found in a monitor, different messages may be sent to a handler, which is logically equivalent. But what about condition variables? Of course, there is no SIGNAL or WAIT; similar effects, however, can be achieved by means of the save queue. It is easy to see that saving the message of a calling thread is equivalent to the WAIT operation of a condition variable. Reading and processing a saved message corresponds to a SIGNAL operation where the signalled thread is given precedence over the signalling one. It is therefore straightforward to use handlers as synchronisation mechanism in the spirit of monitors, even though the code will look quite different.

The same is obviously true for semaphores, which may be emulated by a handler with a single local variable used as semaphore counter. The P operation is substituted for a PROCURE message that simply decrements the semaphore counter if its value is greater than zero, otherwise the message is saved, effectively blocking the calling process. V operations conversely are substituted for VACATE messages that cause a reply to a saved message or just an increment of the semaphore counter if no thread is currently hold off. Thinking about priority inheritance [17] as a way to bound priority inversion, however, requires additional care. While calling a handler to perform some processing might result in priority inversion, too, this is automagically detected by FREE JAZZ and appropriately signalled to the scheduler. With handlers as semaphores, however, there is no association of a critical section and its protecting semaphore; appropriate signals must be generated by the handler manually. Though FREE JAZZ was developed with exactly this kind of user extension in mind, semaphores are popular enough to provide them right out of the box just for the sake of convenience — it would have been no harder for the designer of a multi-threaded run-time system to add these semaphores by herself.

Memory Management.

Since FREE JAZZ is targeting (soft) real-time environments, its memory management has been designed to be predictable with respect to utilisation as well as worst-case execution times.

Utilisation predictability means that there are no hidden memory allocations which cause an application to become surprised by an out-of-memory error — provided the application-specific demands do not exceed the available memory, of course. Each memory allocation is only done in response to a user request.

[2] The following arguments mostly hold for processes, too; however, it seems more natural to use a handler for synchronisation purposes since it will not produce a context switch as long as no serialisation is needed and incidentally inherits the priority of the invoking thread.

Specifically, FREE JAZZ does not generate messages plainly out of thin air, not even to signal hardware interrupts. The general policy, in contrast, is for an application to provide the message it would like to receive in response to a hardware interrupt, which is simply "bounced" as the interrupt occurs. Consider a timer interrupt, for instance. The application sends a message to the FREE JAZZ timer thread, denoting an interval after that it would like to be notified as the message value. When the specified period of time has expired, the timer thread simply replies the given message without any memory allocation on its own. Since this approach is used throughout, memory utilisation is highly predictable and allows the programmer to keep track of the available memory quite easily.

Predictability with respect to worst-case execution times, in contrast, is concerned with the allocation and deallocation operations. While deallocation including any necessary coalescences of adjacent free blocks always can be done in constant time, that is $O(1)$ steps, allocation can be worst-case bounded to $O(\log n)$ steps at best [11]. FREE JAZZ, however, obeys a user-defined bound k on the number of elements in the free list that are inspected as part of each allocation operation. If no sufficiently large memory block can be found in the first k elements of the free list, memory allocation simply fails. This way a worst-case bound on memory allocation and deallocation can be derived, particularly since FREE JAZZ optionally disables paging for all its memory via the `mlock()` system call.

Going Meta.

By now an alert reader will probably scratch his head and ask himself what scheduling policy eventually is used in FREE JAZZ. Well, this one is easy. *There is none!*[3] Yes, we already mentioned priorities, priority inheritance, and message constraints but recall what was stated in Section 1 as the primary reason for introducing a meta interface in the beginning: "If a user-level threads package is not useful to a system-level programmer, lack of control over scheduling is commonly at the root of the cause." [7]

Consequently, we refrained from adding a covert scheduler, yet systematically transformed thread state transitions into appropriate messages to be processed by user-implemented non-preemptive scheduling handlers. These handlers are responsible for ordering processing requests, assigning appropriate priorities, and for signalling FREE JAZZ whenever a context switch should be performed. To this end the user has complete control over the internal organisation of the ready queue, may it be a doubly linked list, a heap, or something else the world has not seen before. What a priority looks like and the way it should be interpreted are open issues, too, only to be resolved by the user.
Foremost the transitions reported are when a thread becomes ready (marked as *1* in Fig. 2), when a reassignment of the CPU occurs due to preemption or at the will of the active thread (*2*), or when the active thread runs out of messages to process and becomes idle (*3*). In contrast to OPENTHREADS there are no messages when a thread has been newly created (remember that a thread only becomes ready when it receives a message) or is destroyed, nor

[3] Well, not really, of course. A simple rate-monotonic scheduler with a quite rudimentary understanding of what priority inheritance is about has been implemented and is included as fall-back policy.

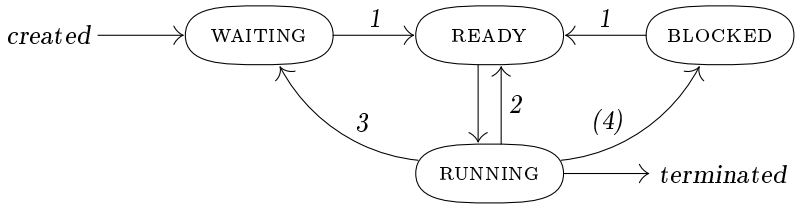

Fig. 2: Thread state transitions.

does FREE JAZZ report system idle times to the scheduler. The user, however, might easily allocate some kind of idle thread to gather load statistics or might add create/destroy messages to be handled by a suitable scheduling component. Furthermore, the scheduler may request always to be called whenever a thread is blocked at a semaphore or while calling a handler, or only for threads that should endure bounded priority inversion at worst (*4*). The former is reasonable if the execution of the first thread in the ready queue might be cancelled due to a missed time constraint, for instance, while the latter allows the scheduler to adjust the priority of a lower-priority thread, effectively bounding the blocking time of higher-priority threads. As soon as the blockade has been raised, a message requests the scheduler to reset the inherited priority to its original value and to reassign the CPU in favour of the waiting higher-priority thread. The scheduler also might take care of nested or multiple priority inheritance on the grounds of internally assigned unique identifiers that link corresponding inherit/reset requests.

Beside these pre-defined messages, the user is free to define additional ones as part of an admission test, for example. After all, a scheduling policy usually affects not only a thread's priority but must also be pervasively reflected by its new queue at least. The default policy is FIFO because the aforementioned constraint field of a message is completely user-defined. It is therefore up to the user to enhance the doubly linked FIFO queue with an appropriate sort function when needed or to provide a complete replacement. Specifically, the sort function may deny a message, which cannot be processed with respect to the constraint given. And finally, although this is not an issue of priorities, a thread might use its own save queues as replacement of or in addition to the standard FIFO save queue.

This only leaves the waiting queues of semaphores open for discussion, which, of course, should be priority-driven, too. However, since FREE JAZZ again does not prescribe the structure of a thread's priority similar to the constraint field of a message, it cannot anticipate a reasonable sorting policy on its own. Queueing at semaphores is therefore done in FIFO order by default but may be overridden by any user-implemented policy.

4 FACING THE INTERFACE

While a neat concept surely is more than half the battle, an equally well-devised programmer's interface is needed to keep the promises. What follows should provide enough information to round out the conceptually-driven presentation of the previous section with some practical insight.

In contrast to the approach chosen by OpenThreads, the functional interface and the meta interface are not strictly separated in Free Jazz. It seemed to be more convenient and natural to mix up both kinds of parameters to a certain extend rather than to introduce dedicated functions for meta parameters artifically.

Thread Creation.

From a programmer's point of view, a thread is just a C function that agrees with the following prototype:[4]

```
int (*Code)(void *environment, Object *from,
            long id, long value, long constraint);
```

The `environment` parameter points to a user-defined struct of variables local to that thread, whereas the remaining four parameters describe the message that caused the thread to become active.[5] The return value of a thread's function must be set to `STOP` if the thread has terminated and should be disposed of, while any value equal to or greater than 0 ensures that the thread is kept around.

All messages in Free Jazz adhere to a fixed format of four fields. The `from` parameter always identifies the sender of the message and is set automatically as part of any send operation, while structure and content of `id`, `value`, and `constraint` are entirely user-defined.[6] Both `id` and `value` are evaluated by the recipient of a message only and, thus, their meaning need not be related to their names in any way. The `constraint` parameter, on the other hand, is intended to prescribe any restrictions or special boundary conditions associated with a message (e. g., a deadline). The recipient aside, it may (and probably will) be evaluated by the sort function of the recipient's new queue and by some scheduling components to adjust the priority of the recipient accordingly. The scheduler is not called in if the `constraint` parameter equals `No_constraint`, a user-defined global variable that allows Free Jazz to distinguish whether a constraint potentially affects the recipient's priority or not.[7]

Calling `new_process()` or `new_handler()` transforms any suitable C function into a thread. Since handlers must "borrow" their contexts from the invoking thread, the parameters needed to create a new handler are merely a subset of those needed to create a process. Hence, common parameters are discussed first. While `environment` should point to a struct containing a thread's local variables as described before,[8] the `queue` parameter may be used to override the implementation of the thread's new queue whenever the default doubly linked FIFO queue is inappropriate. Instead of a complete reimplementation it is also possible to stick to the doubly linked list and only to replace the FIFO

[4] The inevitable prefix `FJ_` of all FreeJazz function and struct names has been removed for brevity.

[5] The `environment` comes first to provide for C++ member functions instead of plain C functions, in which case it is equal to the `this` pointer and need not be declared explicitely.

[6] Albeit their declaration as `long`s it should be noted that any pointer may be passed along instead with the appropriate casts at both ends. Thus, all three parameters effectively can carry any amount of data.

[7] Recall that a constraint's structure and content are not restricted in any way and, thus, `No_constraint` is not a pre-defined constant either.

[8] If a C++ member function is used as thread function, the `environment` parameter must be a pointer to an object of the corresponding class.

policy on demand by overriding the `put()` and `get()` function pointers of the `Message_Queue` struct.

```
Object* new_process(Code code, Message_Queue *queue, char *stack,
   long stack_size, Memory *memory, long prio, int tame_prio_inv,
   void *environment);
Object* new_handler(Code code, Message_Queue *queue,
   int tame_prio_inv, void *environment);
```

The meaning of the `tame_prio_inv` flag differs for handlers and processes. Recall that a handler is not allowed to block and, thus, cannot sustain priority inversion at all. However, since handlers are left only as soon as their new queue has been emptied completely, these pending messages may cause a virtually unbounded priority inversion when the handler is invoked from within a critical section or another handler. A higher-priority thread waiting at an outer handler or critical section generally cannot estimate the number of messages waiting at the inner handler and therefore has no idea how long its blocking will last even in the worst case. Consequently, a handler with the `tame_prio_inv` flag set automatically becomes non-preemptive when called from within another handler or a critical section.

For processes, on the other hand, the `tame_prio_inv` flag indicates whether or not blocking at handlers or semaphores should be reported to the scheduler to allow for some sort of priority inheritance. Furthermore, if such a thread invokes an inside handler in the sense as described in the former paragraph, the handler again is invoked non-preemptively, because a process that does not want to suffer from unbounded priority inversion should not cause it either. With the remaining parameters `stack` and `stack_size` a user may provide some pre-allocated stack space to be used by that process or may request Free Jazz to allocate the stack from the free space designated, while `prio` denotes the initial priority of the new process.[9]

Inter-thread Communication.

As mentioned before, messages may either be sent synchronously or asynchronously. In general, both the aynchronous `send()` and the synchronous `call()` must allocate a new message if the recipient is either a handler currently in use or a process. Otherwise message allocation is avoided for the time being due to some lazy allocation scheme that sets aside one pre-allocated message buffer per handler. The asynchronous `reply()` simple reuses the current buffer.

```
int send(Object *to, long id, long value, long constraint);
int call(Object *to, long *id, long *value, long *constraint);
int reply(long id, long value, long constraint);
```

On the recipient's side we must consider reading the next message from the new queue, saving a message that cannot be processed immediately, and fetching these messages from a save queue later on again.

```
int read(Object **from, long *id, long *value, long *constraint, int mode);
int save(long id, long value, long constraint, Message_Queue *queue);
int read_saved(long selection, Object **from, long *id,
   long *value, long *constraint, Message_Queue *queue);
```

[9] Albeit its declaration as `long` it should be noted that any pointer may be stored within all priority fields (similar to the message parameters).

The meaning of these functions and most of their parameters should be self-evident. The `mode` parameter of `read()` distinguishes between blocking read operations that only return when there is actually a message to read, and non-blocking ones reporting an error code if the new queue is currently empty.[10] To retrieve a particular message from a save queue the `selection` parameter of `read_saved()` is interpreted as a bit field that specifies whether and which of the other parameters contain values that should match the message to read. Further on, any number of save queues may be used in parallel, with a doubly linked list sorted by `value` as default queue whenever `queue` is a null pointer.

Synchronisation.

Since there are no dedicated synchronisation primitives in FREE JAZZ apart from semaphores, the programmer's interface in the area of synchronisation is quite easy to survey (cf. the discussion in Section 3).

```
Semaphore *new_semaphore(long count, Queue *queue);
int free_semaphore(Semaphore *semaphore);
```

A semaphore's `count` parameter prescribes the number of threads that are allowed to stay inside the critical section protected by this semaphore simultaneously. Whenever FIFO queueing at a semaphore is inappropriate, the user is free to provide an alternative implementation via the `queue` parameter.

```
int procure(Semaphore *semaphore, int mode);
void vacate(Semaphore *semaphore);
```

`procure()` may be called either in blocking and non-blocking mode similar to `read()`, specifically since handlers are not allowed to block. A non-blocking procure, however, not necessarily ensures access to the critical section but simply may return an error code rather than blocking the calling thread, leaving it up to the user to proceed accordingly.

Scheduling.

A thread's facilities with respect to scheduling are basically limited to yielding the processor or toggling its preemption status.

```
void schedule(void);
int preemption(int switch_flag);
```

While calling `preemption()` only affects the preemption status of the running thread, going non-preemptive silences all other threads until the running thread is done with the current message, blocked during a call or at a semaphore, or releases the CPU at its free will. It should be noted, however, that when turning preemption temporarily off, it is, as a rule, imprudent to simply turn it on again subsequently. It might have been already turned off in the beginning for good reason and, thus, the old preemption status returned by `preemption()` should be restored instead.

And now for something completely different: The scheduling meta interface. As mentioned before, scheduling components in FREE JAZZ, which must be non-preemptive handlers actually, only react to scheduling messages, pre-defined as

[10] Handlers are not allowed to block and, thus, are restricted to non-blocking reads.

well as user-defined ones. The entire collection of pre-defined scheduling messages as derived from Fig. 2 is summarised in Table 1 for easy reference and will be explained step-by-step subsequently. Basically, a scheduling handler never ever performs a context switch by itself, yet directs FREE JAZZ to activate the first thread in the ready queue by setting the global flag `Passivate_running`.

Each scheduling handler may be registered for one or more message types. `register_scheduler()` returns the handler previously registered for this message type to allow for some kind of internal stacking, that is, the new scheduling handler may forward corresponding messages accordingly.

```
Object *register_scheduler(Object *object, long msg_type);
void scheduler_install();
```

During system startup, with no scheduling handler registered at all, FREE JAZZ calls the user-defined function `scheduler_install()`, within the user must nominate (not necessarily different) scheduling handlers to process READY and SCHEDULE messages. If priority inversion should be tackled, suitable scheduling handlers for all of the remaining signals (i. e., CALL, PROCURE, and NICE) must be registered, too. Besides the user is free to define new message types, register appopriate scheduling handlers, and even to modify the scheduling configuration at run time if needed.

The `Thread` struct as defined in FREE JAZZ already comprises a number of relevant fields with respect to scheduling. While some of these should be quite obvious in their meaning, others probably deserve a few words of explanation. Since FREE JAZZ makes no assumptions about the size of the thread struct, additional fields may be easily appended whenever the default set is found lacking.

```
typedef struct Thread_Struct {
  struct Thread_Struct *next, *prev;  /* doubly linked         */
  long prio, save_prio                /* priority fields       */
  long restore_prio, restore_reason;  /* priority inheritance  */
  Message_Queue *new_queue;           /* thread's new queue    */
  Semaphore *semaphore;               /* blocked at semaphore? */
  ...
} Thread;
```

Two fields have been set aside to easily collect threads of similar state or importance in doubly linked lists, which is the default implementation of the ready queue by the way.[11] The scheduler (i. e., more precisely, the team of cooperating scheduling handlers), however, may exert any number of user-implemented ready queues in parallel, activating the one for FREE JAZZ to use on demand. Using the term *queue* in its broadest sense, a call to `ready_queue()` dynamically activates a new ready queue and returns the replaced one.

```
Queue *ready_queue(Queue *new_queue);
```

The internal structure may differ from queue to queue as well, provided the first two fields are arranged to point to the queue's `put()` and `get()` functions.

The priority scheme of FREE JAZZ is specifically designed with soft real-time threading in mind. While a single priority field would be usually sufficient for non-real-time threading, life is not that easy as soon as priority inversion

[11] FREE JAZZ automatically keeps track of blocked threads, that is, there is usually no need to maintain a dedicated list for this purpose.

id	*value*	*contraint*
READY	thread to insert	constraint
SCHEDULE	NOP, PRIO SAVE/RESTORE, NEXT	constraint
CALL	stymie descriptor	constraint
PROCURE	stymie descriptor	unused
NICE	address of semaphore/handler	unused

Table 1: Scheduling messages as defined by FREE JAZZ.

becomes an issue. In general, both fixed-priority scheduling algorithms, which assign priorities to threads once and for all, as well dynamic-priority ones, which evaluate the assigned priority from request to request, are supported by FREE JAZZ, with *rate-monotonic scheduling* (RMS) and *earliest deadline first* (EDF) being their most prominent exponents, respectively. For fixed-priority schemes we assume that the constraint field of each message is always set to the user-defined global variable `No_constraint`, while otherwise the scheduler is called whenever a thread reads or receives a message to adjust the thread's priority accordingly. Anyway, `prio` is intended to hold the basic priority of a thread, that is, the one used to decide which thread should run next.[12] The priority of the root thread may be initially set via the user-defined global variable `Root_priority`.

The scheduling message most easily explained is READY, which tells the scheduler that a new thread has become ready and should be inserted into the ready queue. Its new priority should reflect the constraint passed on, which is taken from the message that caused the thread to become ready. As always, the scheduler may request a context switch to be performed subsequently if the priority of the just inserted thread exceeds the priority of the running one.

Whenever a handler's priority depends on the constraint associated with the current message to process, its priority usually would override the priority of the thread to whom the context belongs. Hence, FREE JAZZ automatically generates a SCHEDULE message with a PRIO SAVE value to indicate that the current priority should be saved before a new one, reflecting the accompanied constraint, is calculated. The scheduler must store a saved priority (unequal to the user-defined global variable `No_save_prio`) in the `save_prio` field to request an appropriate restore message (PRIO RESTORE) as soon as the handler is actually left, that is, its new queue became empty. Furthermore, it might become necessary to save more than one priority whenever a handler itself sends messages to another handler that is temporarily idle. In this case the scheduler should stack the saved priorities accordingly and use the `save_prio` field as pointer to an associated user-maintained stack space (not to be confused with a thread's function-call stack space). Note that in either case FREE JAZZ interprets `save_prio` simply as a flag that indicates whether or not a priority actually has been saved and, thus, a corresponding restore message is expected.

SCHEDULE messages with a NOP value, on the other hand, simply grant a reassignment of the CPU, either on behalf of the currently running thread or due to a preemption signal whose frequency also is user-defined via the global

[12] Again, albeit their declaration as `longs` it should be noted that any pointer may be stored within each priority field.

constant `Timer_quantum`. As far as SCHEDULE messages are concerned, this only leaves the NEXT value open for discussion. Whenever a thread is blocked that does not care about priority inversion, the scheduler nevertheless may request always to be called in with a SCHEDULE/NEXT message, contrary to the default behaviour of advancing to the first thread in the ready queue automatically. The blocked thread, for instance, might have consumed more processing time than anticipated, jeopardising the constraint of the message to be processed by the first thread in the ready queue. Hence, processing this message might cause a domino effect, endangering the constraints of even more messages. It is therefore sometimes reasonable to cancel the processing of a single message before it has started (even though it is not necessarily the message of the first thread in the ready queue that must be sacrificied). To this end, the scheduler has access to a thread's new queue to modify the contents of a cancelled message, effectively telling the recipient about its decision. It is not allowed, however, to remove a message; any error handling required should be performed by the recipient only.

As soon as priority inversion must be bounded to allow for (soft) real-time reasoning, additional measures must be taken. Again, a combination of scheduling messages and variables of `Thread` is used. Priority inversion may occur while calling a handler or procuring a critical section. For processes blocked at a semaphore and with their `tame_prio_inv` flag set, `procure()` sends an appropriate PROCURE message to the scheduler, uniquely identifying the semaphore and providing a pointer to the list of threads currently inside the critical section as part of the stymie[13] descriptor. In response, the scheduler may increase the priority of some or all of the threads inside to minimise and, more importantly, bound the waiting thread's blocking time. The blocking threads, however, may be blocked themselves at another semaphore, making it necessary to update the waiting lists of the semaphores concerned in accordance with the priority just inherited. Even more the priority of any threads inside these seamphores should be possibly adjusted, too, causing some kind of transitive priority inheritance. The thread's `semaphore` field therefore always points to the semaphore a thread is currently waiting for.

Anyway, the priority to restore must be saved in the `restore_prio` field to make sure that `vacate()`, conversely, generates a corresponding NICE message whenever a thread with an inherited priority finally leaves a critical section in question.[14] Thus, the scheduler may be called in twice within a single call to `vacate()`. The first message of type READY concerns the process chosen to enter the critical section next and is sent always. The second message, however, is of type NICE and therefore only sent if the priority of the vacating thread has been increased before due to some priority-inheritance protocol. Consequently, the scheduler is expected to perform a context switch in response to the READY message whenever the priority of the unblocked process exceeds the priority of the vacating thread, and should wait for the corresponding NICE message otherwise.

For processes blocked while calling a handler and with their `tame_prio_inv` flag set, an appropriate CALL message is sent to the scheduler, uniquely identifying the handler in question and providing a pointer to the thread currently

[13] **sty-mie** *n* **1** (in golf) situation on the green in which an opponent's ball is between one's own ball and the hole. **2** *(fig infml)* awkward or difficult situation. (Oxford Advanced Learner's Dictionary, 4th ed.)

[14] Note that `restore_prio` is interpreted by FREE JAZZ as a flag only, similar to `save_prio`.

using the handler as part of the accompanied stymie descriptor. The constraint field is taken from the message the thread was blocked on to judge the priority of the call correctly. Again, the priority to restore as soon as the handler is left by the blocking thread must be stored in the `restore_prio` field to receive an appropriate NICE message later on.

The `restore_reason` field, finally, may point to a user-implemented struct that records the unique identifiers mentioned before, since with nested priority inheritance it becomes necessary to associate the numerous NICE messages accordingly.

5 PERFORMANCE

To demonstrate the viability of our approach a number of experiments on various platforms have been performed, the results of four of which are summarised in Table 2. Of course, flexibility cannot be expected to come for free, yet to exactly quantify the overhead caused by opening the threads package a black-box variant of FREE JAZZ has been implemented. Numbers in bold face denote the measured results for the open implementation — giving an impression of the absolute performance of FREE JAZZ — and their relative performance in percent compared to those measured for the black-box implementation, which are shown in plain text.

It should be noted that all overhead denotes a worst-case loss in the sense that an application's overall performance will suffer from such a slow down only if it does virtually no processing besides sending messages and causing context switches. Consequently, the real-world overhead imposed by factoring out the scheduler will be somewhere between virtually nothing and the percentage given in Table 2, while exact figures are necessarily application-dependent.

The '*send* → *process*' benchmark measures the time needed to send a single message from one process to another one asynchronously, that is, without a context switch. Such a send operation mainly consists of allocating a new message and appending it to the recipient's new queue in FIFO order. The recipient is assumed to be in a ready-to-run state prior to receiving this message and, thus, there is no scheduler intervention at all. Within the '*send* → *handler*' benchmark, in contrast, the handler is assumed to be not in use and is therefore invoked synchronously for the message within the context of the sending process.[15] The handler itself does nothing in response to the message but simply returns.

The '*call* ⇄ *process*' and '*call* ⇄ *handler*' benchmarks, on the other hand, quantify the round-trip time of calling a process or handler, respectively, with an empty message. Again no processing is done by the recipient despite returning the message fields unmodified. For processes ('*call* ⇄ *process*') this operation causes, message allocation aside, two context switches and two scheduler invocations of type READY. Neither of these is needed when calling a handler ('*call* ⇄ *handler*') which explains the evident discrepancy in execution time. Although two scheduler invocations require two send operations to a scheduling handler, the overhead if compared to the black-box implementation is less than the time of two handler invocations (i. e., two '*send* → *handler*' operations).

[15] If the handler would be assumed to be in use, this benchmark and its results would be equal to the '*send* → *process*' benchmark.

	send $\rightarrow$ *process*	*call* $\rightleftarrows$ *process*
SunOS 4.1.3	2.21	68.84
SPARC10 Model 20	**2.34** (+**5.9** %)	**71.90** (+**4.4** %)
LynxOS 2.5.0	0.60	15.43
P166	**0.60** (±**0.0** %)	**16.07** (+**4.1** %)
Linux 2.0.30	0.59	2.78
P166	**0.59** (±**0.0** %)	**3.50** (+**25.9** %)
Linux 2.0.30	0.49	2.20
P200MMX	**0.49** (±**0.0** %)	**2.72** (+**23.6** %)

	send $\rightarrow$ *handler*	*call* $\rightleftarrows$ *handler*
SunOS 4.1.3	3.98	5.01
SPARC10 Model 20	**4.26** (+**7.0** %)	**5.28** (+**5.4** %)
LynxOS 2.5.0	1.04	1.25
P166	**1.06** (+**1.9** %)	**1.30** (+**4.0** %)
Linux 2.0.30	1.03	1.23
P166	**1.02** (−**1.0** %)	**1.26** (+**2.4** %)
Linux 2.0.30	0.85	0.97
P200MMX	**0.79** (−**7.1** %)	**0.95** (−**2.1** %)

Table 2: FREE JAZZ performance figures measured in μs.

Since scheduling handlers are necessarily non-preemptive it was possible to cut corners within FREE JAZZ and to speed up scheduler invocations beyond the time needed for the invocation of a standard, possibly preemptive handler.

On the SPARC processor the overhead of the open implementation versus its black-box variant affects all benchmarks, yet is thoroughly quite low. On the PENTIUM processor, in constrast, the overhead mainly shows up when scheduler invocations are involved. The bad absolute performance of the '*call* $\rightleftarrows$ *process*' benchmark under *LynxOS* is most probably due to a very slow implementation of the `setjmp()` and `longjmp()` library functions, as the otherwise competetive results indicate. These library functions are used in FREE JAZZ to perform context switches and, thus, their performance is crucial. The absolute overhead, however, is comparative to the one measured under *Linux*.

Under *Linux* in general and with the PENTIUM MMX processor in particular, slight cache anomalies show up, suggesting that the open implementation even can outperform the black-box variant for some benchmarks. For real-world applications, however, we expect that such effects will wear away.

6 CONCLUSIONS

As stated in the introduction, no user-level threads package has gained wide popularity among designers of multi-threaded run-time systems or middleware components at the time of writing. This is caused by adhering to the black-box principle, which has been the predominant line of thought when developing user-level threads packages in the past. However, due to the inherent inflexibility that results from this approach, design dilemmas must be solved once and for all, more often than not in mismatch with the requirements a developer of multi-threaded run-time systems tries to meet later on. As a consequence, developers commonly refrain from using a readily available user-level thread packages at all and usually develop yet another one on their own in the end.

Although the open implementation design methodology was specifically developed as a way to open traditional black boxes, the idea of a meta interface was systematically employed for a user-level threads package only recently by Haines, whose OpenThreads, however, are not concerned with (soft) real-time threading in particular [7]. With Free Jazz we propose an open user-level threads package that tries to fill this gap. The chosen interface, foremost intended as proof of concept, admittedly still must stand the test of time and, thus, refinements in response to future experience with Free Jazz are downright inevitable. Nevertheless, the performance figures suggest that the adopted course is promising at least and we will continue to explore the opportunities as well as the limitations of our approach likewise.

Specifically, we are in the process of developing an object-oriented variant of Free Jazz, called Cool Jazz, that will provide the flexibility of Free Jazz via type-safe customisation. Cool Jazz will become the execution environment of our QoS-supporting middleware architecture currently under development [12]. As part of this work we will also investigate ways to integrate user-level schedulers and kernel-level schedulers hierarchically [1, 6, 13], that is, we are ultimately aiming at first-class real-time threads.

ACKNOWLEDGEMENTS

First of all special thanks to Peter Buhler for providing the black-box thread package that served as the starting point for Free Jazz. Approximately 50 % of the original code survived virtually unchanged. Additional thanks (in alphabetical order) to Lothar Baum, Volker Hübsch, Rainer Koster, and Reinhard Schwarz for their valuable comments on this report during its various stages of preparation.

[1] T. E. Anderson, B. N. Bershad, E. D. Lazwoska, and H. M. Levy. Scheduler activations: Effective kernel support for the user-level management of parallelism. In *Proceedings of the Thirteenth Symposium on Operating System Principles (SOSP)*, October 1991.

[2] T. E. Anderson, E. D. Lazowska, and H. M. Levy. Performance implications of thread management alternatives for shared memory multiprocessors. *IEEE Transactions on Computers*, 38(12):1631–1644, December 1989.

[3] B. N. Bershad, E. D. Lazowska, and H. M. Levy. PRESTO: A system for object-oriented parallel programming. *Software — Practice and Experience*, 18(8):713–732, August 1988.

[4] B. N. Bershad, E. D. Lazwoska, H. M. Levy, and D. B. Wagner. An open environment for building parallel programming systems. *ACM SIGPLAN Notices*, 23(9):1–9, September 1988.

[5] D. Finkelstein, N. C. Hutchinson, D. J. Makaroff, R. Mechler, and G. W. Neufeld. Real-time threads interface. Technical Report TR-95-07, Department of Computer Science, University of British Columbia, Canada, 1995.

[6] P. Goyal, X. Guo, and H. M. Vin. A hierarchical CPU scheduler for multimedia operating systems. In *Proceedings of the Second Symposium on Operating Systems Design and Implementation (OSDI)*. USENIX, October 1996.

[7] M. Haines. On designing lightweight threads for substrate software. In *Proceedings of the 1997 Annual Technical Conference*, pages 243–255. USENIX, 1997.

[8] D. Keppel. Tools and techniques for building fast portable threads packages. Technical Report UWCSE 93-05-06, Department of Computer Science and Engineering, University of Washington, May 1993.

[9] G. Kiczales, R. DeLine, A. Lee, and C. Maeda. Open implementation analysis and design of substrate software. In *Tutorial Notes, OOPSLA '95*. ACM/SIGPLAN, October 1995.

[10] G. Kiczales, J. des Rivieres, and D. G. Bobrow. *The Art of the Metaobject Protocol*. MIT Press, 1991.

[11] D. E. Knuth. *The Art of Computer Programming: Fundamental Algorithms (Third Edition)*. Addison Wesley, 1997.

[12] T. Kramp and R. Koster. A service-centred approach to QoS-supporting middleware. Work-in-Progress Paper presented at *Middleware '98 (IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing)*, September 1998.

[13] S. L. Ann Lo, N. C. Hutchinson, and S. T. Chanson. Architectural considerations in the design of real-time kernels. In *Proceedings of the Fourteenth Real-Time Systems Symposium*, pages 138–147. IEEE, December 1993.

[14] B. D. Marsh, M. L. Scott, T. J. LeBlanc, and E. P. Markatos. First-class user-level threads. In *Proceedings of the Thirteenth Symposium on Operating System Principles (SOSP)*, October 1991.

[15] S. Oikawa and H. Tokuda. User-level real-time threads: An approach towards high performance multimedia threads. In *Proceedings of the Fourth International Workshop Network and Operating System Support for Digital Audio and Video (NOSSDAV)*, pages 66–76, November 1993.

[16] J. H. Saltzer, D. P. Reed, and D. D. Clark. End-to-end arguments in system design. *ACM Transactions on Computer Systems*, pages 277–288, 1984.

[17] L. Sha, R. Rajkumar, and J. P. Lehoczky. Priority inheritance protocols: An approach to real-time synchronisation. *IEEE Transactions on Computers*, pages 1175–1185, September 1990.