FLEXIBLE USER/KERNEL COMMUNICATION FOR REAL-TIME APPLICATIONS IN ELINUX

Christian Poellabauer, Karsten Schwan, Richard West, Ivan Ganev, Neil Bright, Gregory Losik
College of Computing
Georgia Institute of Technology
Atlanta, GA 30332
{chris,schwan,west,ganev,ncb,gregor}@cc.gatech.edu

Abstract

Many distributed applications require real-time and quality of service guarantees, which need to be observed even under changing resource requirements and resource availability. Our group is developing a version of the Linux kernel, termed ELinux, that offers mechanisms to deal with dynamic resource changes and to exploit and adjust to runtime changes in user needs, with the goal of leveraging both in order to improve the effective quality of service experienced by end users. Our approach is to construct runtime extensions for the Linux kernel that offer suitable functionality for quality of service management for real-time applications. The resulting ELinux kernel described in this paper offers an innovative mechanism, termed ECalls, that permits even unprivileged users (without superuser privileges) to add or modify the services provided by the operating system kernel. ECalls also permits such user-defined extensions to interoperate with user-level applications, such that the timing requirements of service- and application-level functions are met. Specifically, ECalls permits users to add application-specific event handlers to the kernel, and it implements lightweight synchronous or asynchronous user/kernel communications for use by those handlers. ECalls (Event Calls) addresses several issues in the user/kernel communication occurring for real-time extensions: (1) using multiple methods of control transfer exhibiting alternative real-time behaviors, (2) integrating the scheduling of tasks and of events, (3) using shared memory structures to avoid explicit data passing between kernel services and application, and (4) enabling reductions in the frequency of control transfers with event-specific call filters and delays, and by event batching.

1 Introduction

Dynamic changes in resource requirements and availability make real-time and quality of service guarantees for distributed applications difficult. Our group is developing technologies that are able to deal with dynamic resource changes and to exploit and adjust to runtime changes in user needs. Specifically, we are developing runtime configuration methods to be applied to open source operating systems suitable for embedded platforms, with our current work focused on the Linux OS kernel. Our group's aim is to continuously manage user needs and the platform resources applied to these needs, without rebooting the underlying systems. Consequently, the configuration methods we are creating are themselves represented as dynamically loaded code modules. The resulting *ELinux* (Extensible Linux) system permits applications to extend the operating system kernel with precisely the quality management features they require. For instance, a multimedia streaming application may extend the kernel with functions that efficiently monitor available network bandwidth, so that it may then change compression levels and its manner of packet delivery to adjust its operation to resource availability.

To exploit the advantages of kernel-based services, a flexible interface between applications and kernel services is required, such that service requests and responses can take place in a timely manner. The main part of this paper introduces such an interface implemented for the ELinux system, enabling applications to choose dynamically from different methods of event delivery from user to kernel and kernel to user.

The remainder of this paper is organized as follows: The next chapter introduces ELinux, Georgia Tech's implementation of a QoS-aware version of the Linux operating system. ELinux is an ongoing effort, where some of the modifications and enhancements introduced in Chapter 2 are finished and others are still being implemented. Chapter 3 introduces *ECalls*, a flexible interface between user and kernel space allowing applications to efficiently use the services provided by kernel extensions. Chapter 4 introduces some of the mechanisms implemented in ECalls in further detail, and Chapter 5 concludes and summarizes this paper.

2 ELinux Overview

ELinux is our group's effort to build a version of Linux able to deal with dynamic resource changes and able to exploit and adjust to runtime changes in user needs. ELinux not only offers basic mechanisms for runtime system extension, but it also supports several extensions specifically targeting the real-time domain. These extensions are implemented as kernel-loadable modules.

Real-Time CPU Scheduling. One of the extensions offered by ELinux is a hard real-time scheduler that replaces the standard Linux scheduler. DWCS (Dynamic Window-Constrained Scheduler) [11, 12] schedules processes according to their service constraints: a *request period* T and a *window-constraint* x/y. The request period is the maximum tolerable interval between servicing two requests of a process, where the end of a request period determines a *deadline* by which a process must be serviced. The numerator x of the window-constraint is the number of times a request for a process can be serviced later than the deadline or not serviced at all for every y requests. Given a set of processes, DWCS is able to bound the delay of service to a process, even in overload scenarios (when processor utilization is larger than 100%). Furthermore, feasible schedules exist as long as the utilization does not exceed 100%. Details about the real-time capabilities of DWCS can be found in [9].

Real-Time Network Packet Scheduling. DWCS is also capable of scheduling network packets, where the period T is the interval between two service requests to a packet stream, and the numerator x of the window-constraint x/y is the number of packets that can be transmitted late or even lost for every window of y packets in a stream. The same real-time guarantees as described above apply. Consequently, the packet scheduling extension of ELinux permits applications to control their outgoing packet streams, which is particularly useful for server-based applications, such as media and web servers.
QoS-aware Implementation of Sockets. The DWCS-based packet scheduling supported by ELinux does not offer the 'convenient' interfaces end users may require for controlling their servers' output streams. The 'QSocket' interface implemented in ELinux uses the Linux kernel's recently developed framework for the implementation of QoS technologies, the latter supporting a set of queuing disciplines that include Class Based Queuing (CBQ), Stochastic Fair Queuing (SFQ), and Random Early Detection (RED). We have added DWCS to this list of queuing disciplines, but are also improving the interfaces provided to end users. Specifically, access to the QoS architecture of Linux is currently quite cumbersome; it is typically through the netlink protocol, specifically designed to allow users to call kernel-internal functions. In response, we are implementing a socket library that adds QoS parameters to what appear to be regular socket calls. This library, called *QSockets*, extends the original attribute list of socket calls by a text string containing QoS parameters for the corresponding socket stream. Qsockets allow users to assign and modify per-stream QoS-attributes dynamically for DWCS and other queuing disciplines supported by Linux.

Resource Monitoring – ECalls and Monitoring Handlers. Quality management is not possible without understanding dynamic resource availability. Toward this end, we have implemented a general mechanism for extracting such resource information from the kernel, termed ECalls, and we are demonstrating its utility by construction of specific resource monitoring kernel extensions. The ECall interface and functionality is described in detail in this paper and is overviewed below. Next, we describe briefly one of the monitoring extensions we are currently constructing. Specifically, we are implementing a network monitor in the ELinux kernel that will allow applications to gather information and statistics about network utilization, loss-rate, etc., in an application-specific manner, thereby permitting them to enact application-level adaptations to try to adjust to dynamic changes in network behavior. The monitoring tools reside within the kernel such that only network information desired by the application (using filters) and network statistics are passed to the user-space. We are basing our implementation on Linux Socket Filters (LSFs).

ECalls – Asynchronous and Synchronous User/Kernel Communication. To efficiently access user-provided kernel extensions, such as the CPU and network schedulers and the monitoring handlers described above, the E(vent)Calls mechanism implements event-based communications from user to kernel space and vice versa. While ECalls is the topic of the remainder of this paper, we first outline the nature of the distributed quality management systems we are constructing with ECalls, with monitoring extensions, and with the user-specific and real-time kernel extensions described above.

Adaptive Resource Management with ELinux. Our general approach to quality of service management is the runtime adaptation of resources used by applications. This is achieved by enabling applications to determine or at least influence the adaptation decisions being made on their behalf. This approach is spelled out in more detail in a paper describing the Dionisys quality of service infrastructure [10], which permits applications to determine how, when, and where adaptations take place. This is done by executing application-specific handler functions with service providers, including those in the kernel. Such handlers have two tasks: (1) they monitor current resource usage and (2) they adapt resource allocation in conjunction with application needs. Our previous implementation of Dionisys was as a user-level resource manager on Solaris. Our ongoing work includes the implementation of suitable handlers as well as the resulting handler interactions within the kernel and across the user/kernel boundary in ELinux. In this case, monitor and handler functions can reside either within the application in user space or in loadable kernel-modules in kernel space. Thus, the ECalls mechanism described in this paper is a critical element of the manner in which quality management is performed in ELinux.

Event-based Communication. ECalls permits quality management to take place on a single machine, but it does not support inter-machine quality management actions, as indicated by previous and ongoing research on distributed quality management infrastructures. To address multi-machine quality management, we are adding another critical mechanism to the ELinux system, termed *real-time events*. The resulting *Artemis* real-time event system is offering an event communication library resident in the ELinux kernel, based on which inter-service and inter-machine events may be sent and received. In this fashion, we will construct a distributed implementation of the Dionisys approach to quality management. With Artemis (a) applications can communicate in an asynchronous and anonymous manner with other processes at the same host or at remote hosts, and (b) resource managers, residing either in kernel space or in user space, can exchange adaptation events within one host and across multiple hosts.

An effort somewhat orthogonal to the work described above, but also ongoing, is to more effectively use the fast networks and large main memories now existing for many Linux machines. Towards this end, our group is extending the Linux file system. One outcome of interest to the real-time community will be the file system's ability to control data caching and also the scheduling of caching actions, thereby offering quality of service support for distributed file management. The remainder of this paper focuses on ECalls.

3 The ECalls (Event Calls) Module

A flexible user/kernel interface is required to allow applications to exploit the performance advantages and real-time properties they seek through runtime kernel extension. The *ECalls (Event Calls)* mechanism is a lightweight facility for event-based, synchronous and asynchronous communication between user and kernel. ECalls has been implemented as a kernel-loadable module in the extensible ELinux kernel being developed by our group. ECalls allows unprivileged applications to access the services implemented by their (or other ECalls-equipped) kernel extensions.

To define the functionality of ECalls, we distinguish three different elements of event user/kernel and kernel/user communications:

- **Data Transfer**,
- **Event Delivery**,
- **Event Reception**.

Some mechanisms, like Unix signals, do not include a *data transfer* element (although Unix *real-time signals* are able to carry a 4-byte data value). *Event delivery* can be as simple as setting a flag, or more complex like transmitting a message over the network. *Event reception* is the act of *handling* an event's delivery, typically by invoking a handler function. The implementation of ECalls addresses two problems commonly found in general purpose operating systems like Linux [3]:

Data Passing Problem: Typically, data is passed explicitly, e.g., within a system call. In ECalls, data structures allocated when processes define the kernel events of interest to them are located in main memory accessible from both user and kernel space; they are used to avoid the explicit copying of data.

Control Passing Problem: The performance problems of user/kernel communication methods (like system calls) are well known. We introduce modifications to traditional call methods that result in improved system call performance through aggregation and filtering of events via lightweight non-blocking ECalls. In the reverse direction, from kernel to user, we modify a real-time scheduler developed by our group to efficiently combine the scheduling of user-level processes and events. ECalls also aims to reduce the cost and frequency of data transfer and event delivery, to efficiently execute event handlers, and to reduce the number of context switches incurred during user/kernel and kernel/user interactions.

Any kernel extension using ECalls and implemented as a kernel loadable module (such as resource managers or device drivers) is referred to as *kernel service* in the remainder of this paper. Such an extension uses ECalls as follows. First, it registers with the ECalls module, whereupon second, an application is able to use this service via *User-ECalls*. At the same time, third, other kernel services can use *Kernel-ECalls* to notify an application of the occurrence of certain events. In both cases, events can be accompanied by *event data*, which is copied into a *data segment* shared between kernel and user space and locked into memory.

We next describe the multiple (including real-time) semantics of event communication implemented by ECalls.

3.1 Data Transfer

ECalls uses two separate data segments per application, which are locked into memory (to prevent paging) and accessible from both user space and kernel space. The application uses the first data segment to transfer data from user space to kernel space, and the second data segment to receive data coming from the kernel service.

The implementation of a kernel service using ECalls decides how the structure of a data segment is interpreted. Figure 1 presents both methods: the data segment in (a) is used by the application to transmit data to the kernel service and is implemented as a *list* of data entries. The kernel service supplies the application with a C header file describing this data structure. The data segment in (b) is implemented as a *ringbuffer* with two pointers (front and back) pointing to the beginning and the end of the currently used part of the buffer. Since we use one segment per direction and per application, there is only one possible writer and one reader of this ringbuffer, eliminating the need for synchronization.

A data segment organized as shown in Figure 1a has the following structure:

```
struct ecall_data {
  int flag;
  unsigned long bit_pattern[MAX];
  [data part]
}
```

The first entry, *flag*, is incremented each time a new event is delivered and decremented each time an entry is read from the data segment; it therefore indicates the number of *pending* events, i.e., events that have been delivered but not yet received. The second entry, *bit_pattern*, is an array of MAX words, providing one bit per data entry in the following data part. The writer of the data segment sets the corresponding bit in the bit_pattern, such that the reader can apply simple bit operations to find the position of new events and it therefore allows for faster access to the data. The rest of the structure is the actual data part, which can include any kind of data types (excluding pointers). The reader of a data segment starts looking for new events by investigating the bit_pattern, starting from the MSB (most significant bit) to the LSB (least significant bit), and therefore implementing a priority ordering between the data part entries.

The second possibility of data segment organization is defined by the following data structure:

```
struct ecall_data {
  int flag;
  int front;
  int back;
  [data part]
}
```

The first entry, *flag*, has the same purpose as described above. *Front* and *back* are indexes to the beginning and the end of the part of the *ringbuffer* containing the data. The ringbuffer is implemented in the following data part. Since there are at most one reader and one writer to the ringbuffer, synchronization constructs are not necessary.

3.2 Event Delivery and Event Reception

This section briefly introduces the different methods of event delivery and event reception implemented in ECalls.

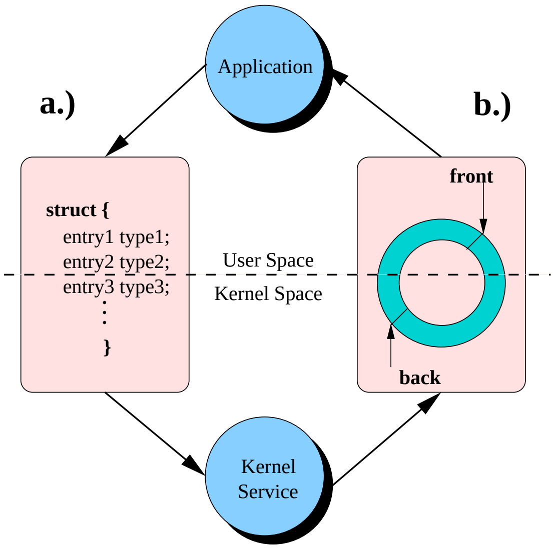

Figure 1: Example of data transfer using ECalls: The application transfers data to the kernel via the data segment shown in a.) (organized as list of data entries) and receives data from the kernel via the data segment shown in b.) (organized as ringbuffer).

3.3 User-ECalls

An application delivering a User-ECall to a kernel service typically increments the flag in the data segment. The next steps depend on the type of User-ECall used:

1. **Polling:** The kernel service may decide to poll for User-ECalls periodically if it runs frequently enough (e.g., a kernel thread running in an endless loop). It does so by calling the *poll function* provided by the ECalls module, which then checks the flags in all data segments of all applications registered with the kernel service. An alternative is to let ECalls poll the data segments periodically with a period T (provided by the kernel service). This allows the kernel service to go to sleep until ECalls finds a new pending User-ECall, in which case, ECalls invokes a handler function in the kernel service.

2. **Generic System Call:** A generic system call has been implemented. When an application calls this generic system call to request a kernel service, this call will be redirected dynamically into the corresponding module, where it will trigger the execution of a handler function supplied by the kernel service.

3. **Fast User-ECall:** A *Fast User-ECall* differs from an ordinary system call in three ways: (1) A non-blocking short-running service function (called *fast handler function*) supplied by the kernel service is invoked instead of a hardcoded (possibly blocking) system call function. (2) This function typically returns immediately without *bottom half handling* (executing the *slow* part of interrupts), *signal handling*, and possible *invocation of the scheduler*, which may occur when a system call returns. (3) The return value of the fast handler function determines if any or all of the tasks described in (2) are performed. It also decides if it is necessary to invoke a system call. In this case the Fast User-ECall acts as a *filter function*, i.e., only if certain conditions hold true (determined by the fast handler function), a system call will be executed.

4. **Delayed User-ECall:** Again, the fast handler function is invoked, but this time the time of execution lies in the future, more precisely, when the scheduler is invoked next. The advantage of this method is that several ECalls to the same kernel service lead to a single invocation of the fast handler function. As a result, several events are aggregated into one.

3.4 Kernel-ECalls

The following mechanisms of event delivery (and any combination thereof) are supported in ECalls:

1. **Real-Time Signals:** The kernel service raises

a.)

```
trap_to_kernel;
save_all_registers;
call_syscall_function;
if (bottom_halves_pending)
   call_bottom_halves;
if (need_resched)
   call_scheduler;
if (signals_pending)
   call_signal_handler;
return;
```

b.)

```
trap_to_kernel;
save_some_registers;
call_fast_syscall_function;
if (return_value & run_syscall)
   convert_into_syscall;
if (return_value & run_bottom_halves)
   call_bottom_halves;
if (return_value & need_resched)
   call_scheduler;
if (return_value & signals_pending)
   call_signal_handler;
return;
```

Figure 2: Simplified pseudo code for system calls (a) and Fast (Delayed) ECalls (b).

a real-time signal to the application. The details of the implementation of real-time signals are described in the POSIX.4 standard [5].

2. **Kernel handler function:** The kernel service executes a short non-blocking function, either supplied by the kernel service itself or by the application via another kernel-loadable module. A timeout algorithm ensures that the handler function finishes in time.
3. **Kernel handler thread:** As in 2., but here the function is allowed to block and will be executed in the context of a kernel thread taken from a thread pool.
4. **User handler function:** Again, as in 2., but the function resides in user space (locked into memory). This method raises severe security problems, which we will not address in this paper.
5. **ECall-Scheduling:** We modified the standard Linux scheduler and a real-time CPU scheduler (DWCS [11, 12, 9]) to support the integrated scheduling of tasks and Kernel-ECalls. Kernel-ECalls have priorities assigned influencing the scheduling decision in case several processes have Kernel-ECalls pending.

4 Implementation Details

In the following sections we investigate some of the mechanisms introduced above in more detail, namely the 'Fast' and the 'Delayed' User-ECalls for events originating in user space, and 'Real-time signals' and 'ECall-Scheduling' for events originating in kernel space.

4.1 Fast and Delayed User-ECalls

The functionality of both Fast and Delayed User-ECalls is basically the same: the kernel service provides a *fast handler function*, i.e., a handler function that runs for a very short time and never blocks (e.g., to set or reset a flag, or to wake up a kernel thread). For this purpose, a new software interrupt has been implemented. A typical system call in UNIX performs the steps shown in the (simplified) pseudo code in Figure 2a.

The code for 'Fast' and 'Delayed' User-ECalls has been modified as shown in Figure 2b. The execution of 'Fast' and 'Delayed' ECalls differs from the execution of system calls in the following points:

- Only those registers are saved that are most likely to be modified by the fast handler function. If other registers are used, it is the responsibility of the kernel service to save and restore them.
- The return value of the fast handler function determines if it is necessary to execute a system call (which is allowed to block), either chosen from the set of standard Unix system calls or another function provided by the kernel service.
- The return value of the fast handler function also determines if it is necessary to execute pending bottom halves, to deliver pending signals, or to invoke the scheduler.

These modifications allow us to use the fast handler function as a filter function for system calls, e.g., a 'Fast' ECall can poll for a condition in the kernel (e.g., the state of a socket) and, if necessary, invoke a system call (e.g., the *read* system call). A 'Delayed'

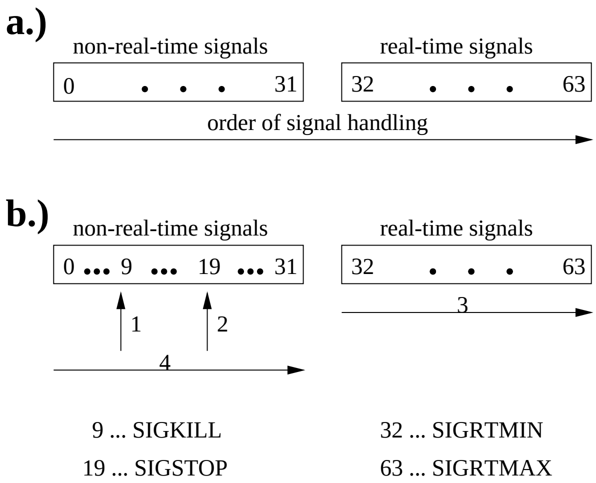

Figure 3: Standard Linux signal handling (a) and modified signal handling in ECalls (b). ECalls ensures that the real-time signals cannot be delayed by non-real-time signals by using the following sequence of signal handling: (1) SIGKILL and (2) SIGSTOP, (3) all real-time signals, and (3) all remaining non-real-time signals.

ECall, unlike a 'Fast' ECall, is not invoked immediately. Instead, its invocation is delayed until the next invocation of the scheduler. When the scheduler executes, ECalls polls for pending 'Delayed' ECalls and executes them before a scheduling decision is made. Note that a fast handler function has to be short and non-blocking to prevent the system of unpredictable behavior. If a fast handler function runs too long (due to error or malicious behavior), a timeout algorithm will abort it. Several 'Delayed' ECalls within one time slice of a process result in only one invocation of the fast handler function (i.e., we *aggregate* several ECalls into one).

4.2 RT-Signals and ECall-Scheduling

RT-Signals. Currently, there are up to 64 signals supported in Linux, 32 non-real-time, and 32 real-time signals. Real-time signals differ from ordinary signals in that they are queued to processes, they are handled according to their priorities, and they can carry some small amount of data. In Linux, signals are handled in the following order (Figure 3a): (1) All non-real-time signals (signals 0 to 31) and (2) all real-time signals from signal 32 (SIGRTMIN) to 63 (SIGRTMAX), giving signal 32 the highest priority among all real-time signals. Since it is possible to catch non-real-time signals (except SIGKILL and SIGSTOP) and execute user-defined handlers instead of the default action, it is possible to delay real-time signals unpredictably. Therefore, we changed the order of checking for pending signals to the order shown in Figure 3b: We first check the two signals which cannot be caught (and which will either exit or stop the application): 1. SIGKILL and 2. SIGSTOP. In step 3 we check all real-time signals beginning with SIGRTMIN and finally, in step 4, we check all remaining non-real-time signals. Using this scheme we are able to prevent a real-time signal from being delayed unpredictably by a non-real-time signal.

ECall-Scheduling with the Linux Scheduler. The Linux scheduler has been modified as follows: If a real-time process (a process in either the SCHED_FIFO or the SCHED_RR queue) has any Kernel-ECalls pending, it will be given preference over processes with the same or smaller priority. If only non-real-time processes are runnable (SCHED_OTHER queue), a process with Kernel-ECalls pending will always be given preference.

ECall-Scheduling with DWCS. To be able to efficiently support real-time processes, we use the hard real-time CPU scheduler DWCS (Dynamic Window-Constrained Scheduler), which assigns each process a period T, a runtime C, and a window-constraint x/y, meaning that a process will be scheduled y-x times for C time slots each, in a window of T*y time units. Each process can be scheduled once in a period of T, unless it is marked as *work-conserving*. In that case it is possible to schedule this process several times within a period. Details about DWCS can be found in [11, 12]. The original DWCS algorithm works as follows:

- The process with the shortest deadline (i.e., the time until its current period T expires) will be selected. If several processes have the same deadline, the process with the tightest current window-constraint x/y is chosen.

- If all processes have been scheduled at least once in their respective current periods, a work-conserving process will be selected according to the rules described above.
- If no real-time process is runnable, the next available best-effort task will be selected.

In a different paper [9], we presented boundaries for the worst-case delays of processes and showed that we are able to guarantee schedulability as long as the *processor utilization U* does not exceed 100%. We modified DWCS such that processes with pending Kernel-ECalls are given preference without violating the real-time guarantees:

- If two or more processes have the same deadline, the process with an ECall pending is selected as long as this does not result in an immediate violation of the other process' real-time guarantees.
- If a work-conserving process has been selected that already ran once in its current period, the next process with an ECall pending is scheduled instead.
- If a best-effort task has been selected by the scheduler, the next process with an ECall pending is scheduled.

In addition, we introduce the notion of an *ECall server*, which is a pseudo task with the attributes x/y, T, and C determined in the following way: x/y = 0/YMAX with YMAX being the highest possible value for the denominator. This assigns the ECall server the tightest window constraint possible. The service time C is the same as the service time of the process with the highest priority Kernel-ECall pending or C = 1 time slice if no Kernel-ECalls are pending. The *rest utilization Ur* of the system, which is the *maximum utilization* minus the *current utilization,* is used to determine the value of the period T (T=C/Ur). Each time the ECall server becomes the highest priority task, the process with the highest priority Kernel-ECall pending is selected instead. If there are no Kernel-ECalls pending, the scheduler selects a process according to the rules of the original algorithm as described above.

5 Summary and Related Work

Summary and Conclusion. Although some important parts of ELinux have been implemented, our work is still at an early stage. ECalls, DWCS task scheduling, and packet scheduling have been completed. The QSocket interface to the packet scheduler is being completed at this time. The Dionisys infrastructure and Artemis are under development, as is the kernel-level network monitoring facilities using ECalls. When completed, these kernel extensions will allow applications to monitor their resource allocations, adapt these allocations via application-aware resource managers, and adapt their own behavior to adjust to changing resource availability. The current implementation of ELinux is based on the Linux 2.2.13 kernel.
Extensibility in operating systems is key to flexibility and configurability. We propose ECalls to support the implementation of kernel services in modules by supplying a flexible interface for real-time and best-effort applications. ECalls distributes events in two directions, from user space to kernel space and vice versa. For events originating in user space, we introduce two new kinds of lightweight system calls, *Fast ECalls* and *Delayed ECalls*, both aiming at reducing the frequency and cost of event communication and context switches. For events originating in the kernel we modified the standard Linux scheduler as well as a hard real-time scheduler, DWCS.
Using DWCS, we are able to give the same hard real-time guarantees for the worst-case scenarios as described in [9], while giving preference to processes with pending events, therefore improving the average case delay.
Related Work. Our work on ELinux is an effort to add real-time and quality of service characteristics to the Linux kernel. Other approaches include KURT [8] developed at Kansas University and RTLinux [2] developed at the University of New Mexico. Some of our work is based on upcalls, proposed by Clark [4] to structure protocol code into layers, which allows for efficient protocol implementations. Gopalakrishnan and Parulkar [6] extended this idea to Real-Time Upcalls (RTUs), which have been introduced as an alternative to real-time threads and are intended for the implementation of user-space protocols with QoS guarantees. Banga, Mogul, and Druschel [1] introduce an event delivery system allowing applications to register interest in event sources like sockets. However, the application still has to poll for events, whereas ECalls is able to notify a process of pending events by executing a handler function and raising its scheduler priority. Finally, Saito and Bershad [7] describe an architecture that allows users to supply application-specific system call handlers in kernel extensions in the SPIN operating system.

References

[1] G. Banga, J. Mogul and P. Druschel, *A scalable and explicit Event Delivery Mechanism for UNIX*, Proc. USENIX Annual Technical Conference, 1999.

[2] M. Barabanov and V. Yodaiken, *Real-Time Linux*, Linux Journal, 1996.

[3] J. C. Brustoloni and P. Steenkiste, *Evaluation of Data Passing and Scheduling Avoidance*, Proc. 7th International Workshop on Network and Operating System Support for Digital Audio and Video (NOSSDAV), 1997.

[4] D. Clark, *The Structuring of Systems Using Upcalls*, Proc. 10th ACM Symposium on Operating Systems Principles, 1985.

[5] B. O. Gallmeister, *POSIX.4: Programming for the Real World*, O'Reilly, 1995.

[6] R. Gopalakrishnan and G. Parulkar, *Efficient User Space Protocol Implementations with QoS Guarantees using Real-Time Upcalls*, IEEE/ACM Transactions on Networking, 1998.

[7] Y. Saito and B. Bershad, *System Call Support in an Extensible Operating System*, Software - Practice and Experience, 1999.

[8] B. Srinivasan, S. Pather, R. Hill, F. Ansari and D. Niehaus, *Firm Real-Time Implementation Using Commercial Off-The-Shelf Hardware and Free Software*, Proc. 4th Real-Time Technology and Applications Symposium (RTAS), 1998.

[9] R. West and C. Poellabauer, *Analysis of a Window-Constrained Scheduler for Real-Time and Best-Effort Packet Streams*, Proc. 21st IEEE Real-Time Systems Symposium, 2000.

[10] R. West and K. Schwan, *Experimentation with Event-based Methods of Adaptive Quality of Service Management*, Technical Report GIT-CC-99-25, 1999.

[11] R. West and K. Schwan, *Dynamic Window-Constrained Scheduling for Multimedia Applications*, Proc. 6th International Conference on Multimedia Computing and Systems (ICMCS), 1999.

[12] R. West, K. Schwan, and C. Poellabauer, *Scalable Scheduling Support for Loss and Delay Constrained Media Streams*, Proc. 5th Real-Time Technology and Applications Symposium (RTAS), 1999.