Providing Support for Multimedia-Oriented Applications under Linux

Sebastian Schöning[†] **Markus Müller**[‡]
Saarland University, Faculty of Computer Science, Operating System Research
Chair of Professor Scheidig Building E1.1, P.O. Box 151150, D-66041 Saarbrücken
[†]schoening@cs.uni-saarland.de [‡]muemar@studcs.uni-saarland.de

October 24, 2007

Abstract

Linux as a general-purpose operating system is designed to support the services required by most generic applications. Today, there is a new generation of real-time applications that are interactive, computation-intensive, and I/O-bound. In the past years, operating system research has addressed each of these challenges in turn. Yet frequently, when an application requires the support of all three aspects in unison, performance suffers. In particular, this is the case for multimedia-oriented applications. These applications have to meet soft real-time requirements, demand for a high data throughput, and need to interact with the user. Furthermore, they are constructed in a highly modular fashion with a multitude of concurrent and interdependent activities.

The software market offers either dedicated multimedia applications or middleware solutions, each addressing these issues at a high level: The solutions are built upon existing operating system interfaces and they rather focus on the support for specific multimedia functionalities, such as decoders, instead of providing an appropriate operation infrastructure for a wider range of applications within that area.

This paper targets the integration of multimedia-oriented applications from a new angle. We improve the support for such applications by means of dedicated system services: Instead of the common process model, our execution paradigm is based on the process network model. This model is an established formalism in the field of embedded systems. We leverage the theoretic findings by implementing them into a real operating system. Additionally, we contribute dedicated buffer management and scheduling facilities by exploiting the advantages of the process network model. The implementation is realized both as a kernel-space and a user-space solution. For the kernel-space solution, we extend the Linux kernel and its system-call interface to use the full potential of kernel-internal features.

1 Introduction

The advancements of computer hardware have paved the way for an ever-increasing demand on multimedia-oriented applications on desktop systems. Usually, such applications are executed on general-purpose operating systems along with conventional interactive applications. Multimedia-oriented applications make heavy demands on resources, require soft real-time guarantees, and the support for user-interaction, which is intrinsically non-deterministic. These applications are highly modular and use specialized function libraries. Furthermore, such applications embody a multitude of concurrent and interdependent activities that are realized in software as well as in specialized hardware.

Figure 1 depicts such a situation. A simple audio-video player stamps text messages into the video output. The nodes in the graph represent individual actions, such as grabbing, splitting, and filtering. The directed edges between nodes show the data dependencies between these actions. For instance, a frame of an audio-video stream is first grabbed (*grabber*), then splitted into separate audio and video frames (*splitter*), and the result is next processed by audio and video decoders (*decoder*). Subtitles are stamped into the decoded video frame (*combiner*), and, eventually, audio and video data are output to the appropriate devices (*buffer*). Subtitles are fetched from an external source that may, for example deliver textual input data, like stock exchange notes. The output stages (*audio buffer* and *video*

buffer) are time-driven by means of some play-out period, such as the frame rate of the display. The processing functions are taken from multimedia libraries, such as the `ffmpeg` library [8] that is part of the MPlayer project [14].

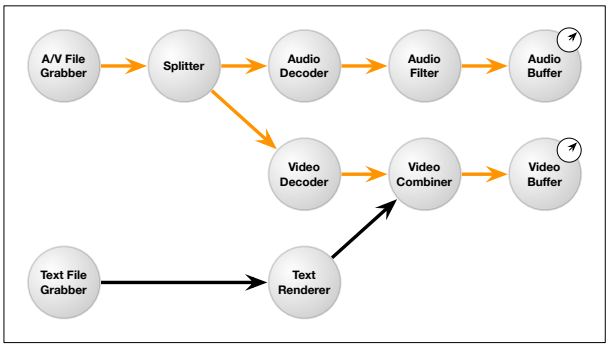

FIGURE 1: *Data flow in an audio-video player with subtitle overlaying.*

1.1 Our Approach

We claim that the combined execution of interactive and multimedia-oriented processing is not sufficiently supported in general-purpose operating systems (GPOS), like Linux. First, such operating systems provide means for generic classes of applications, which are not necessarily adequate for multimedia-oriented applications. Second, the scheduling on general-purpose operating systems was designed to provide a high degree of fairness between competing activities while maintaining a high utilization of the overall system; real-time activities are feasible but these activities infringe the fairness criterion. As a consequence, in systems with a mixed application set either interactive applications are subject to starvation or multimedia-oriented applications cannot meet their quality-of-service requirements.

Our novel approach to this problem extends an off-the-shelf general-purpose operating system to support multimedia-oriented applications by means of the following aspects:

- adoption of a process network model as the execution model,
- provision of a buffer abstraction that allows for a time-driven representation of memory objects,
- implementation of a methodology for application-specific scheduling algorithms based on finite automata, and
- virtualization and dedication of hardware resources to resolve critical concurrent accesses.

We call the implementation of our approach the *Component Extension* (CE). We provide two implementation variants: a kernel-space extension for the demonstration of the feasibility and efficiency of our approach and a user-space variant for comparison with conventional means. In this paper, we describe the concepts underlying our approach and provides a status quo on the development of the kernel-space extension.

1.2 Document Structure

The remainder of this document is structured as follows: Section 2 covers conceptual details of our approach. In Section 3, implementation aspects are introduced and discussed. In Section 4, we discuss the related work. Section 5 concludes the document with summarizing remarks and some words about current and future work.

2 Concepts

This section introduces the concepts underlying the Component Extension in a top-down fashion. We first motivate the use of a different execution model. Then, we introduce our notion of a multimedia-oriented application, which is called application scenario. Subsequently, we describe the details: components (execution primitive), channels (communication means), buffers (memory objects), and the application scheduling. We conclude the section with discussing the problem of concurrent device accesses with respect to timeliness and our approach to its remedy.

2.1 Execution Model

General-purpose operating systems provide efficient means for the execution of generic classes of applications, such as interactive applications. The general notion of activity is based on the process concept [20]. Processes incorporate internal parallelism by threads. These operating systems provide means to coordinate sets of processes and threads, yet the overall structure of applications remains hidden from the operating system itself: processes and threads are organized in a “flat” manner (meaning as unstructured pools) and the operating system is only involved in considering dependencies among them if the need for exchanging data or synchronization evolves.

Multimedia-oriented applications execute a multitude of concurrent and interdependent activities. Consequently, their internal structure is frequently complex. Therefore, the process model doesn’t seem

to be sufficient for such an application class because structural knowledge that already exists is not exploited by the operating system. Therefore, we argue for an adoption of the process network model [10] as the basis for the representation of multimedia-oriented applications.

2.1.1 Application Scenario

We define an *application scenario*, which represents a multimedia application, as a directed acyclic graph with *components* as nodes and *channels* as edges. Components accommodate single functions that are executed sequentially. These functions take and provide parameters of fixed sizes and amounts. Function parameters (such as video frames) are encapsulated into *buffers* that are communicated via unidirectional first-in, first-out channels between components. The transmission semantics can be defined as "block on empty input and unblock on output": the execution of a component is blocked if it reads from an empty input channel; a write operation to an output channel may unblock subsequent components.

The application scenario can be partitioned into disjoint *protection domains* that correspond to different address spaces. The buffer handling facility of the CE manages the allocation of memory for protection domains and transmission between protection domains. Hence, conventional protections mechanisms can also be utilized for multimedia-oriented applications.

The discrimination between components versus processes on the one hand and channels versus specific communication means on the other hand gives rise to a transparent mapping between the application layer and the operating system layer. Depending on the implementation, components can be mapped to processes, threads, or to any other type of activity that is provided by the system. Even a hierarchical mapping of components to activities of varying weights is conceivable, e.g. components are mapped to coroutines, linear sequences of components are mapped to threads, and protection domains are mapped to processes. Similarly, channels can be mapped to an appropriate communication means, like the shared-memory facility, or even be implemented anew.

2.1.2 Components

Components are characterized statically prior to compile time by several parameters: *Consumption scores* are assigned to input channels and *production scores* are assigned to output channels. Input scores specify for each input channel the amount of input data that is needed in order to produce the amount output data that is specified by the production scores. Channels can be indicated as *optional*, which is used for components that consume or produce interactive data that may or may not be available prior to or after component execution.

Source or sink components are characterized either as *active* or *passive*. Active components are activated externally, like a service routine that act on behalf of an interrupt handler. Passive components have to be activated by the CE, like a component that grabs data from a file.

Application scenarios are specified prior to runtime by the following attribute, which characterizes the runtime behavior of an application scenario. For each component, a tuple (L, P, F) of positive real numbers is given, with L specifying the *lead time*, with P specifying the *period*, and with F specifying the *follow-up time*. Periods represent time intervals for the periodic activation of components; the lead time represent the maximum time shift for an early activation and the follow-up time represent the maximum time shift for the finishing. All periods of an application scenario have to correspond with respect to the consumption and production scores of all components.

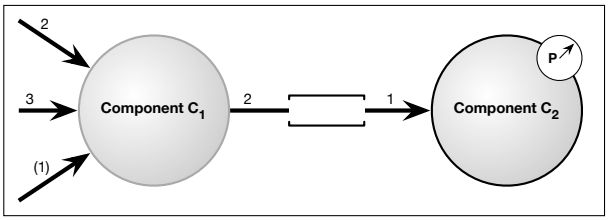

FIGURE 2: *Fragment of an application scenario consisting of a passive filter component (C_1) and an active sink component (C_2).*

Figure 2 depicts a fragment of an application scenario that consists of two components. Component C_1 embodies a filter function that requires three input channels and a single output channel. The input channels have consumption scores 2, 3, and 1, the output channel has a production score 2. The third input channel of component C_1 is declared optional. Component C_2 contains an active sink function, i.e. some output device, that is activated periodically with period p.

In Figure 3, the application scenario is shown that corresponds to the example application from Figure 1. Production and consumption scores that equal 1 are omitted. Component C_4 consumes 3 data units in order to produce 3 data units. Components C_5 and C_8 are active sink components that are activated externally with periods P_5 and P_8. Component C_7 has an optional input channel that delivers interactive data.

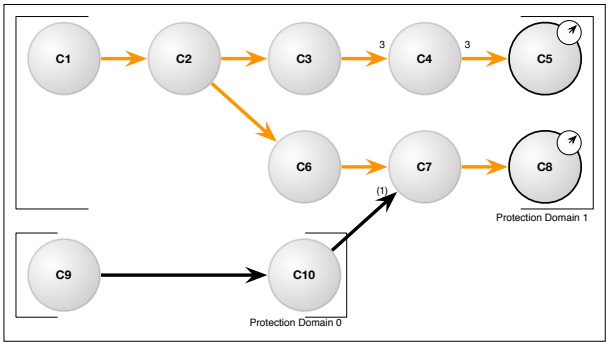

FIGURE 3: *The application scenario that corresponds to the example in Figure 1.*

2.1.3 Signal Handling

In order to handle events, such as an end-of-stream event, we adopt the POSIX signal handling for the Component Extension [20]. Our approach differs in the following aspects: setup and delivery.

For any given component, *signals* and their respective *signal handler* functions are set-up prior to component compile time and may not be changed later on. A number of application-specific signals can be defined. A set of default signals is required to be defined by the CE, such as the `abort scenario` signal.

The signal delivery and the component execution are synchronized. Whenever a component is about to be executed, i.e. sufficient data has arrived, the signal handling is executed first, if signals are *pending*. Then, the component control flow is continued where it was stopped during the last execution. Hence, signals can be used to encode data-stream markers, such as the end-of-stream marker. The misuse of net data for that purpose is avoided and the component code for data processing and stream control is strictly separated.

Signals can be delivered in broadcast mode or in unicast mode. In broadcast mode, a signal is delivered to every component of the application scenario, regardless of their connection to the sending component. For the time being, the broadcast mode is reserved for the set of default signals. In unicast mode, signals are delivered with respect to the topology of the scenario graph, i.e. from component to component.

The application scenario defines for each component which signals that are masked or unmasked for reception, i.e. the delivery is disabled or enabled. By default, all signals are masked except for unmaskable signals that are required by the Component Extension.

2.2 Buffer Handling

General-purpose operating systems provide facilities for memory management and communication that can be considered simple and efficient. Yet for multimedia-oriented applications, these means lack important features due to the following reasons:

1. Multimedia-oriented applications process time-based data, which cannot be represented properly in conventional memory objects.

2. Usually, multimedia data has vast extents and changes frequently. Hence, needless copying or caching should be avoided. Conventional communication is directed towards the transmission of small data quantities, for which copying and caching is of minor consequence. However, substantial overhead is introduced for the transmission of multimedia data. Shared-memory communication can be considered the only exception, yet the available means support a coarse granularity only and the management of shared-memory regions is completely delegated to the application.

3. Neither facility takes the internal structure of multimedia-oriented applications into account. For example, conventional memory allocation schemes neglect processing paths that data buffers undergo.

2.2.1 Buffer Abstraction

These issues have been addressed in the literature in different contexts [2, 7, 12]. We adapt these approaches to our needs: we introduce *buffers* as the sole abstraction of memory objects that is available to components. We combine the management of memory objects and their communication based on the notion of buffers.

2.2.2 Buffer Operations

A set of operations is defined on buffers. Buffers can be acquired (`bacquire()`) with respect to the channel that will be used to transmit the buffer or disposed (`bdispose()`). Buffers that are not subject to any communication can locally be allocated (`balloc()`) and freed (`bfree()`).

Buffers can be copied (`bcopy()`) such that a copy-on-demand mechanism is applied whenever a buffer copy is overwritten. Based on this virtual-copying mechanism, a buffer splitting operation (`bsplit()`) and a buffer joining operation (`bjoin()`) are defined.

Finally, buffers are transmitted by send and receive operations (`bsend()`, `breceive()`). The send

operation removes a buffer from the context of the current component and reposits the buffer in the output channel it was sent to. The receive operation introduces a buffer into a component context. The receive operation may block on empty input channels, the send operation does not block.

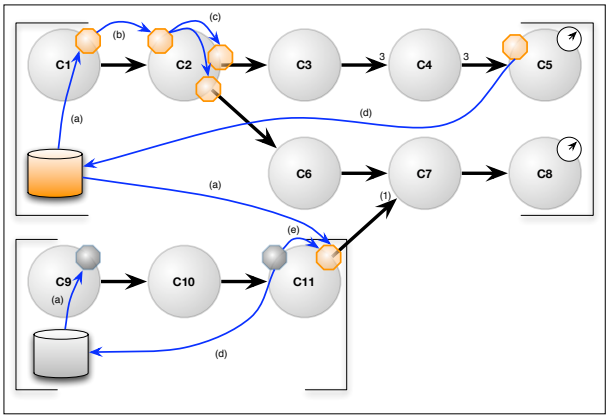

FIGURE 4: *Buffer operations that may occur during the execution of the example application scenario: (a)* `balloc()`*, (b)* `bsend()` *and* `breceive()`*, (c)* `bcopy()`*, (d)* `bdispose()`*, and (e) copying and disposal for a buffer that crosses protection boundaries.*

2.2.3 Buffer Parameters

During their life time, buffers are equipped with additional informations. Upon creation, buffers are assigned a *creation time* t_c.

A tuple (c, d, e) of positive real numbers with $0 < c < d \leq e$ and $t_c \leq c$ is assigned to a buffer. The parameter c denotes the *commencement* time, d denotes the *deadline*, and e denotes the *expiry time*. The commencement time represents the earliest instant the buffer or its descendent is to be delivered at the closest final sink, the deadline represents the latest instant the buffer is to be delivered, and the expiry time represents the instant the buffer completely loses its value for any further computation. The tuple is derived from static scenario parameters and can be constrained by components during runtime.

A stream of multimedia data may contain data of varying importance. For example, an MPEG video stream contains so-called intra-frames (I-frames), predicted frames (P-frames) and bi-predicted frames (B-frames) [19]. I-frames constitute key frames that should not be lost, whereas the loss of P-frames or B-frames entails minor consequences only. Buffers are assigned a *persistence value* v from the natural numbers that represents the importance of that buffer. A persistence value $v = 0$ specifies that the buffer may not be dropped and persistence values $v > 0$ specify that the buffer can be dropped if the need occurs.

For example, interactive or hard real-time buffers are assigned a persistence value $v = 0$, i.e. these buffers will not be dropped at all. Contrariwise, multimedia buffers can have assigned $v > 0$ values. A multimedia buffer with $v_t = k$ and $k > 0$ that is dropped at time t leads to the disposal of any subsequent buffer with persistence value $v_{t'} > k$ at any later time $(t' > t)$.

2.2.4 Buffer Pooling

Since the application structure is statically defined, memory requirements can be estimated prior to scenario runtime. Then, buffer memory is allocated to *buffer pools* that are assigned to processing paths with respect to protection domains. During the scenario execution, buffers are acquired from or disposed to the respective buffer pool. Thereby, costly memory management operations (like expensive page-table manipulations) can be avoided.

2.2.5 Example

In Figure 4, buffer operations are shown that can occur during the execution of the example application scenario (cf. Figure 3). First, buffers are acquired from the respective buffer pools (a). Then, buffers are processed and communicated (b). Buffers are copied (c) and, eventually, disposed (d), which returns buffers into their pools. Since the scenario consists of two protection domains, buffers are copied for cross-domain communication (e).

2.3 Scheduling

The advent of parallel processors on the desktop computer market fosters the emergence of multimedia applications with increasing complexity and processing requirements. The operating system faces two potentially opposing challenges: the utilization of available hardware parallelism and processing power versus the accommodation of all application requirements (real-time versus best effort). The system scheduler is assigned to fulfill both challenges at once as effective as possible. Usually, for dynamic activity sets a generic scheduling approach is applied. The set is disjoint into activity classes that are subject to different scheduling strategies, such as the static first-in, first-out strategy or the dynamic best-effort strategy [3, 20].

In contrast to generic scheduling, we adopt an application-dedicated scheduling approach. In our context, the application structure and relevant application parameters are known prior to runtime. In

combination with information about present hardware resources, a dedicated scheduling is realized on top of the scheduler that is provided by the system.

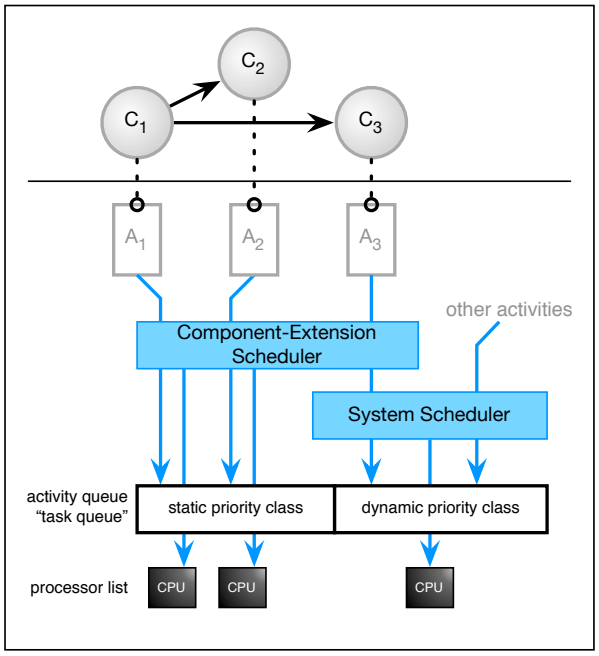

FIGURE 5: *Integration of the CE scheduling into Linux. Components are associated with system activities like threads. The outcome of CE scheduling is delegated to the system scheduling.*

2.3.1 System Integration

The integration of the CE scheduler into a general-purpose operating system, like Linux, is depicted in Figure 5. The components of an application scenario are associated with appropriate system activities (e.g. threads). The CE scheduler assigns multimedia activities to a static priority class (e.g. the FIFO class) and interactive activities to a dynamic activity class (like the best-effort class). Static priority classes are not subject to reordering by the system scheduler and are preferred to dynamic priority classes [3]. Hence, the priority assignments for multimedia components, given by the CE scheduler, are enforced by default. Furthermore, the CE scheduler sets component-to-processor affinities that are respected by the system scheduler. Interactive components share their processing time with other best-effort applications. The CE scheduler implements constraints on the processor utilization of multimedia components to prevent interactive activities from starvation.

2.4 Application-Dedicated Scheduling

Our approach to an application-dedicated scheduling is realized by means of timed automata. A timed automaton is a finite automaton that is augmented with *clocks* assigned to states [4]. Time is only allowed to pass in states as long as clock *invariants* hold, which are imposed on states.

Transitions, which lead to state changes, are *guarded* by clock constraints and are labeled by *actions*. Transitions may induce resets on certain clocks.

Actions are differentiated into *full actions* and *half actions*. Full actions can be executed immediately, whereas half actions need to be paired in order to compose a full action. Half actions represent communication events, which are indicated by the symbols "?" for inquiring half actions and "!" for asserting half actions.

Parallelism is expressed by networks of timed automata. Timed automata are a common model that is used in the field of embedded systems to model and verify hard real-time systems [4].

We use a network of timed automata that is called the *scheduler automaton* to represent the overall state of the application scenario. Any automaton of the network either corresponds to a component, to a hardware resource (i.e. a processor), or to the entire application scenario. Automata that represent hardware resources are used to model resource allocations. Automata that represent the application scenario are used to implement global constraints.

Transitions can be taken explicitly or implicitly. An *explicit transition* is taken whenever a running component issues a system call that is related to the CE. An *implicit transition* is taken whenever a clock invariant is violated and the CE has induce a scheduling decision.

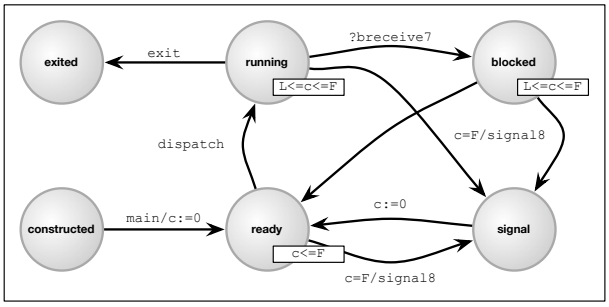

FIGURE 6: *Simplified timed automaton that models the component behavior of component C_8 from the example application scenario.*

In Figure 6, a timed automaton is depicted that represents component C_8 from the example application

scenario (cf. Figure 3). Upon the entry of the main function of component C_8, a transition from state `constructed` to state `ready` is made and the clock c is set to zero. The clock invariant for state `ready` ascertains that time never elapses over the follow-up time F that was defined the component. If clock c equals the follow-up time, a implicit transition is triggered, a signal is issued, and the clock is reset.

When component C_8 is allocated to a processor, transition `?dispatch0` is taken. By definition, a transition `!dispatch0` has to be active in any other automaton (i.e. it can be taken) before component C_8 can reach state `running`. Whenever the component issues a `breceive()` system call and no input data is available, transition `breceive7` to state `blocked` is taken. As soon as a transition in component C_7 becomes active that is labeled `!bsend7`, component C_8 can transition into state `ready`.

A scheduling facility is established that executes the scheduler automaton during the runtime of the application. Scheduling decision are made, whenever several transitions become active. The scheduling facility provides all available timing information about the application scenario to the scheduling automaton, which in turn facilitates this information with respect the scheduling strategy that the automaton models.

The scheduler automaton is provided externally as part of the application. Thus, we enable the application developer to tailor the scheduling strategy for the application in question. Additionally, we provide a way to construct a heuristic scheduling automaton that delivers sufficient results in the general case, which is not discussed here.

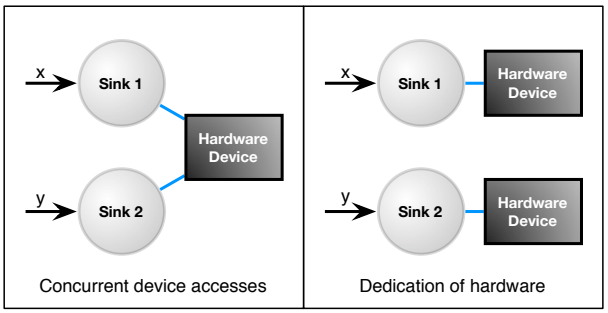

FIGURE 7: *Illustration of the problem of concurrent device accesses by two independent components that reflect on the same physical hardware (left hand-side). The problem can be solved by resource dedication (right hand-side).*

2.5 Dedication & Virtualization

General-purpose operating systems do not provide means for scheduling of accesses to hardware devices, e.g. some physical input/output device. Instead, concurrent accesses to a single hardware resource are ordered indirectly according to the outcome of the process scheduling, unless the resource itself doesn't establish some access reordering. Neither the fairness for interactive applications nor the quality-of-service for multimedia-oriented applications can be guaranteed in general if concurrent accesses of both kinds of applications occur. The dedication of hardware resources solves that problem, yet this solution entails additional hardware costs.

Some hardware resources cannot be dedicated since multimedia-oriented and interactive applications need to access these resources concurrently. For example, a frame-buffer device that is connected to some video adapter is used to visualize application output, which is generated by both kinds of applications. For such so-called *critical resources* another solution has to be found.

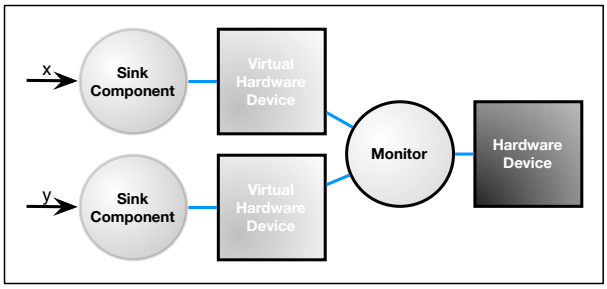

FIGURE 8: *Virtualization solves the problem of concurrent devices accesses. Components access virtual instances of the physical device. A monitoring instance controls the physical device and enforces a static access ordering similarly to static priority schemes used for processor scheduling.*

Consequentially, we devise a virtualization approach for critical resources: virtual resources are dedicated to applications instead of physical resources [15]. The control of the physical resource is maintained by some coordinating instance, the so-called *monitor*, which provides the virtual resources and which can also establish some scheduling of resource accesses. Each critical resource has to be virtualized individually. The virtualization solves the concurrency problem, yet it does not come without additional costs (processing time and memory), which may account negatively for the overall system performance.

3 Implementation

The implementation of the Component Extension is provided in two variants: as a kernel extension and as a user-space library. The kernel extension facilitates

the full potential of kernel capabilities, whereas the user-space library bases solely on conventional user-space means. The user-space library serves as a base for comparison, since scenarios based on that implementation can be regarded as conventional multimedia-oriented applications. Both implementations are based on a standard Linux kernel.

The subsequent section gives an overview of the implementation of the kernel extension at the present moment.

3.1 Kernel Extension

The kernel extension provides Component Extension services by means of additional Linux system calls. For the time being, system calls are defined for registration (`register_scenario()`), setup (`setup_scenario()`) and start (`start_scenario()`) of scenarios as well as communication between components using user-space buffers (`bacquire()`, `bdispose()`, `bsend()`, `breceive()`).

The extension comes as a kernel patch and a set of kernel modules. The patch extends the system call interface and exports kernel symbols which are not exported in the default kernel code but needed by the kernel modules.

The kernel modules contain the actual implementation of the Component Extension. These modules register kernel functions that relate to Component Extension system calls. System calls are forwarded to these functions, whenever they occur. Currently, four modules are defined:

- the module `ce_core` defines code and data structures that are needed for registration and setup of scenarios,
- the module `ce_mem` contains code and data structures needed for the buffer management,
- the module `ce_com` contains code for the communication of buffers,
- the module `ce_sched` contains the scheduling framework and the default scheduler.

The Linux kernel extension of the CE is based on kernel version 2.6.22.

3.1.1 Kernel Integration

The kernel integration is depicted in Figure 9. Service requests reach the kernel extension via system calls and are first handled by the scheduling framework. It decides whether the request can be fulfilled immediately (e.g. by the communication- and buffer-management) or if rescheduling is needed. The scheduling framework and the actual scheduler use the information in the scenario description and scheduler automaton. In addition, kernel timers inform the scheduling framework about the begin of new periods of components.

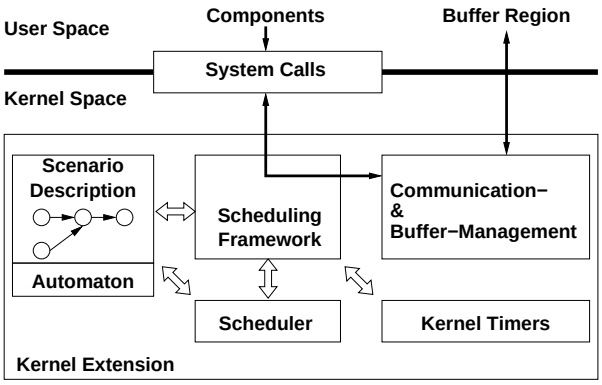

FIGURE 9: *A schematic view on the internal structure of the kernel extension.*

3.1.2 Scenario Setup

Similar to the Linux kernel, which provides the `task_struct` data structure to describe schedulable objects, the kernel extension provides data structures for scenarios (`scenario_struct`), components (`component_struct`) and channels (`channel_struct`). The `component_struct` data structure is of particular interest since it contains information about the mapping of components to Linux task, i.e. it contains a pointer to the corresponding `task_struct`.

The setup of a scenario is done in two steps: First, the application scenario must be *registered*. Thereby, the kernel assigns an unique identifier to the scenario and each component and channel, which is similar to the process identifier (PID) used for Linux tasks and POSIX processes [20].

In the second step the kernel creates a new Linux task for each component in the scenario and ensures that all of these tasks share the same virtual address space. As with POSIX threads, each component gets its own private stack segment.

3.1.3 Buffer Handling

Buffers are blocks of contiguous virtual memory in user space. They are located in an additional memory segment (called *region* in the kernel terminology) and are accessible to the user (cf. Figure 10). Due to the introduction of a new memory segment, buffers cannot interfere with other user-space memory. Furthermore, this region is marked as locked such that memory from that region cannot be swapped out.

To acquire a buffer (cf. Section 2.2.2), a component must issue the appropriate system call to re-

ceive a pointer to a free memory area in the user-space buffer memory area. Further communication between components is then accomplished by additional system calls that simply exchange pointers (e.g. the operations `breceive()` and `bsend()`). The kernel extension has to keep track of which buffers of a scenario are used and which are currently free.

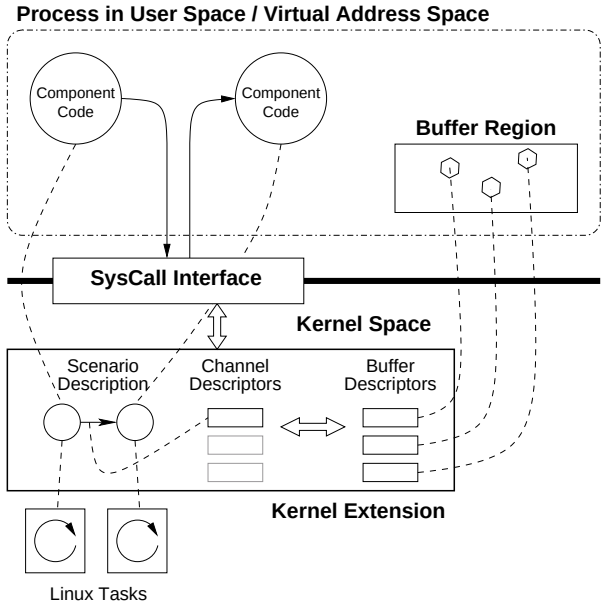

FIGURE 10: *Mapping of application scenarios into kernel objects. Components of a scenario are mapped to Linux tasks that share the same virtual address space. Exchange of data between logically connected components is achieved by system calls that take or return pointers to buffers in user space.*

3.1.4 Scheduling

The scheduler is integrated into the general scheduling subsystem of the Linux kernel. Multimedia components are assigned to the `SCHED_FIFO` static priority class while interactive components are assigned to the general dynamic priority class `SCHED_OTHER`. Since static priority tasks are always preferred to normal dynamic priority tasks it is ensured that multimedia components are executed with the highest priority.

In Linux, each processor in the system has its own *runqueues* for all available scheduling classes. Thus, enforcing the requested processor affinity of components can easily be accomplished by assigning a task to the appropriate FIFO runqueue of the target processor. The bit mask `cpus_allowed` in the `task_struct` data structure is set such that each task is executable only on the processor(s) that it was assigned to. Hence, task migration between processors is avoided, which would otherwise be conducted by the runqueue-balancing mechanism on behalf of the kernel scheduling.

Since the kernel extension has information about the structure of the scenario and is provided a scheduler automaton, the scheduler can avoid unnecessary blocking of components and can meet real time requirements more easily.

Periods are enforced by use of kernel timers. Each time a new period of one of the components starts, the scheduler is activated and gets the chance to reassign runnable components to the CPUs in the system.

3.1.5 Summary

The migration of the Component Extension into kernel space allows us to exhaust the full potential of kernel-internal features. Even though we discussed the use of conventional kernel means, like Linux tasks as execution primitive, we point out that our approach can be augmented to accommodate other Linux extensions that provide improved real-time capabilities, such as the Real-Time Application Interface [6].

The kernel-based implementation of the Component Extension is still in an ongoing project. Among our first, very promising results, we accomplished the integration of essential primitives into the Linux kernel in order to support simple application scenarios. We plan to provide a first fully-functional version by the end of this year.

4 Related Work

The process network model is common for modeling embedded system applications. Since the original work of Kahn [10], several variants have been devised, such as synchronous data-flow networks [11], which are the basis for the work described in this paper.

In the field of networked systems, a stream-oriented buffer management was first introduced by Druschel [7] to facilitate an efficient data-transfer mechanism between protection domains for high throughputs. In field of operating system research, early works of Ritchie [16] addressed the optimization of data routing and processing through the network stack of UNIX operating systems. This work was adopted for Linux in the LiS project [2] and the KStreams project [12]. All these works are rather related to network issues, even though the findings are also relevant to a wider scope.

Several multimedia frameworks and multimedia middleware variants are available that are based on

data-flow representations. Among the most prominent are the Network Multimedia Middleware [13] and the GStreamer framework [9]. Both projects focus on the provision of ways to easily enable programmers to develop codecs, filters or other components for local or distributed applications. Hence, binding mechanisms, QoS arbitration and adaption, and portability issues are profoundly addressed in these works, yet operating system issues, which are discussed in this paper, are only narrowly considered.

In the field of formal verification, timed automata are an established formalism to model software and hardware systems [4]. Among others, Altisen, et. al, [1] and Combaz, et. al., [5] proposed methodologies for the combination of system modeling and system scheduling.

The separation of multimedia and interactive parts of a computer system is a central idea of the TwinUx project [17]. Hardware resources, such as framebuffer devices, are virtualized and then dedicated to either interactive or multimedia-oriented applications. Monitors control physical devices and solve critical concurrent accesses to hardware devices. The feasibility of the concept was shown by Schöning [17] and Simon [18]; Rauch [15] realized the virtualization of memory-based devices, such as framebuffers.

5 Conclusion and Status Quo

In this publication, we give an overview of the concepts that form the basis for the Component Extension. Furthermore, the status quo of the implementation is described. Currently, a user-level implementation of the Component Extension is available, which allows for the execution of applications scenarios, as discussed above. For that version, both the user-space implementation of CE concepts and the adaption of conventional means are provided. The ongoing work encompasses the integration of the Component Extension into a standard Linux kernel to facilitate the full potential of kernel-internal features. Benchmarks will be provided as soon as the kernel-based version of the Component Extension is completed.

References

[1] Karine Altisen, Gregor Gößler, Amir Pnueli, Joseph Sifakis, Stavros Tripakis, and Sergio Yovine. A Framework for Scheduler Synthesis. In *IEEE Real-Time Systems Symposium*, volume 20, pages 154–163, 1999.

[2] Fransisco Ballesteros, Dennis Froschauer, and David Grothe. The Cutting Edge: LiS: Linux STREAMS. *Linux Journal*, (61):14, 1999.

[3] Daniel P. Bovet and Marco Cesati. *Understanding the LINUX Kernel*. O'Reilly Media, 3rd edition, November 2005.

[4] Howard Bowman and Rodolfo Gomez. *Concurrency Theory: Calculi an Automata for Modelling Untimed and Timed Concurrent Systems*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2005.

[5] Jacques Combaz, Jean-Claude Fernandez, Thierry Lepley, and Joseph Sifakis. QoS Control for Optimality and Safety. In *EMSOFT'05: Proceedings of the 5th ACM International Conference on Embedded Software*, pages 90–99, New York, NY, USA, 2005. ACM Press.

[6] DIAPM. RTAI—the Real-Time Application Interface for Linux. `https://www.rtai.org/`.

[7] Peter Druschel. Operating System Support for High-Speed Communication. *Communications of the ACM*, 39(9):41–51, September 1996.

[8] FFMPEG Codec Library. `http://ffmpeg.mplayerhq.hu/`.

[9] GStreamer Open-Source Multimedia Framework. `gstreamer.freedesktop.org`.

[10] Gilles Kahn. The Semantics of Simple Language for Parallel Programming. In *IFIP Congress*, pages 471–475, 1974.

[11] Bart Kienhuis and Ed F. Deprettere. Modeling Stream-Based Applications Using the SBF Model of Computation. *J. VLSI Signal Processing Systems*, 34(3):291–300, 2003.

[12] Jiantao Kong and Karsten Schwan. Kstreams: Kernel Support for Efficient Data Streaming in Proxy Servers. In *NOSSDAV '05: Proceedings of the International Workshop on Network and Operating Systems Support for Digital Audio and Video*, pages 159–164, New York, NY, USA, 2005. ACM Press.

[13] Marco Lohse. *Network-Integrated Multimedia Middleware, Services, and Applications*. PhD thesis, Department of Computer Science, Saarland University, Germany, June 2005.

[14] MPlayer. `http://www.mplayerhq.hu`.

[15] Bernd Rauch. Virtualization of Memory-based Devices. Diploma Thesis, Saarland University, June 2006.

[16] Dennis M. Ritchie. A Stream Input-Output System. Technical report, AT&T Bell Laboratories, 1984.

[17] Sebastian Schöning. TwinUx@SB – eine Plattform zur Integration multimedialer und interaktiver Verarbeitung. Diploma thesis, Saarland University, 2003.

[18] Jens Simon. TwinUx@SB – Eine experimentelle Implementierung von Koordinator und Multimedia-Anteil. Diploma Thesis, Saarland University, 2004.

[19] Ralf Steinmetz. *Multimedia-Technologie – Grundlagen, Komponenten und Systeme*. Springer-Verlag, 2nd edition, 1999.

[20] The OpenGroup. *Single UNIX Specification Version 3*, 2004.