Safe and Cooperative Coexistence of a SoftPLC and Linux

Robert Kaiser, Stephan Wagner And Alexander Zuepke
SYSGO AG
Am Pfaffenstein 14, D-55270 Klein-Winternheim, Germany
{rob,swa,azu}@sysgo.com

Abstract

Combining Linux and a softPLC in a single system stands to reason: Linux offers many facilities that modern PLCs are expected to support. However, existing Linux-based softPLC implementations so far have always placed the PLC "on top" of the Linux kernel, so its functional reliability depends on the correctness of the kernel. Due to its size, the Linux kernel can not be exhaustively validated or even proven correct. This has hampered applicability of the concept to safety-critical PLC systems. The approach described in this paper puts Linux and a softPLC "side by side" on top of a small microkernel, thus the two subsystems can coexist safely without being forced to depend on each other. In this way, the trusted code base of the PLC is reduced by several orders of magnitude, thereby enabling its certification according to applicable standards for safety-critical systems.

1 Introduction

Programmable logic controllers (PLCs) are the backbone of today's automation industry. Invented in the late 1960's [16], they have gradually evolved from a replacement for hard-wired relay circuitry to complex systems which are nowadays often expected to offer features such as a graphical user interface or internet connectivity. These features (and a lot more) are readily available from a general purpose operating system such as Linux, so the idea of combining Linux and a software-based PLC implementation (*softPLC*) in a single machine stands to reason.

However, PLCs are also frequently used in safety-critical applications. A malfunction in such a system could cause significant damage or even death or injury of human beings, so these systems have be to far more reliable than the average workstation computer. This reliability has to be proven prior to deployment by means of testing or even formal verification. While the standardised PLC programming languages ([10]) lend themselves to such exhaustive testing or validation, the same can not be said about the platform executing the code (i.e. the softPLC). Thoroughly testing such a platform is a costly task and the cost increases proportionally to the amount of code that needs to be examined. This "trusted code base" of the softPLC comprises all code that has the potential of affecting its correct function. If Linux is used as a base to support a softPLC, this amount of code increases by roughly one million lines (the code size of the Linux kernel), making a thorough validation or verification quite infeasible.

This paper introduces a way to enable safe and cooperative coexistence of a softPLC and Linux in the same machine. Unlike previous approaches, Linux is *not* used as a base for the PLC, rather, both Linux and a PLC exist side by side, based on a minimal amount of commonly trusted code. Our approach avoids dependencies between the two components that would otherwise require one of them to trust in the other's well-behaving. We first look at some existing Linux-based softPLC implementations, analysing their dependency paths. Subsequently, we present our approach (which is based on a microkernel), and finally, we show some initial results of an ongoing project which applies the presented approach. This project is aimed at integrating Linux and the well-known softPLC "CoDeSys" into a single system.

2 Existing Linux-based softPLCs

Technically, a softPLC is a program that executes specialised code, much like a Java virtual machine (JVM), however, unlike a JVM, a PLC needs to execute its code in a timely fashion. Therefore, any

softPLC needs its underlying operating system to guarantee real-time execution. Standard Linux can do this only to a limited extent, so, Linux-based softPLCs up to now have either used one of the various real-time kernel extensions, to provide "proper" real-time support for the PLC, or they have settled for the Linux kernel's limited real-time performance.

2.1 softPLCs based on Linux Real-Time Kernel Extensions

Real-time extensions to the Linux kernel such as RTAI [15] or RTLinux [2], work by integrating a dedicated real-time programming interface into the Linux kernel. There is only a formal separation between this interface and Linux in that real-time code is not *supposed* to invoke functions from the Linux kernel, but there is no way to *enforce* this rule: technically, any real-time application is able to invoke any Linux kernel function (which, in most cases, will lead to undefined behaviour). All real-time activities, and thus, in our case, the softPLC, run as privileged code in the same address space as the Linux kernel. Both Linux and the softPLC have uncontrolled access to the same code, data and I/O ports: they are not spatially separated.

The real-time scheduler runs Linux as its lowest priority process, thus, Linux only gets to execute when none of the real-time processes are ready. This means that for access to computation time, Linux is at the softPLC's mercy: if the PLC never blocks, Linux will not execute. So, in addition to the spatial dependency, Linux also depends on the softPLC temporally (but not vice versa): they are not temporally separated.

While the Linux kernel is able to affect the PLC, Linux application programs are not, unless they run with root privileges. So, assuming we can rule out any security holes (e.g. root exploits) from the Linux kernel, the softPLC need not trust Linux applications. Therefore, the trusted code base in this configuration comprises the Linux kernel and the softPLC.

2.2 User Space softPLCs

The plain, unmodified Linux kernel already comes with some facilities that allow for limited real-time functionality. If a softPLC application can accept deviations of timing in the range of a few milliseconds, it can be implemented as a Linux user process[18] [1]. Also, some of the above mentioned Linux real-time kernel extensions offer optional support to make the real-time interface accessible to user-space programs (e.g. LXRT [5]). This could also be used to implement a softPLC in user space, without having to make concessions regarding real-time performance.

With both approaches, the Linux kernel and the softPLC do not share the same address space and the PLC can not easily compromise the Linux kernel. Conversely, however, the PLC does depend on Linux making memory and I/O resources available to it as required, so, the two components are still not spatially separated.

Regarding temporal separation, if the PLC exists as a standard Linux user process, it needs to receive its allocation of computing time from the Linux kernel, while, if a real-time extension is used from user space, Linux again only gets to execute when all real-time processes are blocked. So, with both methods, there is no temporal separation: either the softPLC is at the Linux kernel's mercy, or vice versa.

Again, the total amount of trusted code, i.e. code which is needed to establish the softPLC's functionality, includes both the Linux kernel and the softPLC's code.

3 Approach: Separation of Resources

In both approaches mentioned above, the Linux kernel is in control of the spatial and -except for the LXRT case- also the temporal resources of the system. The softPLC crucially depends on this kernel: it must assume its entire correctness. Given the kernel's sheer size, this assumption is not very likely to be met, and auditing the trusted code base according to standards such as [9] is not feasible. Hence, such systems can not be used in safety-critical applications.

However, looking at a typical softPLC implementation, it turns out that only a very small part of the kernel's functionality is actually required to support it: All it really needs is a basic run-time environment, i.e. access to memory, I/O and processing time. Many of the advanced features that the Linux kernel has to offer are not required by the PLC. Nevertheless, since these features are implemented in the kernel, the corresponding code has to be trusted.

In order to reduce the amount of trusted code, our approach only implements those functions at the kernel level that are either technically impossible to do otherwise (i.e. require privileged instructions) or

[1] However, it should be noted that the timing deviations cited in [18] have been obtained by experimenting. Such experiments can only yield an *observed* worst case value, not the worst *conceivable* value. A safety-critical system must not assume that this value can never be exceeded.

that are needed to establish a secure runtime environment for user level programs. All other functionality that classical monolithic kernels like Linux tend to do in privileged mode can just as well be done by user level programs. A kernel that is designed according to these paradigms is usually referred to as a "microkernel"[12]. Such a microkernel provides only basic mechanisms which allow to divide a system's temporal and spatial resources into individual subsets. These subsets can also be regarded as virtual machines[2], each of which can host a complete operating system such as Linux along with all of its applications. Unlike RTAI or RTLinux, this approach in principle allows for any number of virtual machines to exist in a single machine, i.e. we can have more than one softPLC running independently in one machine. Any code running inside a virtual machine must run in user mode. Hence, an operating system will in most cases need some adaption. User-space applications, however, need not even be aware of the difference. Since it is confined to its own set of resources, an operating system running inside a virtual machine can not affect another operating system running in another virtual machine. The system is divided into *partitions*, which are completely independent of each other. Applying this to the case of a softPLC in combination with Linux, we can assign each of them to a different partition and thus be sure that they are independent of each other: the softPLC is no longer required to trust Linux and vice versa (See figure 1). Both subsystems coexist side by side (as opposed to one on top of the other). Both share a common trusted code base consisting of the microkernel itself (the only software component that runs in privileged mode) and a "System Software" Layer, a software module which runs in user mode, on top of the microkernel. The latter implements the policies for managing the partitions, based on the mechanisms provided by the microkernel.

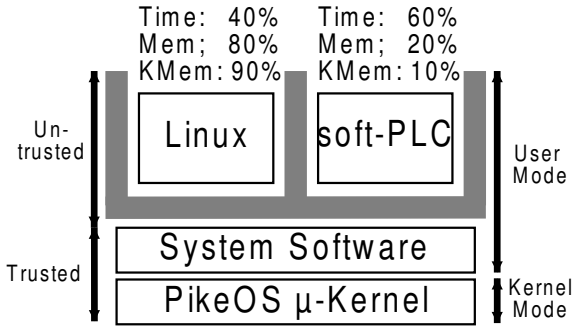

FIGURE 1: *Partitioned system*

In the following subsections, we will describe the separation of resources in more detail, looking at spatial and temporal resources in turn.

3.1 Spatial Separation

To be able to run a guest operating system in a partition, a microkernel must provide access to portions of the system's hardware, i.e. memory, memory-mapped or (on x86) port-mapped I/O registers, and interrupts. It must also provide mechanisms for the guest operating system to distribute and revoke access to these resources by applications, running on top of it. This has to be done as flexibly as possible to allow all kinds of operating systems as guests and to keep the porting complexity small [12].

To ensure independence between guest systems, these resources must be distributed by a commonly trusted party (the kernel). To do so, the microkernel assigns to each partition a set of virtual address spaces, which act as containers for resources. With these methods, the microkernel is able to guarantee that no partition can interfere with another.

But this only covers user space resources. The microkernel itself needs memory to manage pagetables, stacks, ready-, and wait queues, etc. These resources also need to be separated, or else one partition could mount denial of service attacks by making the kernel consume huge amounts of memory, e.g. by mapping one page to all available virtual addresses within its address space [14].

Spatial separation in our approach is geared to the ARINC 653 standard [1]: the distribution of resources is made statically according to a fixed configuration. The standard requires the most severe restrictions to the system setup. The static configuration of resources is implemented by the "System Software Layer". In addition to the seperation facilities, the System Software Layer also offers communication services like shared memories and notification mechanisms between partitions. This allows for secure communication between guest systems accross partition boundaries.

3.2 Temporal Separation

Most microkernels were not initially designed with the goal of real-time execution in mind. Consequently, many of them provide for spatial resource separation as described in the previous subsection. However, only few also provide the necessary facilities to enable deterministic distribution of temporal resources (i.e. processing time).

[2]This is also referred to as *para*virtualisation in [3]. In fact, we consider virtual machine monitors such as Xen to be specialised implementations of the microkernel concept.

The goal of a virtualisation environment is to give every operating system executing within a partition the illusion of having a constant proportional share (i.e. a percentage) of the overall processing time available for its own use. This would imply that the execution time assigned to individual partitions increases in a linear fashion as shown by the dash-dotted lines in figure 2. In practice, however, the processor(s) can only be time-multiplexed across the partitions, i.e. every partition has a time slot during which it is active. The idealised linear progression of execution time is approximated by a ramp-like function as shown by the solid lines in figure 2.

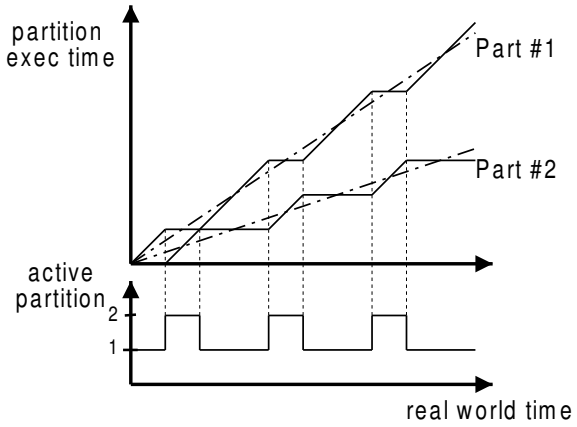

FIGURE 2: *Per-partition execution time vs. real world time*

The quality of this approximation improves as the granularity of time assignments, i.e. the absolute duration of time slots is made shorter, however, the resulting increase in switch frequency leads to excessive overhead.

A softPLC is a classical example of a time-triggered system: It has to be invoked periodically at fixed points in time. If these invocations are not synchronised against the partition switches, the softPLC will experience unpredictable delays. These delays can hit the softPLC anywhere during its cycle, i.e. they can delay the PLC from becoming active (thus increasing jitter), or they can delay its computation, thus increasing any deadline the PLC may have to maintain. Obviously, it would be desirable to switch between partitions as quick as possible in order to keep these delays small, however, we have already seen that this is limited by the switching overhead quickly becoming excessive. Another, possibly better way is to synchronise the partition switches against the softPLC's cycle: The virtual machine hosting the softPLC must be activated just in time for the softPLC to become active and it must receive an amount of time sufficient for the PLC to complete its job. In this way, a softPLC hosted by a virtual machine can deliver the same real-time characteristics as a non-virtualised one. Since a PLC is a strictly time-triggered system, the points in time when it is to be activated are known in advance, i.e. they are defined as a function of time only. From this follows that, if partition switching should be syncronised, the partition switches must also be defined strictly as a function of time only. Most existing virtualisation environments fail at this point: They allow their individual virtual machines to suspend themselves and, once a VM does so, they immediately switch to the next one. This method, while advantageous from a processor utilisation point of view, makes it impossible to predict the partition switch times and so, a time triggered system (e.g. a PLC) running in a virtual machine can not be syncronised against the partition switches.

The time partitioning concept of our approach is an enhanced variant of ARINC 653[1], a standard which is popular in avionic systems: The ARINC 653 standard assigns periodically repeated time slices of fixed duration to virtual machines. Every VM can tell in advance when and for how long it will be active, so it can synchronise its own scheduling to the scheduling of virtual machines. However, a disadvantage of the ARINC 653 method is that all VMs must consume all of their allocated time: there is no way to suspend VMs when they are idle, so VMs have no choice but to "burn away" any time that they can not use for themselves. The amount of time allocated to a VM is determined by worst case assumptions: A real-time system running in a virtual machine must be able to complete its task within this time frame under all conceivable conditions. However, worst case scenarios, though possible, tend to occur only rarely and so most real-time virtual machines tend to have far more time available than they actually use in the average case. Therefore, systems using the ARINC 653 approach tend to exhibit a rather poor processor utilisiation. If all VMs would host real-time systems only, this waste of resources would be inevitable, however, in our system, we also have Linux, a non-real-time system which could make good use of any excess computational resources. Therefore, in our approach, we combine the strictly time-driven ARINC 653 scheduler with a priority-driven scheduler: any amount of time that is assigned to, but not used by the softPLC, is dynamically re-assigned to Linux. This is achieved by placing Linux into a "background" time partition which is always in a runnable state and assigning a priority lower than the softPLC's to it.

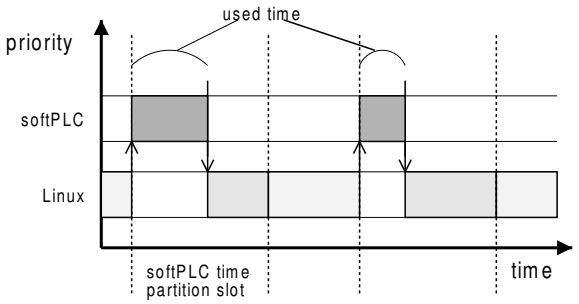

FIGURE 3: *A high priority softPLC in an own time partition and Linux in the background.*

The amount of time that the softPLC can consume is limited, so, even if the PLC made an attempt to monopolise the processor at high priority, it could still not starve Linux (as would be the case with RTAI or RTLinux). The softPLC always gets its guaranteed share of resources at predetermined points in time and also Linux has a guaranteed minimum amount of CPU time. Linux cannot disturb the PLC during its execution and the PLC can not detain Linux's share of CPU time: The softPLC as well as Linux need not trust each other any more.

4 Practical Implementation

In this section we describe some implementation details of the microkernel with a special focus of partitioning, CoDeSys, and the Linux kernel as guest operating systems.

4.1 PikeOS Microkernel

In the 1990s, Sysgo have been developing their own microkernel. Initially modeled after the L4 version 2.0 microkernel API as specified by Jochen Liedtke in [13], the kernel evolved from the generic microkernel approach to a specialized kernel for embedded systems with a focus on real-time and partitioning. The kernel is written almost entirely in C to facilitate porting. Currently supported architectures are: x86, PowerPC, MIPS, and ARM.

The general system design today is a static configuration approach to fit the microkernel concepts to practical embedded systems, which usually do not need complex dependencies and capabilities. The complexity depth was reduced to typical use-cases, like a Linux environment running in a partition on top of a System Software Layer, next to other services with a similar grade of requirements.

4.2 Basic Concepts

Like L4, the PikeOS Microkernel provides the concepts of tasks, threads and inter process communication (IPC). Unlike L4, it extends the task concept with the notion of "abilities", and it offers partitioning of resources as well as computation time.

A thread is the basic entity of execution. It has a unique identifier (UID) and is bound to a task. It is implemented by a register context which is scheduled at a given priority and time partition. Optionally, it can register a handler to receive exception messages. A thread can also register itself as interrupt handler, thus facilitating interrupt handling at user level.

Each thread is assigned to a task. A task is a container for threads, which all share the same virtual address space. Furthermore, it defines a set of "abilities", which restrict the allowed system calls of its threads. Tasks are organised in a hierarchical tree. All tasks have a parent task and may have child tasks. Unlike the clans and chiefs concept [12], the microkernel manages task based communication and interrupt handling rights. These rights can be inherited from their direct parent by child tasks, where the parent is able to further restict them. Thus, a child can at most have the same rights as its parent, but never more. A task also restricts the priorities of its threads to an upper bound, the so-called "maximum controlled priority" (MCP). Again, a child's MCP can at most be the same as its parent's, but never higher.

Tasks are also grouped in "resource partitions": Tasks belonging to the same resource partition share a common kernel resource pool. Whenever a kernel call is made which causes the kernel to allocate a resource, that resource is taken from the caller's kernel resource pool. Thus, tasks belonging to the same resource partition can mount denial-of-service attacks against each other, but tasks with different resource partitions can not.

The kernel supports inter process communication (IPC) as standard communication mechanism. An IPC operation is synchronous and allows to transfer an arbitrary set of data or memory mappings between threads in different tasks.

Further extensions to the L4 concepts are an asynchronous notification mechanism and a special map system call. The notification mechanism has been added to simplify notification in a time partitioned environment. The map system call has been added to simplify process creation. This allows to establish all mappings before the first thread of a new task is activated.

4.3 System Software Layer

The System Software Layer distributes all available resources of the system according to a static configuration. Dynamical resource allocation is explicitly not possible.

In addition to this resource distribution , the system software layer also provides facilities for communication across partition boundaries. Probably the most basic of these facilities is shared memory: It is possible to define shared memory segments which are accessible to multiple guests in different partitions. To all communicating parties, such a segment appears as an external resource, i.e. neither of them "owns" it. This is necessary in order to maintain secure isolation between the parties. In addition to shared memory, the system software also provides two slightly more elaborate communication mechanisms: There are *queuing ports*, unidirectional buffered, message based communication channels similar to pipes, and *sampling ports*, unbuffered mechanisms roughly comparable to mailboxes. Both concepts were modeled after the ARINC 653 standard [1]. These ports can be accessed in a blocking and a non blocking mode. Again, resources used by these communication facilities are owned by the system software, a trusted component, so any operating system hosted in a partition can rely on them.

Also, the System Software Layer provides mechanisms for health monitoring and partition control. This makes it possible to define how to react on exceptions and errors, depending on the current system, partition, or process state.

The system software layer also provides a rudimentary file system and a a generic device driver infrastructure.

4.4 Porting CoDeSys

There already exist a large variety of different softPLC implementations, both as commercial products or as open source projects. Therefore, rather than re-invent yet another softPLC implementation which would surely have been an inferior to these mature products, we decided to port a readily available softPLC implementation to our virtual machine environment. We chose "CoDeSys" by 3s Smart Software Solutions because of its availability in source code form, its portability and –last but not least– because it is well accepted in the market.

PLC programming is typically done with an offline tool in one of the programming languages standardised in [10]. In the case of CoDeSys, the tool runs on a standard PC and connects to the softPLC via Ethernet and TCP/IP. The tool generates IEC compliant code which is downloaded to the softPLC for execution. The tool also supports remote debugging of PLC code via the same network link.

Internals

The softPLC itself runs several threads:

- The *IEC threads* execute the system's IEC function blocks. They can be time-triggered, in which case they are executed periodically at predefined points in time, or they can be "event-triggered", in which case their execution is controlled by an "IEC event" (basically a boolean variable). It should be noted that even "event triggered" IEC threads are actually time-triggered from the microkernel's point of view: Technically, they poll their IEC event periodically and, depending on the event's current value, they either execute IEC code or wait for the next cycle.
- The *control thread* triggers the PLC's IEC threads and it supervises them: If any of them exceeds their time allocation, it stops the offending thread and signals a fault condition.
- The *communication threads* are responsible for communication via TCP/IP or –optionally– serial link. Unlike the other threads mentioned so far, these threads are event-triggered as they are activated by external events. Also unlike the other threads, these threads do not have to fulfill hard timing requirements.

The CoDeSys PLC has previously been adapted to a number of different real-time operating systems. It expects its run-time environment to support basic real-time, multithreading functionality. To port CoDeSys to our environment, we used the already existing implementation of the POSIX PSE51 ("Minimal Realtime System Profile") [11] API to provide this functionality. CoDeSys also needs TCP/IP sockets for communication. These were provided by a port of "lwIP", the "light weight IP" stack [7].

Hardware interfacing

In typical softPLC applications, the PLC interacts with the outside world via external I/O components which are connected to it either directly ("local I/O") or via various different fieldbuses. Thus, it is crucial for our PLC implementation to support a wide variety of fieldbuses. In the current prototype version, local I/O and Profibus are already supported and ongoing development aims at adding support for CAN, EtherCAT and Profinet. Unlike most other virtual machine environments, our system is able to partition the hardware's I/O resources, too, i.e. for

every partition, it is possible to individually select the memory-mapped or port-mapped I/O registers that should accessible to it. In this way, the available selection of CoDeSys drivers for different fieldbus modules can be re-used in our environment with little to no change.

Communication with Linux

Communication between the softPLC and Linux is facilitated by the communication mechanisms provided by the system software layer. Queuing ports and sampling ports can be accessed from both sides like normal I/O devices. Either side can use this interface in ways to eliminate unwanted dependencies beween the two sides, for instance, the PLC can avoid being blocked by an attempt to send messages in case the Linux side does not read its end of a sampling port fast enough.

4.5 Implementing Linux as a Virtual Machine

Porting Linux to a virtual machine is not a new concept. The currently existing ports to IBM s390, L4 Microkernel [8] and XEN [3] are paravirtualization approaches and have proven that running Linux on a hypervisor is possible and efficient. With User Mode Linux (UML) [6], even a port to Linux itself exists, where the virtual machine is a Linux process on the host system.

Another method for running Linux (or in fact, *any* operating system) in a virtual machine is full system virtualization. Virtualizers like VMWare [17], Virtual PC or QEMU [4] execute an unmodified operating system environment, where the OS kernel is unaware of the virtualization. The technique typically used by these virtualizers is dynamic recompilation of the binary code at runtime, when the code cannot be executed directly. In most cases all access to hardware is emulated, it is almost impossible to enable direct access to the host system's hardware without compromising the entire system. Only simple, protocol based hardware like USB can be directly virtualized.

The emulation approaches require a set of drivers on the host system to be used by the virtualized hardware. For example, some kind of graphics driver is necessary to display the emulated framebuffer. However, this also increases the trusted code base of the host system, which makes this approach infeasible for our purposes.

We use the paravirtualization approach where the Linux kernel is aware of the microkernel environment. Then, Linux can properly use the assigned components of the underlying hardware *and* use specialized drivers e.g. to communicate with other virtual machines.

Our paravirtualized kernel, named *P4Linux*, was designed according to the following design goals:

- Inhibit any side effects on other virtual machines.
- Keep full ABI compatibility, don't rewrite applications.
- Keep overhead small for fast execution.
- Ease porting to other processor architectures.

Details on the Implementation

The Linux architecture layer is scalable enough to run on either a small embedded system, or a large multi processor server cluster. Therefore, we implemented the microkernel awareness like a new architecture.

To do so, the following concepts had to be adapted:

- Virtual Memory Management
- I/O resource management
- Exception Handling
- Interrupt handling

Since User Mode Linux (UML) was already already available, and since it uses similar abstractions, we used its "SKAS" approach as a starting point.

Memory Management The Linux memory management layer internally uses multilevel pagetables like on x86 to maintain the virtual address space of its userspace processes. After modifications to a pagetable, the Linux kernel has hooks to flush a CPU's Translation Lookaside Buffers (TLBs). Taken from the UML approach, we hooked these functions, too, and used the mechanisms provided by the microkernel to modify address spaces. The microkernel offers system calls to map a set of pages from one address space to another, and to revoke this mapping by unmapping pages from one's address space.

However, the mapping system call transfers pages from the virtual address space of one task to the virtual address space of another. This stands in contrast to a *normal* Linux implementation, where physical pages are mapped to virtual addresses in a process' address space.

From the Linux kernel's point of view, two different virtual memory views exist: the virtual memory of its own address space and the virtual memory of its userspace processes.

So, to keep the implementation easy, only one big chunk of physically contiguous memory can be used as Linux main memory. This eases calculation of DMA transfer addresses, as a fixed offset is used to calculate bus addresses.

Also, there is a restriction: as the P4Linux kernel and all user space processes run in their own address spaces, the Linux kernel cannot access the memory of its userspace processes directly. Taken from the UML approach, all copy in and out operations by the kernel must be translated back to the corresponding addresses in the Linux kernel address space.

Memory mapped I/O resources are no problem at all. The Linux kernel starts with all I/O resources it can access already mapped at start up. A driver calling *ioremap()* to get a virtual mapping of an I/O device now gets a pointer to these premapped memory areas.

Exceptions From the microkernel's point of view, all Linux userspace activities cause exceptions. Exceptions, like TLB misses or non-microkernel system calls, are propagated to a userspace exception handler. For Linux processes, the P4Linux kernel registers as this exception handler. On each exception, it receives an IPC containing the complete register context of the faulting process and tries to solve the exception by mapping a page in case of a pagefault, executing a Linux system call, or sending a signal.

Interrupts The microkernel's mechanisms to serve interrupts to userspace handlers need an active thread to wait for an interrupt. The P4Linux kernel offers a set of threads waiting for these interrupts and other asynchronous events, for example reception of data from other partitions. Finally, these threads call the Linux *do_IRQ()* implementation to invoke the registered handlers.

An interrupt blocking mechanism is implemented by priority: if the Linux kernel doesn't want to receive interrupts, it raises its priority above all asynchronous helper threads and it lowers its priority again after the critical section. This mechanism is quite fast, because the microkernel supports lazy priority switching.

Problems and Side-Effects

The isolation between partitions crucially depends on the microkernel's address spaces, which are implemented by means of memory management hardware (an MMU). But if there exist devices which are able to access the bus directly, bypassing the MMU, then this safety concept fails. The next generation machines are likely to have IOMMUs that will solve this problem once and for all, but in current hardware, there are several devices capable of mastering the bus while bypassing the MMU. Any guest operating system that has access to one of these busmasters is indirectly able to access any location in the system, i.e. it can cause any amount of damage.

One solution for this problem is to restrict hardware access only to polling devices or framebuffers and revoke any bus mastering capabilities from the devices.

Another solution is to divide a driver into a trusted and an untrusted section: only the trusted parts must access the hardware and are implemented outside the Linux kernel in a separate server, whereas the untrusted parts remain inside the Linux kernel and handle the software stacks.

Both solutions introduce additional communication overhead.

The current implementation is also restricted to uni processor execution, due to its simple but fast locking approach via priority switching.

Performance

An approach to port Linux to a microkernel can never be as fast as a native Linux kernel running on the bare hardware.

We benchmarked the overhead of our approach with two different benchmarks: a microbenchmark calling performance critical functions in a loop and a real-world benchmark, compiling a Linux kernel. The target platform for the tests is a 600 MHz Intel Celeron / 256 MB RAM embedded industrial PC with Debian 3.1 installed. We chose this platform, as it is widely used in fanless industrial environment.

The Linux kernel used for this benchmark is taken from SYSGO's ELinOS distribution. It is a modified 2.6.15 kernel. We compare the time of a kernel build on a native i386 target against our microkernel environment, running P4Linux. When possible, both kernels utilize the same set of drivers and configuration options.

The microbenchmark calls performance critical functions like *getpid()*, a null system call, 1000 times in a loop and compares the CPU's time-stamp counter before and after the loop. The time-stamp counter on the x86 architecture is incremented every CPU cycle. The presented values are the normalized results of 1000 calls, also presented in CPU cycles.

Testcase	Native	P4Linux	Factor slower
getpid()	344	7004	20,4
fork()+_exit()	66428	388883	5,9
vfork()+_exit()	19199	54524	2,8
fork()+execve()	216041	1537256	7,12
vfork()+execve()	214620	1536807	7,16

TABLE 1: *Microbenchmark*

The *getpid()* benchmarks shows an overhead of about 6700 CPU cycles for all system calls compared to the native implementation, caused by two address space switches and IPC messages for one system call.

Especially performance critical are *fork()* and *execve()* operations. The table shows the differences in process creation between *fork()* and *vfork()* when the created child immediately terminates. The last two tests show the costs of address space space filling and immediately flushing (the forked child process terminates immediately).

Linux	Compilation time
Native Linux	361 s
P4Linux	441 s

TABLE 2: *Compiling the Linux Kernel*

The macrobenchmark shows the overall system performance impact. We compiled a Linux kernel (2.6.16 for i386, configuration "allnoconfig") on the test system and measured the overall time. The P4Linux system is about 22 percent slower. This is mainly caused by extensive use of the *fork()* system call by the build system.

5 Conclusion and Outlook

In this paper, we introduced a new approach to coexistence of a softPLC and Linux in a single machine. Unlike previous solutions, this one does not force the two components to trust each other. It thus makes it possible to apply the approach to safety-critical systems. The method does have an impact on system performance, however, we feel that this impact is acceptable: most of today's computer systems do not suffer from lack of performance, instead they have severe safety and security problems. With this background it seems sensible to sacrifice some performance while gaining significantly on the safety and security side. The softPLC is only one example of a real-time system that can be hosted by PikeOS, there are many others.

Current work on the PikeOS system aims at providing a large variety of real-time or non-real-time operating system interfaces to run on top of PikeOS (one of them being the CoDeSys softPLC). Work on the softPLC itself aims at increasing the number of fieldbuses that it supports.

In the next stage, the softPLC system is planned to be turmed into a networked, distributed system of PLCs which can share portions of their state accross a network by means of a publish/subscribe mechanism.

From the Linux point of view, the next goal is to approximate performance to the native implementation. One of the most important issues is to reduce process creation time. Furthermore, the overhead of processor mode switches can be decreased by batching system calls. Additionally, new TLB interfaces are verified, which support page table virtualisation.

Future improvements to the PikeOS Microkernel are support for multicore systems and current hardware virtualisation techniques for the x86 architecture.

References

[1] ARINC. Avionics Application Software Standard Interface. Technical Report ARINC Specification 653, Aeronautical Radio, Inc., 1997.

[2] M. Barabanov. A Linux-based RealTime Operating System, 1997.

[3] P. Barham, B. Dragovic, K. Fraser, S. H, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the Art of Virtualization, 2003.

[4] F. Bellard. Qemu, a fast and portable dynamic translator. In *USENIX 2005 Annual Technical Conference, FREENIX Track*, pages 41–46, 2005.

[5] E. Bianchi and L. Dozio. Some experiences in fast hard realtime control in user space with rtai-lxrt. 2nd Realtime Linux Workshop, Orlando, 2000.

[6] J. Dike. User-mode Linux. Online: user-mode-linux.sourceforge.net, 2001.

[7] A. Dunkels, L. Woestenberg, K. Mansley, and J. Monoses. lwIP embedded TCP/IP stack. http://savannah.nongnu.org/projects/lwip/, Accessed 2004.

[8] M. H. Hermann Haertig and J. Wolter. Taming linux, 1998.

[9] IEC. Functional Safety of Programmable Electronic Systems: Generic Aspects. IEC 65A (Secretariat) 123, International Electrotechnical Commission, February 1992. Technical Committee no. 65, Working Group 10 (WG10).

[10] IEC. IEC 61131-3: Programmable Controllers - programming languages. Technical report, International Electrotechnical Commision, 1993.

[11] IEEE. *1003.13-1998 IEEE Standard for Information Technology — Standardized Application Environment Profile (AEP) — POSIX® Realtime Application Support*. 1998.

[12] J. Liedtke. On μ-Kernel Construction. In *Proceedings of the 15th Symposium on Operating System Principles*, pages 237–250, 1995.

[13] J. Liedtke. L4 Reference Manual - 486, Pentium, Pentium Pro, 1996.

[14] J. Liedtke. Preventing denial-of-service attacks on a μ-kernel for WebOSes. In *Proceedings of 6th Workshop on Hot Topics in Operating Systems*, Cape Cod, MA, May 5–6 1997.

[15] P. Mantegazza, E. L. Dozio, and S. Papacharalambous. RTAI: Real Time Application Interface. *Linux J.*, 2000(72es):10, 2000.

[16] R. Morley. *History of the PLC*. R. Morley Inc.

[17] VMware. VMware ESX Server Online Documentation, 2005.

[18] P. Wurmsdobler. Linux for real-time PLC control? Slower is easier. *InTech*, Industrial Computing:49–51, November 2001.