A Multithreading Platform for Multimedia Applications

Rainer Koster and Thorsten Kramp

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

ABSTRACT

Complex multimedia applications have diverse resource and timing requirements. A platform for building such programs therefore should supply the developer with mechanisms for managing concurrency, communication, and real-time constraints but should remain flexible with regard to scheduling policies and interaction models. We have developed such a platform consisting of a user-level threads package and operating system extensions. The threads package offers a message-based threading model uniformly integrating synchronous and asynchronous communication, inter-thread synchronisation, and signal handling as well as real-time functionality and application-specific scheduling. To support this user-space flexibility an upcall mechanism links the user-level scheduler to the kernel.

Keywords: User-level threading, application-specific scheduling, message-based inter-thread communication, scheduler activations, upcalls

1. INTRODUCTION

Complex multimedia applications need to cope with a variety of timing constraints. Data from different sources needs to be received, buffered, processed, synchronised, and played-out. The resource allocation of these tasks in general and their timing in particular determine the quality of service (QoS) that is provided to the user. While threads are widely recognized as a useful abstraction for structuring these different activities, desirable support by a multithreading platform should take manifold characteristics into account:

- Multimedia applications are largely event-driven: Most actions are taken in response to timers, arriving pieces of data, or events from the user interface.
- Activities ranging from decoding video frames to transferring samples to the audio device at the right moment are diverse in terms of execution times and timing constraints. Stream synchronisation and jitter of received data add dynamic dependencies between different tasks. A scheduler must deal with these timing requirements in an application-specific way.
- Multimedia applications are I/O intensive. Hence, input and output of data should be well integrated with the platform.
- Integration with commonly used operating systems facilitates everyday-use on computers also running other programs. Standard graphics and interface libraries as well as drivers, for network adapters or frame grabbers for instance, are readily available, in this case.

We are developing a platform addressing these requirements. A key component is a user-level thread package called Cool Jazz that uses a message-based threading model.[1] Threads work like state machines reacting to messages and sending messages to other threads, either synchronously or asynchronously. Using messages rather than plain shared memory and semaphores for inter-thread communication and synchronisation is more intuitive for many applications and less susceptible to errors. Moreover, messagess are a sound basis for discrete notifications as well as for continuous media streams. Nonetheless, Cool Jazz does not preclude other styles of communication and also provides locks and semaphores for situations in which they may be more appropriate.

Providing the right scheduler is a critical capability for a thread package. However, there will not be one optimal scheduler for all applications and in some cases it is even desirable to use a specific scheduler that is aware of application semantics. To provide this versatility, we have not built one or a few schedulers into Cool Jazz but

Email: {koster,kramp}@informatik.uni-kl.de

provide a well-defined interface between the thread package and the actual scheduler. In this way, the developer may plug in an available scheduler suitable for a particular application or even build a specific one.

Extensive real-time support has been integrated with Cool Jazz. Besides thread priorities, constraints such as deadlines can be attached to messages, automatically associating the scheduling of a thread to the urgency of the job it is currently performing. To bound priority inversion, support for priority-inheritance has been added to the message processing and synchronisation primitives with appropriate calls to the scheduler.

User-level threads on their own have some drawbacks such as the impracticality of SMP utilisation and blocking system calls. With some effort, user-level threads can be mapped onto a set of kernel level threads to run on multiple CPUs simultaneously. Blocking system calls are typically used for I/O operations. With a plain user-level thread package these operations would halt the entire application. A work-around is using non-blocking counterparts that either require polling or asynchronous notifcation by signals. Exclusive use of kernel-level threads avoids these problems, but solely relies on the scheduler of the operating system. The benefits of application-specific policies as discussed above cannot be used. In addition, user-level threads provide better scalability due shorter creation, deletion, and context switching times. Hence, neither pure user-level nor kernel-level threads are satisfactory for a multimedia platform.

An efficient approach for interaction between kernel and user-level scheduler are scheduler activations.[2] The kernel notifies the scheduler of the application whenever one of its threads could trigger a scheduling decision, for instance, if a thread blocks or deblocks in the kernel. In these cases, an upcall directly to the user-level scheduler causes an application-specific scheduling decision, which thread to run next. With activations, both flexible user-defined scheduling and convenient I/O programming can be provided. Additionally, the upcall mechanism can be extended to support asyncronous I/O operations more efficiently than signals.

This kind of system support, however, is not commonly found in general-purpose operating systems, but mostly in specialised multimedia OS such as Nemesis.[3] There are, however, several reasons for using an off-the-shelf OS: Multimedia applications frequently need to control hardware devices such as sound or frame grabber cards, which require device drivers readily available only for commonly used systems. Also software libraries for image processing or building graphical user interfaces can be more easily reused on these standard platforms. Moreover, many multimedia applications are used on desktop computers along with other programs rather than on dedicated machines with a specialised system. Due to these reasons, we have integrated support for Cool Jazz into the Linux operating system. The changes in the kernel are limited to few locations in the source code and have virtually no effect on other applications.

Section 2 discusses the basic principles of our platform. Its funtionality in user- and kernel-space is presented in Sections 3 and 4 respectively. Section 5 discusses related work and Section 6 outlines conclusions and future work.

2. THREADING MODEL

Complex multimedia applications usually consist of many interacting components, such as a graphical user interface, some control unit processing user commands, video and audio streams, and so on. Processing of continuous media streams usually can be further subdivided into tasks such as retrieving data, buffering, decoding, scaling, buffering one more time, and finally delivery to the output device. Mapping these activities on threads facilitates the design of this kind of application. Often threads implement a stage of a pipeline reacting to some input events similarly to a state machine. Starting from this insight, in this section we describe characteristics of a threading model well-suited for a multimedia platform.

2.1. Message-Based Interaction

A straightforward approach to building an application that reacts to network packets, timer interrupts, and GUI events, for instance, is running one thread of control in a main loop waiting for events and calling the respective handler functions. While this design does not require any synchronisation effort and no context switches, there are two main drawbacks: An event handler that needs to wait for a further event will block the entire system. Moreover, a long running handler can unacceptably delay the processing of more urgent events. Finally, multiple processors cannot be used.

With multithreading, in contrast, several threads of control with their own state can act independently of each other. While one thread needs to wait for some event, others may do useful work. Moreover, a scheduler can interrupt

threads and control the timing of their execution. Communication between threads, however, introduces considerable complexity.[4,5] Data is usually exchanged via shared-memory with access to it being synchronised by semaphores, for instance. Errors in applying synchronisation primitives easily lead to data corruption or deadlocks – often in an non-deterministic, hard to debug manner. Additionally, fine-grained locking may introduce significant overhead, coarse-grained locking may unnecessarily reduce parallelism.

We have proposed a combination of both approaches using an message-based model for inter-thread communication.[1] Each thread has a code function, which, in contrast to conventional threads, is not called at creation time but each time the thread receives a new message. Hence, it resembles the body of an event loop. In the code function, the thread can wait for further messages and send messages to other threads. Sending may be done either *synchronously* if there remains nothing to do for a thread until a reply is received, or *asynchronously* whenever a reply is not needed immediately to continue or no reply is required at all. Messages provide a general means of inter-thread communication and can be used for synchronising access to shared data. This reactive style of programming tends to be less prone to synchronisation errors.

In this processing model, threads represent *extended finite state machines* with the code function implementing the state transition function. Since this approach closely resembles that of SDL, the transformation of SDL specifications to implementations is facilitated. Moreover, in contrast to shared memory, message-based communication can be easily extended to distributed applications. The platform just needs to be able to send messages to threads in different address spaces and on different nodes. According to RM-ODP,[6] events are a suitable infrastructure for discrete interaction as well as for continuous media streams. Even asynchronous notifications such as signals can be mapped to messages. By this versatility, the programming model can uniformly handle all types of events.

2.2. Application-Specific Scheduling

It is well known that the conventional UNIX scheduler is not suitable for multimedia applications.[7] While the POSIX real-time extensions provide some more control over scheduling, it still is a very inflexible mechanism. Looking for better support, a variety of scheduling algorithms for multimedia applications has been proposed.[8,9,3,10,11] Being able to use the kernel scheduler that is most suitable for a paritcular application is highly unlikely, because system-wide scheduling is necessarily subject to some limitations: The scheduler is hardcoded in the OS, it must work reasonably well for all types of multimedia applications as well as for interactive and background computations, and the kernel must protect applications from each other and enforce CPU allocation policies.

In contrast to that, scheduling among threads of one application can be done a lot more efficiently. Threads may rely on each other to work cooperatively. Moreover, application-specific semantic information can easily be exploited in scheduling decisions. Consider, for instance, a feedback-driven real-rate scheduler,[10] which adaptively schedules tasks based on their progress. While providing semantics information about progress to the kernel scheduler via fixed system interfaces in general is not simple, this algorithm can be efficiently used for scheduling threads of one application. A user-level scheduler can have a clear notion of progress and have detailed information about inter-thread dependencies. For these reasons, we favour user-level threads for building complex applications with timing constraints.

For designing a user-level thread package there still is no optimal scheduler for all scenarios. Even providing a set of predefined policies cannot avoid all *design dilemmas*. We therefore apply the *open implementation* design methodology[12,13] and enable the user to control all critical design decisions. All scheduling code has been separated from the rest of the thread package allowing the user to plug-in a specific scheduler via a well-defined interface. This scheduler is called whenever some incident might induce a scheduling decision. Typical examples are thread state transitions or timer interrupts. The platform also needs to support timing constraints, which can also be defined in an application-specific manner. In this way, for instance, deadlines can be attached to the processing of messages, or fixed priorities can be associated with threads as needed. Moreover, the thread package should support the user-defined scheduler in bounding priority inversion by notifying it whenever a higher-priority job waits for a lower-priority job to make progress.

This flexible approach is beneficial whether or not there are guarantees from the system. The user-level scheduler has detailed knowledge about timing constraints, slack times, and inter-thread dependencies, and can easily monitor the progress of all subtasks. Hence, it can efficiently use reserved CPU capacity as well as adapt to resource fluctuations. Nonetheless, kernel support for resource reservation is desirable. As a basis for predictable user-level platforms, we are investigating proportional share policies such as lottery and stride scheduling.[11,14,15]

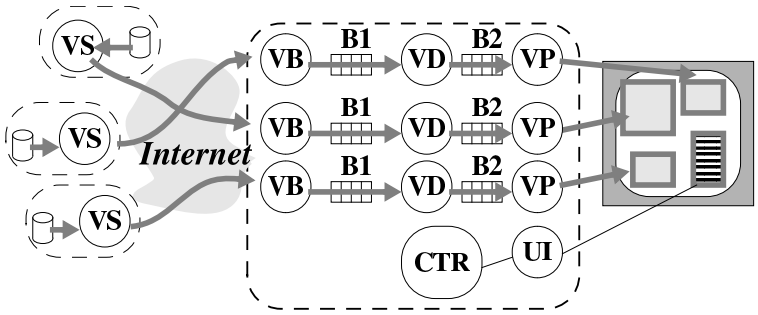

Figure 1. Player with multiple pipelines

2.3. Application Scenario

Consider a video and audio player that shows complex presentations composed of several streams retrieved over the Internet in real time, for example the player developed in the QUASAR project.[16,17] For streams that are played in parallel, several video and audio pipelines must be run simultaneously as shown in Figure 1. Each of these pipelines typically consists of stages for receiving and defragmenting data from the network, adapting to network bandwidth, for decoding, and finally for displaying frames and playing audio samples. Between these stages, frames and samples need to be buffered, and these shared memory buffers need to be protected by semaphores. A further component is needed to control the timing and synchronise the streams.

With a platform as proposed above, the design of similar applications of many interacting components is significantly simplified. Frames and groups of audio samples are put into inter-thread messages and are attached timing constraints according to their playout time. By modelling the pipeline stages as reactive threads there is no need to explicitly maintain and synchronise buffers any more, because messages are automatically queued by the platform. Timing and stream synchronisation can be directly controlled by an application-specific scheduler rather than a clock component notifying other threads of timing events.

3. USER LEVEL SUPPORT

To implement the threading archtitecture outlined above we have developed a user-level threads package called Cool Jazz.

3.1. Message-Based Threading

As introduced in the previous section, Cool Jazz threads work like extended finite state machines reacting to messages received. Unlike conventional threads, their code function is not called at thread creation time, but each time a message is received. When the processing is finished the code function returns. Unless indicated by a special return code, however, the thread is not terminated, but waits for further messages. In the code function, a thread can send messages to other threads either asynchronously or synchronously. In the latter case, it blocks waiting for the reply. A thread may also explicitly block to wait for new messages.

Figure 2a shows the structure of a thread. Besides a code function, a Cool Jazz thread consists of one *new-queue* and at least one *save queue*. The threading platform places messages to a thread into its new-queue containing all messages the thread has not looked at yet. Additionally, a thread may maintain save-queues to intermediately store tasks that can only be completed at a later time, for instance after receiving a reply from a thread performing a subtask.

A Cool Jazz thread can dynamically enable or disable preemption. If preemption is turned off, the entire code function becomes the unit of mutual exclusion and a context switch can only occur when the thread finishes processing a message or blocks. To provide more fine-grained control among threads actually competing for specific resources, also shared memory synchronisation by locks and semaphores is supported. Hence, developers may choose the most appropriate interaction model, although Cool Jazz encourages message-based synchronisation.

In addition to local interaction, remote communication can also easily be mapped on inter-thread messages. To do so, we have developed a flexible binding framework providing ports that forward messages to remote threads using some transport protocol.[18] Moreover, signals from the operating system indicating asynchronous events such

as timer interrupts are transformed into messages. In this way, Cool Jazz messages provide a uniform abstraction for concurrency, communication, and signal handling.

Modelling threads as extended finite state machines and using explicit messaging for inter-thread communication has been proven to be an effective means of expressing concurrency and eliminating synchronisation errors. Predecessors of Cool Jazz have been used in a number of larger projects such as a fault-tolerant distributed shared memory[19,20] and within the core of a mobile agent platform.[21]

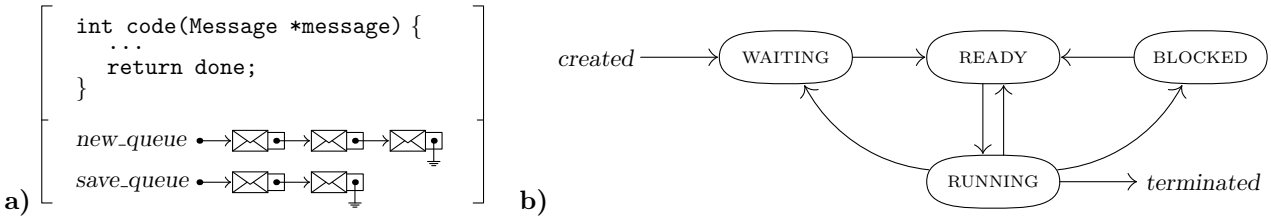

Figure 2. Cool Jazz threads. a) structure b) state transitions

3.2. Real-Time Support

Cool Jazz provides extensive support for timing constraints. In addition to thread priorities, constraints may be attached to messages.* The priority of a thread is derived from the constraint on the message it currently processes or, if the thread is in the ready state, on the constraint of the first message in its new queue. If no constraint is specified for the respective message, the priority statically assigned to the thread is used. That is, whenever an idle thread receives a message and, thus, its state changes to ready, the scheduler is invoked to calculate an appropriate priority from the message constraint, and to insert the thread into the ready queue. If further messages arrive, they are inserted into the new-queue according to their constraint. Then, the scheduler is invoked, only if the new message becomes the first one in the queue, because in this case a more important message has been received and the priority of the thread has to be recalculated. The scheduler reorders its ready queue according to the new thread priorities. The priority of a thread is reset according to the next message whenever it finishes processing a message with a constraint by sending it to another thread, by storing it in a save-queue, by deleting it, or by explicitly reading the next message. Note that for having subtasks processed by other threads in parallel new messages with the same constraint can be created, while passing on a message also transfers the priority derived from its constraint.

This approach supports scheduling based on static priorities by using thread priorities only, on dynamic priorities by using message constraints only, or on a mixture of both. Additionally, admission tests can be easily integrated in new-queues by optionally rejecting messages.

To bound priority inversion, priority inheritance[22] is supported in semaphores as well as in message queues. A low-priority thread holding a semaphore inherits the priority of a more important thread waiting in a P-operation. Similarly, if a thread is processing a message at a priority lower than that mandated by the constraint of the first message in its new-queue, it also inherits this priority. Both mechanisms deal with priorities and, hence, are compatible to each other. Note that the scheduler is free to implement other priority inheritance protocols.

In a dynamic real-time environment it may be necessary to asynchronously terminate a thread that cannot meet its deadline any more. In general, such a thread cannot simply be killed because it may hold some resources that must be freed again. To handle this case, Cool Jazz provides two mechanisms similar to POSIX cancellation points. A thread may indicate when asynchronous termination is save. When this flag is not set and a termination request arrives, the request is queued until termination is enabled again. Additionally, a thread may register an abort-function, which is called at asynchronous termination and can perform any cleanup tasks such as releasing resources held by the thread.

*Priorities as well as constraints are scheduler-specific (and, hence, user-defined) and are not necessarily represented by mere numbers.

3.3. Application-Specific Scheduling

Since there is no optimal scheduler for all applications that can be built with a thread package, Cool Jazz supports application-specific schedulers. Hence, the actual scheduling code has been separated from the rest of the thread package. The platform provides an interface for a user-defined scheduler that is called whenever a scheduling decision may be necessary. Such events are thread state transitions as shown in Figure 2b, priority adjustments, and external events such as when a thread becomes ready, when a reassignment of the CPU occurs due to preemption, when the active thread runs out of messages and becomes idle, when priority inheritance needs to be performed, when a inherited priority needs to be reset, when a timer expires, or when a signal arrives.

Besides the scheduler itself, other relevant components of the system must be customisable. The type of priorities or constraints used, for instance, is not defined by Cool Jazz, but may be chosen matching the scheduler. Moreover, the new queues have to take message constraints into account in ordering its contents and, in this way, schedule when messages processed by the thread. These queues collaborate with the scheduler and may be user-defined in a similar way.

4. KERNEL LEVEL SUPPORT

Pure user-level threading approaches incur two main drawbacks: the inability of utilising multiple processors on SMP machines and the impossibility of using blocking system calls. SMP support can be added by mapping user-level threads to a few kernel level-threads running on different CPUs. This functionality mainly requires an extension of the thread package. Convenient integration of blocking system calls, however, requires support by the kernel, which is discussed in this section.

A user-level thread blocking in the kernel will block the entire application, although other user-level threads may be runnable. A work-around is using non-blocking variants, where possible. For instance, read and write operations on sockets on UNIX systems may be configured to always return immediately, indicating that the operation could not be performed yet, if necessary. By their own, these calls could only be used to periodically poll all applicable sockets. On most systems a signal can be requested whenever *some* socket has become ready. If there is support for POSIX real-time signals, a signal can also carry information what socket caused its generation. In any case, the actual system call must be performed in addition to the signal processing. While a user-level thread package can encapsulate this functionality, it incurs some management overhead and inconveniently restricts the application programmer from utilising synchronous calls. Note that using a central blocking select call is not compatible with concurrently running user-level threads. For accessing files, a similar work-around is possible only if asynchronous I/O operations (as defined in POSIX 1003.4,[23] for instance) are supported. Moreover, device drivers for hardware such as a video frame grabber card may exclusively communicate via blocking calls and provide no alternative at all. Since multimedia applications are typically I/O-intensive, the problems outlined above are very common. Hence, it is not reasonable to preclude the use of blocking system calls on a multimedia threading platform. The developer should have the choice of synchronous and asynchronous I/O according to the requirements of the application.

A way for overcoming these shortcomings are upcalls from the kernel to user space that notify the user-level scheduler of any relevant event in the kernel.[2,8] *Scheduler activations* as proposed by Anderson et al. make upcalls whenever a kernel thread is made available to the application, has been preempted, has blocked, or has unblocked. In this way, the user-level scheduler always knows what kernel threads are available and has complete control over assigning user-level threads to them, while kernel- and user-level scheduling policies remain completely independent of each other. Hence, the user-level platform can be used with a scheduler that supports some kind of guarantees or even with the regular UNIX scheduler. Although there is only best-effort scheduling in the latter case, activations enable the user-level scheduler to optimally assign the allocated CPU among its threads in an adaptive way.

As part of our platform we have developed a similar mechanism for the Linux operating system, which is described in the following sections. For brevity, the abbreviations *u-thread* for user-level thread and *k-thread* for kernel-level thread are used.

4.1. Design

For clarity, we first describe the design of the activation mechanism for the uniprocessor case. It can be assumed that only one k-thread of an application is running. SMP support is discussed afterwards.

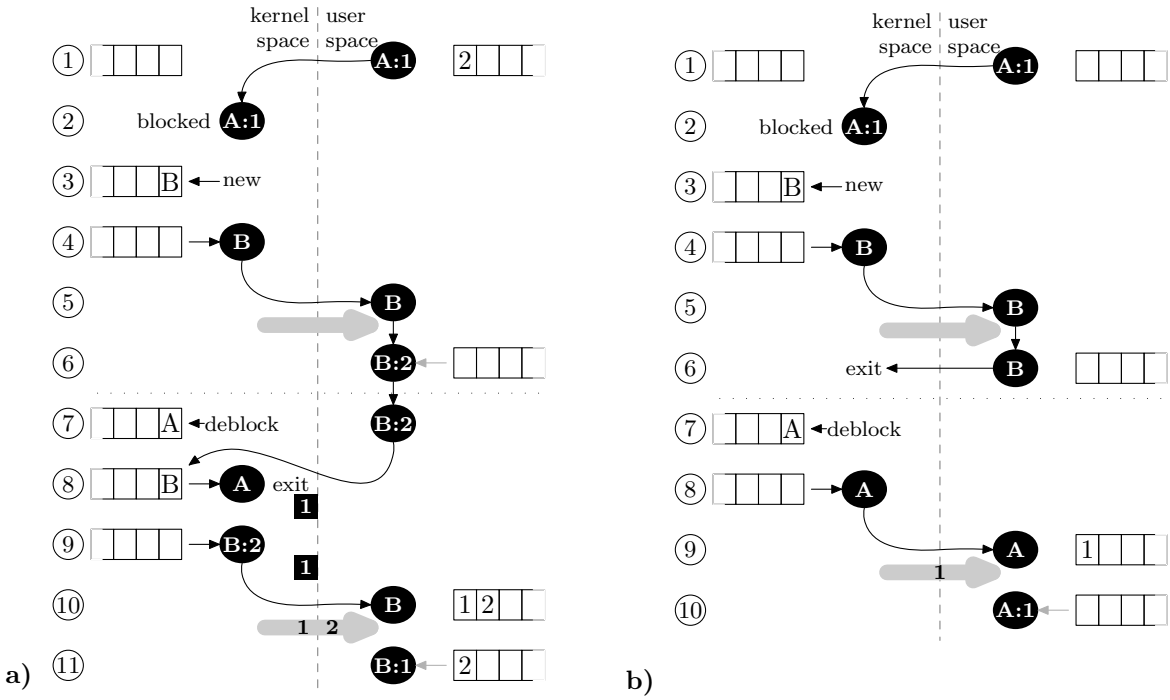

Figure 3. Upcalls for a blocking system call. a) with other u-threads b) no other u-threads

An application can register an upcall handler to be notified of all kernel scheduling events affecting its k-threads. Messages about these events are queued in the kernel. Whenever one of the k-threads of that application returns from the kernel to user-space (after a system call or after being scheduled, for instance) and messages are pending, the upcall handler is invoked and passed all event notifications. The user-level threads then can be rescheduled to reflect the new situation.

There are the following types of upcall messages:

block	The k-thread of the application has blocked in the kernel. The message contains an identifier of the blocked k-thread.
deblock	The k-thread has deblocked. The thread only returns to user-space if no other thread of the application is running. Otherwise it saves its context in the kernel and exits. The other thread picks up the context and sends it to the user-level scheduler by an upcall.
preemption	The k-thread has been preempted. In the uniprocessor case, there is no other k-thread left to do an upcall at preemption time. This message can merely be used to notify the user-level scheduler of the elapsed time when the k-thread regains the CPU.
this	Any k-thread of an application may be used for performing an upcall. This thread calls the upcall-handler and does not immediately return to its user-space computation. Since this u-thread is to be resumed at some time, its context is included in a *this* message and can later be resumed by the user-level scheduler.

Figure 3a shows events caused by a blocking system call. k-threads are identified by characters and u-threads by numbers. Ready queues for kernel- and user-level scheduler are shown as well as threads running in the kernel or in user space. At the beginning, the application is assigned a k-thread **A**. The user-level scheduler has scheduled the u-thread **1** while **2** waits in its ready queue. **A:1** denotes u-thread **1** running on k-thread **A**.

1. **A:1** performs a blocking system call. It enters the kernel.
2. **A:1** blocks inside the kernel.
3. Another k-thread **B** is assigned to the application, either from a pool or newly created.
4. Since **A** is blocked, the kernel schedules **B** now.
5. **B** performs an upcall informing the user-level scheduler that **A** has blocked.

6. The user-level scheduler assigns u-thread **2** to its k-thread **B**.
7. At some time, **A:1** deblocks in the kernel and becomes ready.
8. Some time later, the kernel scheduler will preempt **B:2** and run **A:1**. Two k-threads of one application would now compete with each other for the CPU, while the user-level scheduler should decide whether **1** or **2** should continue. To induce this decision, k-thread **A** saves the context of **1** and terminates.
9. The kernel schedules **B:2** again.
10. **B** performs an upcall passing a *deblock*-message containing the context of **1** and a *this*-message containing the context of **2**.
11. The user-level scheduler inserts **1** and **2** in its ready queue and chooses, which of them to run on **B**. In this example, it schedules **B:1**.

A slightly different procedure is necessary, if the entire application becomes idle. Figure 3b shows the same scenario as Figure 3a at the beginning except that **1** is the only runnable u-thread. Steps 1 through 5 are the same as in the previous example.

6. The user-level scheduler does not have any u-thread to run and, hence, terminates its k-thread **B**.
7. At some time, **A:1** deblocks in the kernel and becomes ready.
8. **A:1** is run by the kernel scheduler.
9. Since there is no other k-thread of the application running, **A** does not terminate, but performs the upcall with a *deblock*-message containing the context of **1**. In this special case, there is no *this* message in the upcall.
10. The user-level scheduler runs its only u-thread as **A:1**.

In most cases described above, threads returning to user-space carry a u-thread context. These threads have been set up before they entered the kernel and have their own stack. The case of a blocking thread (as in steps 3 to 5 above) is an exception, because it involves the creation of a new kernel thread. For it, a temporary stack is necessary. When the upcall has completed, the temporary stack may be reclaimed, because either the application is idle and the thread has terminated (step 6 in Figure 3b) or a user-level thread with its own stack is assigned to the new kernel level thread (step 6 in Figure 3a).

On a multiprocessor machine, scheduler activations are even more useful. Consider an application running two u-threads **1** and **2** with **1** having a higher priority. Also two kernel threads **A** and **B** are assigned to that application, so **A:1** and **B:2** are run. If now the kernel scheduler decides to preempt **A**, u-thread **2** is running while **1** with a higher priority is stuck in the kernel ready queue. Hence, when **A** is preempted, **B** needs to be interrupted to perform an upcall carrying both contexts **1** and **2** to let the user-level scheduler choose again. We have not implemented the SMP support yet, because we are still working on an SMP version of Cool Jazz.

Some upcall messages contain the context of a thread, which needs to be captured in the kernel. Fortunately, this state is largely saved by the system, anyway. Whenever a trap or interrupt occurs, the processor switches from the user-space stack to a stack inside the kernel. Then, the OS saves most registers on the kernel stack from where they are retrieved at return to user space. This state can be copied into upcall messages and can be manipulated to perform the upcall, that is, to jump to the registered handler and pass the messages to it.

In user space, care must be taken of locking when calling functions of the thread package itself. The core of the package including scheduler and dispatcher maintains data structures such as the ready queue that must be accessed in a synchronised way. Particularly, user-level context switches are performed while access to the core is locked. Because the upcall handler has to interact with the scheduler and upcalls can be triggered asynchronously to the thread package, they must be deferred when the scheduling core is locked. Moreover, when dispatching a u-thread the two types of contexts must be handled separately. When resuming a user-level context that has been saved inside the core of the thread package, the lock must remain set, because the thread resumes inside the core. Contexts from an upcall, however, resume outside the core and, hence, the lock must be released when jumping to them. The scheduling core itself must not do any blocking system calls, because an upcall could be triggered while the lock remains set resulting in a deadlock.

We are working on a more fine-grained synchronisation mechanism. One approach is to decouple upcall-handler and the actual scheduler. Reentrant upcall handlers can be called any time and simply queue their messages in the threads package. They activate the scheduler only if the core is not locked. If the core is locked the the scheduler itself has to check for new messages whenever the core lock is reset.

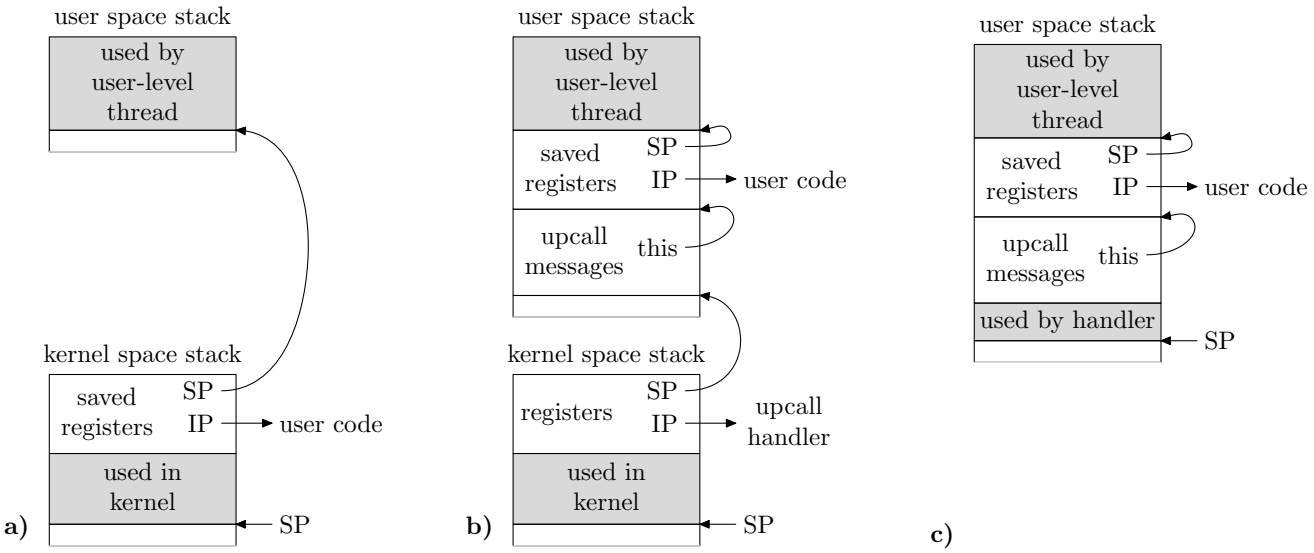

Figure 4. Stack contents. a) thread in kernel, b) thread in kernel prepared for upcall, c) in an upcall handler

The upcall mechanism has been extended to provide better support for asynchronous I/O as well by replacing timer and I/O signals. To deliver a signal, the kernel switches to user-space and calls the signal handler. The handler traps back into the kernel, which then again returns to the point in the user-program where is has been interrupted by the signal. An upcall, in contrast, jumps from the kernel to the user-level thread package calling the appropriate handler. Then, the user-level scheduler on its own resumes the execution of its interrupted thread. Hence, there is only one switch between kernel and user space rather than three. With more elaborate support, upcalls can even be extended to carry data, from network packets for instance.

4.2. Implementation

At the time of writing, we have implemented the upcalls for uniprocessors in a Linux 2.2.14 kernel. For controlling them only one system call had to be added: `int upcallctr(void* handler,int* lock, unsigned long* stk);` `handler` identifies the upcall handler for the application, `lock` points to a flag indicating whether a thread runs in the scheduling core and no upcalls are allowed at that time, and `stk` provides the temporary stack mentioned above.

Modifications of the kernel were limited to few places: In the scheduler the relevant events are detected and queued for upcall. In the `exit`-syscall the core lock must possibly be reset. Finally, when a k-thread leaves the kernel, it is checked whether the thread has deblocked or an upcall is pending. A deblocked k-thread saves its context for the next upcall and exits. Pending upcalls are performed when the user-level scheduling core is not locked. All pending messages are saved on the stack of the upcalling thread and its return context at the top of its kernel stack is changed to jump to the upcall handler. Finally, this context is restored when leaving the kernel.

Figure 4 illustrates how upcalls are performed. Figure 4a shows the user-space and the kernel stack, when a thread runs inside the kernel. At the (logical) bottom of the kernel stack the register contents have been saved and are restored when returning to user-space. The stack pointer (SP) points to the top of the user stack, the instruction pointer IP to the user code, from where the kernel was entered by a trap or interrupt. When an upcall is performed as shown in Figure 4b, the saved registers are put onto the user stack for later retrieval by the user-level scheduler as part of the *this* message. Then, the actual messages are pushed onto that stack. Finally, the register contents on the kernel are manipulated to perform the upcall: IP points to the upcall handler, SP points behind the message to provide a stack the handler may work on without overwriting data. The upcall handler receives a pointer to the message buffer structured as described below. Messages consist of an ID and an optional argument:

ID	`THIS`	`BLOCKED`	`DEBLOCKED`	`END`
argument	context of upcalling thread	thread ID	context of deblocked thread	

The `END` message indicates the end of the message queue. While the handler is running the stack typically consists of the following parts as shown in Figure 4c: The stack before the system call or interrupt, the context of the thread before exiting the kernel, the upcall messages, and the data put on the stack by the handler function itself.

To suspend and resume u-threads, the thread package uses `setjmp` and `longjmp`. These contexts, however, differ considerably from those captured in the kernel. The latter ones are larger than the few registers saved in `setjmp`, because for `setjmp` the compiler makes sure that only these registers are actually in use. To avoid dealing with two types of thread contexts in the user-level scheduler, the upcall handler creates for each context from the kernel a `setjmp`-compliant context that points to a restore-function that loads the entire context. Moreover, this function releases the lock of the scheduling core. In this way, in user-space the efficient `setjmp`-based context switching can be used and upcall-context can be handled uniformly. Interaction between upcall handler and scheduler becomes very simple: Each context from an upcall message (*this* or *deblock*) is added to the ready queue and the scheduler is called to dispatch a user-level thread.

Care must be taken when the application becomes idle and the upcalling thread is terminated. A call to `exit` would cause the C standard library to perform cleanup tasks and release resources that are still needed. These functions must be bypassed by exiting directly via the system call `_exit`.

We have performed initial performance measurements on a 350 Mhz Pentium II. A context switch between Linux kernel-level threads takes about 6.7 μs, a switch between Cool Jazz user-level threads about 1.5 μs. As expected, user-level switches are considerably faster, but the absolute overhead is small in either case. More importantly for providing flexible user-space scheduling, the overhead of upcalls is acceptable: From the time a thread initiates a blocking system call to the time another user-level thread is scheduled it takes about 10 μs. This delay includes the trap into the kernel, scheduling another kernel thread from a pool, performing the upcall, and scheduling another thread in Cool Jazz.

5. RELATED WORK

In the SUMO project upcall-driven application structuring has been identified as a useful principle in supporting multimedia applications on micro-kernels.[24] Communication events are initiated by the system, which calls user-level handlers that have been attached to I/O ports. While this approach mandates asynchronous I/O, in our system upcalls are also used for interaction between kernel and user level scheduler to enable the application to efficiently use synchronised, blocking I/O operations as well.

Real-time upcalls also use an upcall mechanism as a basis for implementing communication protocols in user space.[25] In this way scheduling and data movement are combined, and a high efficiency is achieved. Moreover, the system provides QoS guarantees to the protocol handlers.

The original split-level scheduler[8] uses upcalls called user-interrupts to directly invoke the user-level scheduler from the kernel. The scheduling in the kernel takes deadlines of user-level threads into account, tightly coupling both schedulers and requiring cooperative behaviour. Moreover, this approach focuses on asynchronous I/O, too.

Due to the deficient multimedia support of schedulers in general purpose operating systems such as the commonly used UNIX scheduler,[7] a variety of scheduling algorithms have been proposed.[8,9,3,10,11] Although we also investigated an extension of a stride scheduler,[15] our platform does not rely on resource reservation from the kernel. If there are some guarantees, the user-level scheduler can take advantage of them, if not, it can work adaptively.

Scheduler activations have been proposed as an efficient means of coordinating kernel and user level schedulers while keeping their actual scheduling policies independent of each other.[2] The mechanism allows to combine the performance and flexibiliy of user-level threads with the possibility to use multiprocessors and blocking I/O supported by kernel-level threads. Activations have been used in research OS such as Nemesis.[3] Recently, a variant has been implemented in Linux to support a library for parallel multithreaded programming.[26,27]

On the user-level, Cool Jazz is similar to GOPI, which also aims at providing a middleware platform for multimedia applications.[28,29] It also implements user-level threads and allows for user-defined schedulers. In contrast to Cool Jazz, however, GOPI uses the conventional threading model and the interface for user-defined schedulers cannot be easily extended. Moreover, some real-time functionality such as priority inheritance is not directly supported by GOPI.

With respect to flexible scheduling, our work is particularly related to OpenThreads.[30] In this work, also the notion of an open implementation has been systematically exploited, and a user-defined scheduler is invoked whenever a thread state transition occurs. OpenThreads, however, has not been developed with real-time environments and multimedia support in mind.

Message-based inter-thread communication and a real-time scheduler are provided by RTThreads.[31] Its scheduling policy, a multi-level approach applying EDF within each priority, supports real time requirements, but the problem of priority inversion is not addressed. In contrast to Cool Jazz or OpenThreads, the scheduler cannot be easily modified or replaced by the developer if needed.

6. CONCLUSIONS

We have proposed a platform for multimedia applications based on a message-based threading model and user-defined scheduling. It aims at leaving as much flexibility with the developer as possible while encapsulating necessary mechanisms in the system.

The message-based threading model is more versatile than the conventional purely sequential processing in each thread. The developer can easily model concurrency as it fits best for a particular application. We have shown that inter-thread messages and a reactive style of programming provide a uniform abstraction for controlling concurrency, asynchronous communication and external events. The interface to the operating system is extended by upcalls to efficiently support blocking system calls and asynchronous I/O. Again, the developer has the choice, what type of communication fits best. In this way, our platform represents a suitably flexible infrastructure to cope with the diversity of tasks in a multimedia application.

To take application-specific needs for thread scheduling into account, an interface for a user-defined scheduler is provided. The developer can device scheduling policies tailored for the particular problem while the platform takes care of mechanisms for managing the timing constraints and bounding priority inversion. Scheduler activations enable the interoperability of user-level scheduling with blocking system calls and SMP utilisation.

Currently, we are implementing a video player similar to that mentioned in Section 1 to evaluate the performance of the platform as well as to gain further experience with the programming model and application-specific schedulers. As future work, SMP support needs to be implemented in kernel and user space. Besides several locking issues, it should be investigated whether u-threads should freely be mapped on available processors or whether some locality should be maintained. Moreover, if one k-thread is preempted another one of the application needs to be interrupted to do an upcall and to inform the user-level scheduler. The overhead of these operations is probably too high for very short periods of preemption as frequently caused by some system processes. Here, appropriate heuristics need to be found for an efficient solution.

REFERENCES

1. T. Kramp and R. Koster, "Flexible event-based threading for QoS-supporting middleware," in *Proceedings of the Second International Working Conference on Distributed Applications and Interoperable Systems (DAIS)*, IFIP, July 1999.
2. 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)*, Oct. 1991.
3. I. Leslie, D. McAuley, R. Black, T. Roscoe, P. Barham, D. Evers, R. Fairbairns, and E. Hyden, "The design and implementation of an operating system to support distributed multimedia applications," *IEEE Journal on Selected Areas in Communications* **14**, pp. 1280–1297, Sept. 1996.
4. J. Ousterhout, "Why threads are a bad idea (for most purposes)," 1996. Invited talk given at USENIX Technical Conference, available at `http://www.scriptics.com/people/john.ousterhout/threads.ps`.
5. R. van Renesse, "Goal-oriented programming, or composition using events,or threads considered harmful," in *Proceeding of the 8th ACM SIGOPS European Workshop*, Sept. 1998.
6. ITU, "*Recommendation X.901–X.904, Open Distributed Processing – Reference Model*," 1997.
7. J. Nieh, J. G. Hanko, J. D. Northcutt, and G. A. Wall, "SVR4 UNIX scheduler unacceptable for multimedia applications," in *Proceedings of the Fourth International Workshop on Network and Operating System Support for Digital Audio and Video (NOSSDAV)*, Nov. 1993.
8. R. Govindan and D. P. Anderson, "Scheduling and IPC mechanisms for continuous media," in *Proceedings of 13th ACM Symposium on Operating Systems Principles*, pp. 68–80, Oct. 1991.
9. J. Nieh and M. S. Lam, "The design, implementation and evaluation of SMART: A scheduler for multimedia applications," in *Proceedings of the 16th Symposium on Operating Systems Principles (SOSP-97)*, vol. 31,5 of *Operating Systems Review*, pp. 184–197, ACM Press, (New York), Oct. 1997.

10. D. Steere, A. Goel, J. Gruenberg, D. McNamee, C. Pu, and J. Walpole, "A feedback-driven proportion allocator for real-rate scheduling," in *Proceedings of the Third Symposium on Operating Systems Design and Implementation*, pp. 145–158, Feb. 1999.
11. C. Waldspurger, *Lottery and Stride Scheduling: Flexible Proportional-Share Resource Management*. PhD thesis, Massachusetts Institute of Technology, Sept. 1995. Also appeared as Technical Report MIT/LCS/TR-667.
12. G. Kiczales, R. DeLine, A. Lee, and C. Maeda, "Open implementation analysis and design of substrate software," in *Tutorial Notes, OOPSLA '95*, ACM/SIGPLAN, Oct. 1995.
13. G. Kiczales, J. des Rivieres, and D. G. Bobrow, *The Art of the Metaobject Protocol*, MIT Press, 1991.
14. C. A. Waldspurger and W. E. Weihl, "Lottery scheduling: Flexible proportional-share resource management," in *Proceedings of the First Symposium on Operating Systems Design and Implementation (OSDI)*, Nov. 1994.
15. R. Westbrock, "Flexible scheduling for LINUX," diplom thesis, Department of Computer Science, University of Kaiserslautern, Dec. 1999.
16. J. Walpole, R. Koster, S. Cen, C. Cowan, D. Maier, D. McNamee, C. Pu, D. Steere, and L. Yu, "A player for adaptive mpeg video streaming over the internet," in *Proceedings of the 26th Applied Imagery Pattern Recognition Workshop (AIPR-97)*, SPIE, Oct. 1997.
17. R. Koster, "Design of a real-time communication service for local-area networks," diploma thesis, Department of Computer Science, University of Kaiserslautern, May 1998.
18. T. Kramp and G. Coulson, "The design of a flexible communications framework for next-generation middleware." Accepted at the Second International Symposium on Distributed Objects and Applications (DOA), Sept. 2000.
19. V. Hübsch, *Transaktionsbasierter DSM*. PhD thesis, Department of Computer Science, University of Kaiserslautern, 1998.
20. T. Kramp, "Transaktionsbasierter DSM – Ein Mechanismus zur verteilten fehlertoleranten Programmierung," diploma thesis, Department of Computer Science, University of Kaiserslautern, July 1996.
21. H. Peine and T. Stolpmann, "The architecture of the ARA platform for mobile agents," in *Proceedings of the First International Workshop on Mobile Agents (LNCS 1219)*, Springer, Apr. 1997.
22. L. Sha, R. Rajkumar, and J. P. Lehoczky, "Priority inheritance protocols: An approach to real-time synchronisation," *IEEE Transactions on Computers* , pp. 1175–1185, Sept. 1990.
23. B. O. Gallmeister, *POSIX.4: Programming for the Real World*, O'Reilly & Associates, Jan. 1995.
24. G. Coulson and G. Blair, "Architectural principles and techniques for distributed multimedia applications support in operating systems," *ACM Operating Systems Review* **29**, pp. 17–24, Oct. 1995.
25. R. Gopalakrishnan and G. M. Parulkar, "Efficient user-space protocol implementations with QoS guarantees using real-time upcalls," *IEEE/ACM Transactions on Networking* **6**, pp. 374–388, Aug. 1998.
26. V. Danjean, R. Namyst, and R. D. Russell, "Linux kernel activations to support multithreading," in *Proceedings of the 18th IASTED International Conference on Applied Informatics (AI 2000)*, IASTED, Feb. 2000.
27. V. Danjean, R. Namyst, and R. Russell, "Integrating kernel activations in a multithreaded runtime system on Linux," in *Parallel and Distributed Processing. Proceedings of the 4th Workshop on Runtime Systems for Parallel Programming (RTSPP '00)*, vol. 1800 of *Lecture Notes in Computer Science*, pp. 1160–1167, Springer-Verlag, May 2000.
28. G. Coulson, "A configurable multimedia middleware platform," Tech. Rep. MPG-98-32, Distributed Multimedia Research Group, Lancaster University, 1998.
29. G. Coulson, "A distributed object platform infrastructure for multimedia applications," *Computer Communications* **21**, pp. 802–818, July 1998.
30. M. Haines, "On designing lightweight threads for substrate software," in *Proceedings of the 1997 Annual Technical Conference*, pp. 243–255, USENIX, 1997.
31. D. Finkelstein, N. C. Hutchinson, D. J. Makaroff, R. Mechler, and G. W. Neufeld, "Real-time threads interface," Tech. Rep. TR-95-07, Department of Computer Science, University of British Columbia, Canada, 1995.