‘QoS Safe’ Kernel Extensions for Real-Time Resource Management

Richard West and Jason Gloudon

Computer Science Department
Boston University
Boston, MA 02215
{richwest,jgloudon}@cs.bu.edu

Abstract

General-purpose operating systems are ill-equipped to meet the quality of service (QoS) requirements of complex real-time applications. Consequently, many classes of real-time applications have either been carefully developed to compensate for inadequate system support, or they have been developed to run on special purpose systems. This paper focuses on a safe extension architecture for general purpose systems, to allow applications to customize the behavior of the system for their individual needs. Using Linux as the basis for our work, we describe how application programmers can safely incorporate ‘service extensions’ into the kernel, so that application-specific QoS guarantees can be provided. We introduce the notion of ‘QoS safety’, which is concerned with meeting the QoS constraints of applications while maintaining system integrity.

Our safe extension architecture supports the dynamic-linking of code into the address space of the kernel, to affect service management decisions. Extensions are written in a type-safe language, to monitor and adapt resource usage on behalf of specific applications. Experimental results show that safe kernel extensions can lead to fewer service violations (and, hence, better qualities of service) for real-time tasks, compared to user-level methods that monitor and adapt system resources.

1. Introduction

Existing general-purpose operating systems are ill-equipped to meet the QoS requirements of emerging applications that are inherently real-time in nature. Consequently, many classes of real-time applications have been custom-developed to compensate for inadequate system support, or they have been designed to run on special-purpose systems. This has lead many researchers to provide middleware solutions (e.g., [12, 2, 19, 16]), that bridge the ‘semantic gap’ between the needs of applications and the services provided by the system. Unfortunately, middleware solutions lack fine-grained control over system resources, thereby limiting the rate at which resource allocations can be adapted. For example, many of the CORBA-based (middleware) adaptation frameworks now being developed (e.g., [19, 16]) are subject to the costs of monitoring and adapting system resource usage via processes and system calls, that have limited capabilities.

It is desirable to implement QoS management abstractions at the kernel-level, as opposed to using middleware, so that resources are managed more efficiently and better service is provided to real-time applications. Moreover, if these abstractions are implemented in general-purpose systems, improved service is possible for many desktop applications with QoS constraints (e.g., RealNetworks RealPlayer, or Windows Media Player). It is therefore beneficial to extend the behavior of existing commercial off-the-shelf (COTS) operating systems, by incorporating new service abstractions and interfaces. Unfortunately, most COTS systems are structured around monolithic kernels that have not been designed to support extensibility [18, 6, 3, 17] or specialization of system behavior. Although some systems support limited extensibility, by allowing device drivers to be loaded at run-time (e.g., Linux and Solaris), they lack support for code extensions that override or alter the behavior of system-wide service policies.

This paper focuses on a safe extension architecture for general purpose systems, that allows real-time applications to customize the behavior of the system for their individual needs. By incorporating application-specific code in the kernel, the system can make better resource management decisions for applications that need service guarantees. However, this approach raises a number of safety issues. First, traditional notions of safety concerning address space protection, type-safety and memory bounds checks for extension code must be enforced. Second, local resource management decisions for one application should not adversely affect the integrity of the system, or the service provided to other applications. Finally, the execution time of extension code must be bounded and small enough not to impact the overall service provided by the system. For this reason, we introduce the notion of *QoS safety*, which is concerned with meeting the QoS constraints of applications while maintaining system integrity.

Our architecture, known as 'SafeX', provides QoS safe guarantees for application-specific kernel extensions, by enforcing both compile- and run-time safety checks. SafeX supports the dynamic-linking of code into the address space of the kernel, to affect service management decisions. This approach provides finer-grained control over system resource management than current middleware methods implemented at user-level. As a result, system scalability is improved, while the service constraints of real-time applications are guaranteed. Specifically, SafeX extensions monitor service and negotiate changes to parameters of the management policies implemented by the core kernel, to ensure QoS guarantees for the corresponding applications.

Using Linux as the basis for our work, we describe how application programmers can safely incorporate service extensions into the kernel, so that application-specific QoS guarantees can be provided. We show the usage of SafeX to implement a simple CPU service manager. This service manager adapts the scheduling of various real-time tasks, to ensure their QoS constraints are met even when there are run-time changes in resource needs.

In summary, the goals of this work are:

• to provide a mechanism by which untrusted applications may dynamically-link 'QoS safe' code into the Linux kernel, to improve or guarantee the service provided to such applications, while maintaining system integrity,
• to define and implement appropriate safe interfaces that allow application-specific extensions to monitor and adapt services, thereby guaranteeing QoS even when there are changing resource demands, and
• to demonstrate that 'QoS safe' kernel extensions improve the service to real-time applications, compared to user-level methods of service (and, hence, resource) management.

Using safe kernel extensions, the following benefits are possible:

• resource management decisions can be made without having to schedule processes (that would otherwise make the decisions), thereby improving performance,
• finer-grained control of resource usage and allocation is possible, compared to user-level methods of monitoring and adapting such resource usage and allocation, and
• service is provided in a manner that better matches the needs of individual applications, that would otherwise have to play tricks with system calls and the basic abstractions offered by the system.

The rest of the paper is as follows: the next section describes some of the related work. Section 3 discusses SafeX in more detail, outlining some of the QoS safety issues. In Section 4, we show the performance of safe kernel extensions using a simple PID controller, that adapts the scheduling parameters of real-time tasks. Finally, conclusions and future work are discussed in Section 5.

2 Related Work

There has been significant work on adaptive resource management [14, 8, 15] and reservation [21, 10, 7, 4]. Likewise, many researchers have implemented entire QoS architectures [1, 12, 2] to meet the service requirements of real-time applications. By comparison, our work focuses on the provision of QoS safe mechanisms at the kernel-level of existing general purpose operating systems.

General purpose systems provide a set of generic service policies that are ill-suited to the needs of many applications, such as those with real-time constraints. This has stimulated research in extensible operating system designs [18] which give applications greater control over the management of their resources. In contrast, microkernels such as Mach offer a few basic abstractions, while moving the implementation of more complex services and policies into application-level components. By separating kernel- and user-level services, microkernels introduce significant amounts of interprocess communication overhead. This has caused microkernels to fall out of favor despite substantial reductions [9] in communication costs.

Other OS approaches, such as the Exokernel [5], try to efficiently multiplex hardware resources among applications that utilize library operating systems, implemented at user-level. In contrast, SPIN [3] is an extensible operating system that supports extensions written in the Modula-3 programming language. This language provides type-safety and memory protection, by enforcing interface contracts between code modules. Extensions signed by the trusted system compiler are deemed safe and may be loaded into the kernel address space at runtime. Interaction between the core kernel and SPIN extensions is mediated by an event system, which dispatches events to handler functions in the kernel, without the overhead of kernel/application boundary crossing. By providing handlers for events, extensions can implement application-specific resource management policies with low overhead.

Other related systems research includes the VINO [17] operating system. VINO supports system extensions known as *grafts*. These are object files generated and digitally signed by a trusted compiler as in SPIN. Unlike SPIN and the approach taken in SafeX, VINO employs C++, which is not type-safe. To enforce memory protection, *sandboxing* [13] techniques are applied to grafts. In fact, VINO runs graft code in the context of transactions, so that the system can be returned to a consistent state if execution of a graft is aborted.

Our approach has particular similarities to SPIN. However, our work differs in a few key areas. Namely, we provide a mechanism by which safe extensions can be incorporated into existing COTS systems. Furthermore, our approach supports extensions that monitor and adapt resource usage, to guarantee or improve QoS for real-time applications. Using SafeX, extension code is guaranteed to be QoS safe, since safety checks beyond those limited to existing type-safe languages (e.g. memory protection) are supported. These additional safety aspects of SafeX are discussed in the following section.

3 The SafeX Approach to QoS Safe Extensibility

SafeX supports both compile-time and run-time safety checks to:

- guarantee QoS constraints to real-time applications – the *QoS contract* requirement,
- enforce timely and bounded execution of service extensions – the *predictability* requirement,
- guarantee that a service extension does not improve the QoS for one application at the cost of others – the *isolation* requirement, and
- guarantee the internal state of the system is not jeopardized – the *integrity* requirement.

Collectively, the above requirements are needed to enforce a QoS safe system. With these requirements in mind, we can now discuss the SafeX approach to QoS safe extensibility in more detail.

Language Support – SafeX requires that service extensions be written in the Popcorn [11] programming language. Popcorn is designed for syntactic similarity to C, and is compiled to TALx86, an extended version of the Intel IA-32 assembly language. TALx86 is an instance of Typed Assembly Language (TAL) [11] that, by adding typing annotations and typing rules to traditional assembly language guarantees memory, control flow and type safety of TAL programs. Popcorn is supported by a number of TALx86 tools that can verify internal type consistency of TALx86 source files and linked object code.

Memory Protection – Extensions running within the kernel address space may potentially access and modify any data within the kernel and violate the memory protections enforced on user processes as well as the integrity of kernel data structures and code. The type safety of Popcorn prevents extension code from forging pointers to arbitrary addresses or casting pointers to arbitrary types. Therefore, by controlling the pointers passed to extension code, the parts of the kernel address space that may be accessed by an extension can be finely controlled. This provides the basis for SafeX interfaces discussed later.

Another issue raised by passing pointers to extensions is the possibility that memory referenced by a pointer may be deallocated or reused by the core kernel. Extension code cannot be trusted to stop using pointers to such memory after reuse or deallocation. Consequently, some form of garbage collection must be used to safely manage memory referenced by extensions. The current safe extensions implementation does not do such garbage collection, but defers deallocation of memory objects until all extensions referencing them are unloaded from the kernel's address space.

CPU Protection – Extension code may potentially execute for unbounded periods of time, taking control of the system. SafeX requires that applications reserve CPU time for extensions before they are executed. SafeX enforces time limits by aborting execution of extension code that exceeds its reservation. In this way, SafeX can limit the total amount of CPU time given to and used by extensions. It is worth noting that the CPU time used by an extension is charged to the associated application so that the total CPU time consumed on the behalf of the application is considered in its scheduling.

SafeX tracks extension execution time by decrementing a counter at each system timer interrupt. Consequently, extension code may not be executed while interrupts are masked, and they may not directly disable interrupts. In the current implementation, SafeX aborts extensions on return from the system timer interrupt to extension code, if they exceed their time limit.

Exception Handling – Checks inserted by the Popcorn compiler may detect certain errors as extension code executes, such as arithmetic exceptions and null pointer dereferences, which raise exceptions that may be caught and handled by extension code. If such exceptions are not caught within an extension, they are caught by a handler provided by the safe extension environment.

Synchronization – Certain kernel functions require synchronized access to shared resources. The Linux kernel uses locks and interrupt masking to create critical sections of code. As explained under CPU protection (above), extensions may not be allowed to mask interrupts. Locks are potentially problematic, as an extension holding a lock may be aborted when failing to catch an exception or by exceeding its execution time. Aborting an extension under such conditions may leave the resource being accessed in an improper state with locks unreleased.

SafeX addresses this issue by restricting the use of synchronization primitives to core kernel code or SafeX code. Critical sections of code requiring the use of such primitives must be implemented entirely within the trusted core kernel and SafeX. Since SafeX does not preempt the execution of core kernel code or SafeX code, critical sections are guaranteed to execute in their entirety, so that shared resources are never left in an unknown state. Extensions can then access shared resources through SafeX interfaces that encapsulate the necessary synchronization.

The above design issues solve the *predictability* and *integrity* requirements of a QoS safe system. However, further support is necessary to satisfy the *QoS contract* and *isolation* requirements. For this reason, we describe the additional management features that SafeX leverages to provide QoS safety.

3.1 Additional QoS Management Features Required for QoS Safety

To provide QoS safe resource management, new abstractions must be developed inside the kernel. For this reason, we have developed an adaptive QoS management system, called Linux Dionisys [20] (see Figure 1(a)). In this system, SafeX is incorporated into *daemon* processes running on each host of a distributed system. Application processes link with the Dionisys library to create service extensions. These service extensions are either *monitors*, *handlers* or *service managers*. A service manager is an encapsulation of a resource management subsystem within the kernel, and has a corresponding policy for providing service of a specific type. For example, a CPU service manager has a policy for CPU scheduling, while a network service manager might have a policy for flow control and/or packet scheduling. Observe that kernel service managers (see Figure 1(b)) run as *bottom half* handler routines that are invoked periodically or in response to events within the system. This enables extensions to execute asynchronously to application processes, so an application may influence the way resources are allocated to it even when not scheduled.

Each service manager has two queues for monitor and handler functions. A monitor function, M, bound to the queue of service manager, S, is executed periodically on behalf of an application, A. Using the interfaces provided by SafeX, M observes the service provided to A by S. Similarly, a handler function, H, that is bound to the queue of S and operating on behalf of an application, A', is executed in response to *events* from a corresponding monitor function associated with A'. H uses SafeX interfaces to influence changes to the service provided by S, so that improved service (or guaranteed QoS) is provided to A'. Moreover, applications can create *event channels* between their monitor and handler functions, so that deficiencies in one monitored service can trigger compensatory changes in the same or other service managers.

Monitors and handlers operate on *attribute classes*. These are data structures that hold the names of various QoS attributes and their corresponding values. For example, a name-value pair referring to CPU scheduling priority would be *priority-numeric_value*. As another example, an application-specific attribute for a video application might be *frame_rate-numeric_value*. These attribute classes are created by applications and linked into the kernel address space of a given host by a SafeX daemon. Service extensions and applications get and set these attributes by name, as long as they have the necessary access rights.

Each host has a separate attribute store for each application, that is identified by a *class descriptor*. Application processes and service extensions on a given host may only retrieve and set attributes in the corresponding attribute class on the local host (as identified by a *class descriptor*). As with service extensions, an application process can create an attribute class for deployment on a remote host. Access to this class is granted to remote processes that acquire permission from the class creator. Observe that the nameserver in Figure 1(a) maintains a database of bindings of class descriptors to attribute classes on each host, event channel ids to actual channels, and service extension names to their actual locations.

Each service manager is equipped with a *guard function* that is automatically generated by the code generator in a SafeX daemon process running on the same host. A guard function is responsible for the mapping of attributes, contained in attribute classes, to kernel policy-specific structures. It ensures that attributes are within valid ranges and will not affect the QoS guarantees to the corresponding application, or other applications. Observe that each SafeX daemon process implements a code generator and linker/loader, for compiling and linking extension code into the host's kernel address space. Moreover, each SafeX daemon is capable of generating code for run-time safety checks of extensions, thereby guaranteeing they have bounded execution time.

In summary, SafeX is used to implement safe kernel extensions in our QoS system called Linux Dionisys. Monitor and handler extensions affect resource management decisions so that QoS guarantees for applications can be met. This is necessary for the *QoS contract* requirement of a QoS safe system. Likewise, guard functions help satisfy the *isolation* requirement of a QoS safe system.

3.2 SafeX Interfaces

To affect changes to the service received by an application, the handlers (in a system such as Linux Dionisys) need interfaces to adjust the parameters of the underlying mechanism providing the service. Though handlers execute within the kernel address space, they cannot be trusted to directly modify core kernel data. SafeX, therefore, provides service extensions with interfaces to manipulate kernel data structures and perform operations requiring special privileges. For example, as mentioned previously, extensions must use SafeX interfaces to request operations requiring synchro-

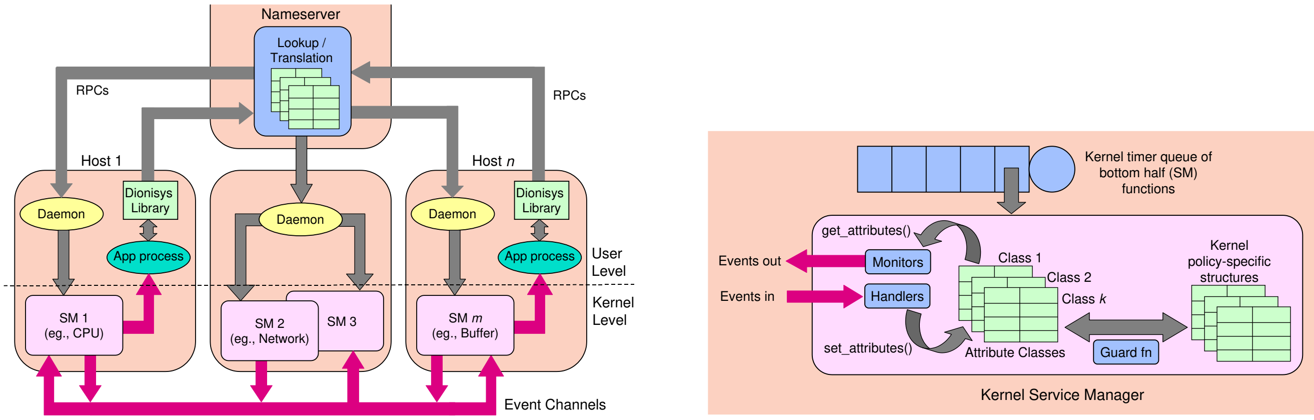

Figure 1. (a) The Linux Dionisys QoS system, and (b) the internals of a kernel service manager. The SafeX code generator and linker reside in the daemon processes of Linux Dionisys.

nization. SafeX interface functions may be used only by extensions possessing the capabilities for these interfaces. SafeX capabilities are in fact pointers which are unforgeable due to the type safety of the extension language.

SafeX interfaces, like system calls, must validate arguments passed to them by extensions. They must also ensure that requested operations are safe, as some operations or decisions, while not violating system protections, may have a negative effect on system performance. SafeX interfaces are therefore responsible for limiting the possible global effects of operations requested by extensions and require careful design balancing the degree of application control over resource allocations with concern for system stability.

Internally, SafeX uses functions that are similar to those provided by the Dionisys library. Applications directly access the Linux Dionisys interface, as shown in Figure 2, to specify service extensions. These library routines contact a nameserver, which in turn contacts the SafeX daemons using RPCs to take appropriate action.

4 Experimental Evaluation

This section describes a series of experiments performed on a 500MHz Intel Pentium III with 128 Megabytes of RAM, running a patched Linux 2.4.17 kernel that supports Dionisys and SafeX QoS management features. These experiments compare the performance of safe kernel extensions against user-level approaches for monitoring and adapting system resource usage. An elementary CPU service manager is created and used to evaluate the ability of safe extensions to meet the real-time needs of processes, even when there are fluctuations in resource demands. The CPU service manager provides the ability for service extensions to monitor and adapt the allocations of CPU time to corresponding applications over finite windows of real-time.

Figure 3 and shows the pseudo-code for monitor and handler extensions created on behalf of all application processes. QoS safety checks are performed using a simple guard function, as described in Figure 4. The CPU service manager decides that it is safe to allow a process to execute if its CPU utilization measured over twice its request period is at or below its target utilization.

Comparison experiments performed at user-level involve a real-time process (acting as a service manager) that accesses the `/proc` filesystem for state information about the CPU usage of all monitored processes. Using this information, the service manager issues system calls to affect the service provided to application processes. In each experiment, a set of applications specify their service constraints in terms of target, minimum and maximum CPU time required over finite windows of real-time. The performance of user-level versus kernel-level monitoring and adaptation is compared, when the progress of all service-constrained applications is affected by a bursty disturbance process. The disturbance attempts to hog all available CPU cycles during its active periods. It is modeled as a Markov Modulated Poisson Process, with average exponential inter-burst times of 10 seconds and average geometric burst lengths of 3 seconds. Both the user-level service manager and the disturbance process have Linux real-time priorities of 96. To ensure the system behaves correctly, kernel daemons are modified to run with real-time priorities set at 97.

Experimental Setup – User- and kernel-level service managers initialize each test application with a static priority of $80 * (target/period)$, where $target$ and $period$ denote the target CPU time required by an application in a given request period, measured in milliseconds. The CPU allocation for each process is monitored every 10 milliseconds. In the case of kernel monitoring, an extension function is

DClass_descr_t **DinitApp** (*void*)

- Obtains from the nameserver a globally unique class descriptor for an application.

int **DCreateAttrClass** (*DClass_Descr_t cd, char *sm_name, DAttr_t *attrib_class*)

- Creates an attribute class, referenced by *attrib_class* and associated with *cd*, in the service manager named *sm_name*.
- Returns 0 on success, -1 on failure.

int **DCreateSM** (*char *sm_name, int period, char *src_file*)

- Creates a service manager named *sm_name* from *src_file*. The service manager is executed every *period* clock ticks.
- Returns 0 on success, -1 on failure.

int **DCreateMonitor** (*DClass_Descr_t cd, char *mon_name, int period, char *mon_src_file*)

- Similar to **DCreateSM**(). The monitor is executed every *period* clock ticks and gets values from the attribute class identified by *cd*.
- Returns error code (for now, 0-success, 1-SM_no_exist).

int **DCreateHandler** (*Dclass_Descr_t cd, char *hand_name, char *hand_src_file*)

- Creates a handler named *hand_name*, from *hand_src_file*.
- Returns error code (for now, 0-success, 1-SM_no_exist).

int **DCreateEventChannel** (*DClass_Descr_t cd, char *mon_name, char *hand_name*)

- Adds *hand_name* to the subscription list for an event channel from *mon_name*.
- Returns error code (for now, 0-success, 2-Monitor_no_exist).

- *sm_name* uniquely identifies a service manager on a given host.
- *mon_name* and *hand_name* uniquely identify monitors and handlers associated with specific service managers.

Figure 2. Overview of the Dionisys API. SafeX uses similar functions internally.

```
void monitor(){
   //  Get target and actual CPU usage
   //  from application's attribute class.
   actual_cpu = get_attribute("actual_cpu");
   target_cpu = get_attribute("target_cpu");

   // Generate an event carrying difference
   // in target and actual values.
   raise_event("Error", target_cpu-actual_cpu);
}

typedef struct {char *name; int value;} event;

void handler(event ev){
   // Get nth sampled error between target
   // and actual monitored values.
   e[n] = ev.value;

   // Update process' timeslice by PID fn
   // of target and actual CPU usage.
   // u[n] is the timeslice adjustment
   // at the nth sampling interval.
   // Kp, Kd and Ki are PID constants.
   u[n] = (Kp+Kd+Ki)*e[n]-Kd*e[n-1]+u[n-1];

   // Set timeslice adjustment field of
   // an application's attribute class.
   // A guard function will
   // subsequently verify the QoS safety
   // of the new timeslice value.
   set_attribute("timeslice-adjustment", u[n]);
}
```

Figure 3. Pseudo-code for monitor and handler extensions used in the experiments.

```
guard (attribute, value):
  if (attribute == "timeslice-adjustment")
    if (CPU utilization is QoS safe)
      timeslice=max(0,target_cpu+value);
    else block process;
```

Figure 4. Guard function pseudo-code.

invoked periodically from a Linux timer queue. Handler functions adjust the timeslice of each process as necessary, using a PID controller similar to that described in Figure 3. As stated earlier, a guard function allows a process's timeslice to increase as long as its average CPU usage, measured over twice its request period, is not above the target utilization. For these experiments, the PID controller constants are $Kd = 1.3$ and $Kp + Kd + Ki = 1.4$. These values are the result of empirical studies to obtain stable results for the kernel service manager.

Figure 5 shows the percentage of CPU time allocated to three competing CPU-bound MPEG encoders (P_1, P_2 and P_3) having target CPU demands of $20mS$ every period of $100mS$, $30mS$ every period of $100mS$, and $80mS$ every period of $200mS$, respectively. The top figure shows the performance when service monitoring and adaptation is done at user-level, using the POSIX.4 SCHED_FIFO scheduling policy for the disturbance. The middle figure also shows user-level service management with the disturbance scheduled using SCHED_RR scheduling (with a $110mS$ timeslice), while the bottom figure shows the results for kernel-level service management.

In each case, the three MPEG encoding processes repeatedly encode 56 kilobyte (160x120 pixel, 24-bit color) frames into groups of pictures (consisting of 30 frames). The encoded frame sequence consists of a repeated pattern of I, P and B frames in the order $IBBBBPBBBB$. Each experiment runs for about 90 seconds, after which a script triggers the termination of all processes. This accounts for the sudden drop in CPU utilization (measured over 1 second intervals) at the end of the experiments.

The user-level service management suffers from the need to run a process to control resource allocation. When the disturbance uses SCHED_FIFO scheduling it cannot be preempted by the service manager. In all other cases, the service manager dynamically reduces the priority of the disturbance to allow the three time-constrained processes to run when necessary. However, the user-level approach requires knowledge of system-wide priorities, and service management is at the granularity of a timeslice. By contrast, kernel-

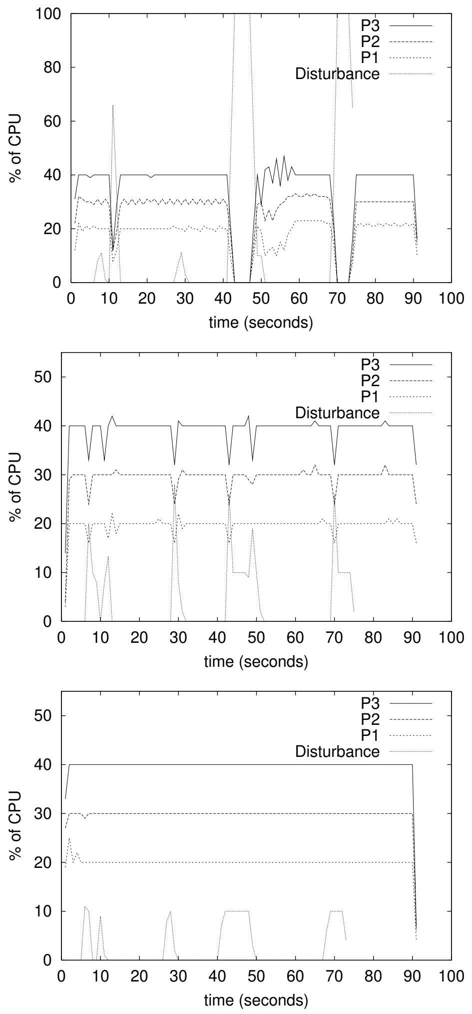

Figure 5. User- versus kernel-level CPU service management.

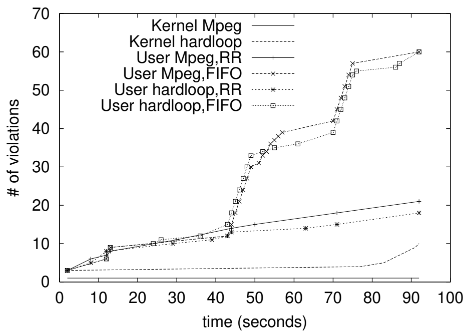

Figure 6. Total service violations over time.

level service management is not dependent upon scheduling policies and timeslice granularity.

Figure 6 shows the service violations for all three MPEG encoding processes over time. A violation occurs if a process receives less than its target fraction of CPU time over its specified period of real-time. Included in the figure are results for similar experiments in which the MPEG encoding processes are replaced by pure CPU-bound processes that simply perform infinite loops (i.e., "hardloop" processes). As can be seen, kernel-level service management leads to fewer service violations than all user-level cases, due to its ability to provide finer-grained resource management.

The user-level service manager controls the timeslice of each application process, P_i, by using the `kill()` system call to issue `SIGSTOP` and `SIGCONT` signals to P_i. Consequently, the additional overheads incurred by user-level service management include: (1) the sending of a signal to an application process, (2) the context-switching between the real-time service manager and an application process, and (3) the time to access the `/proc` filesystem for monitoring information.

Microbenchmarks for the above experiments yield the following. Over a series of 4000 measurements, the overhead of sending a signal to a process is $1.5\mu S$. The time to context-switch between any two processes is $2.99\mu S$. This is the result of using `sched_yield()` to ping-pong between two processes with highest real-time priority on an idle system. Reading `/proc/pid/stat` for an application process with a given `pid` is $53.87\mu S$. Finally, the average time to run both monitors and handlers at user-level is $190\mu S$ for all three time-constrained processes in our experiments. This includes the overhead of reading `/proc/pid/stat` three times. Collectively, these overheads affect the granularity of service management compared to our kernel-level approach. The only significant overheads incurred with the kernel approach come at the cost of executing monitor functions via a kernel timer queue (as well as associated handlers). These incur a cost of $20\mu S$ for all three processes, in the experimental scenario shown above.

Observe that with the above experiments, the timeslice is adapted for each real-time application process. In other experiments, we have adapted process *priorities* to manage the window-constrained allocation of CPU cycles. This shows slight performance improvements for user-level service management, because of the reduced overheads of manipulating priorities from user-level using system calls such as `sched_setparam()`. However, for situations where the 'semantic gap' between the service needs of real-time applications and the service abstractions provided by the kernel is significant, the performance of user-level service management can suffer greatly. This is due to user-level service management techniques having to leverage service abstractions via system calls in complex and unconventional ways.

5 Conclusions and Future Work

The SafeX system demonstrates safe downloading of executable code into the Linux kernel and the potential for efficient extensibility afforded by this capability. By using compiler and language run-time support, SafeX is able to create logical protection domains within the core kernel, isolating it from faults within application provided code. SafeX lays the groundwork for developing service managers within the Linux kernel which allow applications to monitor and customize their resource allocations in the manner most appropriate to their needs.

Using SafeX, we have implemented a simple CPU service manager, that adapts the scheduling of various real-time tasks, to ensure their service constraints are met even when there are run-time changes in resource demands. By embedding code inside the kernel, finer-grained management of system resources can be achieved. Experimental results show that safe kernel extensions can lead to fewer service violations (and, hence, better qualities of service) for real-time tasks, compared to user-level methods that monitor and adapt system resources.

Future work with SafeX includes the development of service managers and interfaces for memory and network resources, as well as the design of guard functions appropriate for different service policies. Support for stability analysis and verification of adaptive systems will also be developed.

References

[1] T. F. Abdelzaher and K. G. Shin. End-host architecture for QoS-adaptive communication. In *Proceedings of the IEEE Real-Time Technology and Applications Symposium*, Denver, Colorado, June 1998.

[2] C. Aurrecoechea, A. Campbell, and L. Hauw. A survey of QoS architectures. *Multimedia Systems Journal, Special Issue on QoS Architecture*, 1997.

[3] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. Fiuczynski, and B. E. Chambers. Extensibility, safety, and performance in the SPIN operating system. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, pages 267–284, Copper Mountain, Colorado, December 1995.

[4] J. Bruno, J. Brustoloni, E. Gabber, B. Ozden, and A. Silberschatz. Retrofitting quality of service into a time-sharing operating system. In *Proceedings of the USENIX 1999 Annual Technical Conference*, June 1999.

[5] F. K. Dawson R. Engler and J. O'Toole. Exokernel: An operating system architecture for application-level resource management. In *Proceedings of the 15th ACM Symposium on Operating System Principles*, pages 251–266, Copper Mountain Resort, Colorado, USA, December 1995. ACM.

[6] D. Ghormley, S. Rodrigues, D. Petrou, and T. Anderson. SLIC: An extensibility system for commodity operating systems. In *Proceedings of the 1998 USENIX Annual Technical Conference*, June 1998.

[7] M. B. Jones, D. Rosu, and M.-C. Rosu. CPU reservations and time constraints: Efficient, predictable scheduling of independent activities. In *Sixteenth ACM Symposium on Operating System Principles*, volume 31, pages 198–211. ACM, December 1997.

[8] B. Li and K. Nahrstedt. A control-based middleware framework for quality of service adaptations. *IEEE Journal of Selected Areas in Communication*, 17(9):1632–1650, September 1999.

[9] J. Liedtke. On micro-kernel construction. In *Proceedings of the 15th ACM Symposium on Operating System Principles*. ACM, December 1995.

[10] C. Mercer, S. Savage, and H. Tokuda. Processor capacity reservation for multimedia operating systems. In *IEEE International Conference on Multimedia Computing and Systems*. IEEE, May 1994.

[11] G. Morrisett, K. Crary, N. Glew, D. Grossman, F. Smith, D. Walker, S. Weirich, and S. Zdancewic. TALx86: A realistic typed assembly language. In *ACM SIGPLAN Workshop on Compiler Support for System Software*, pages 25–35, Atlanta, GA, USA, May 1999.

[12] K. Nahrstedt and J. Smith. The QoS broker. *IEEE Multimedia*, 2(1):53–67, 1995.

[13] T. A. R. Wahbe, S. Lucco and S. Graham. Software-based fault isolation. In *Proceedings of the 14th SOSP*, Asheville, NC, USA, December 1993.

[14] R. Rajkumar, C. Lee, J. Lehoczky, and D. Siewiorek. Practical solutions for QoS-based resource allocation problems. In *IEEE Real-Time Systems Symposium*. IEEE, December 1998.

[15] D. Rosu, K. Schwan, S. Yalamanchili, and R. Jha. On adaptive resource allocation for complex real-time applications. In *Proceedings of the 18th IEEE Real-Time Systems Symposium (RTSS)*, San Francisco, USA, December 1997.

[16] D. C. Schmidt, A. Gokhale, T. Harrison, and G. Parulkar. A high-performance endsystem architecture for real-time CORBA. *IEEE Communications Magazine*, 14(2), February 1997.

[17] M. I. Seltzer, Y. Endo, C. Small, and K. A. Smith. Dealing with disaster: Surviving misbehaved kernel extensions. In *Proceedings of the 2nd Symposium on Operating Systems Design and Implementation*, pages 213–227, Seattle, Washington, 1996.

[18] C. Small and M. I. Seltzer. A comparison of OS extension technologies. In *USENIX Annual Technical Conference*, pages 41–54, 1996.

[19] R. Vanegas, J. Zinky, J. Loyall, D. Karr, R. Schantz, and D. Bakken. QuO's runtime support for quality of service in distributed objects. In *Proceedings of the IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware'98)*, Lake District, England, September 1998.

[20] R. West. *Adaptive Real-Time Management of Communication and Computation Resources*. PhD thesis, Georgia Institute of Technology, August 2000.

[21] L. Zhang, S. Deering, D. Estrin, S. Shenker, and D. Zappala. RSVP: a new resource reservation protocol. *IEEE Network*, 7(5), September 1993.