Utilizing Linux Kernel Components in K42

K42 Team

modified October 2001

This paper discusses how K42 uses Linux-kernel components to support a wide range of hardware, a full-featured TCP/IP stack and Linux file-systems. An examination of the run-time environment within the Linux-kernel is the starting point for a presentation of the strategy taken to incorporate Linux-kernel code in K42. There are four basic aspects of this strategy; presenting K42 as an architecture target for Linux-kernel code, developing interfaces allowing K42 code to call Linux-kernel code, implementing some Linux-kernel internal interfaces using K42 facilities and selectively building Linux-kernel code to include only the desired components. Of these four aspects the first is the most technically interesting and is the focus of most of this paper.

1. Introduction

One of the goals of K42 is to develop a platform that can become a basis of operating systems research for a broad community. To attain this goal K42 must acquire a critical mass of available applications and support a broad range of hardware devices to allow it to be used with commonly available, inexpensive hardware. This paper addresses K42's approach to the second problem of how the existing Linux-kernel code base is used to provide the desired range of hardware drivers available for use in K42 (the first problem is addressed in another white-paper). In addition to using Linux for hardware driver support, Linux networking and file system code is used to provide a stable, inter-operable environment and provide a basis for comparative evaluation of K42 and Linux implementations of core system services.

2. Constraints

There are a number of high-level issues to be considered that determine our approach to using Linux-kernel components in K42. This section outlines the constraints from which the chosen approach arises.

The goal of effort described in this paper is to allow for a broad range of Linux-kernel components to be used in K42. Modifying each component to work with K42 would be an enormous task, made even more daunting by the constant development activity being performed on this code (which makes maintenance of changes as serious problem). Therefore, it is highly desirable to develop a means by which Linux-kernel components can be used without modification. Instead, appropriate interface translation mechanisms must be developed that allow for running unmodified Linux-kernel components.

Secondly, K42 aims to explore new operating system design principles, without the burden of legacy architectural decisions. Any method chosen for incorporating Linux-kernel components into K42 must do so in a manner that does not impose significant architectural requirements on K42. This is an issue that must be addressed in using Linux file systems and block devices, both of which are closely entangled with Linux's memory-management infrastructure (a component that is not intended to be incorporated in K42).

In considering these constraints, the chosen strategy calls upon K42 to provide to the desired Linux-kernel components a run-time environment consistent with what these components would expect in an unmodified Linux system. The first step to providing such an environment is for K42 to present itself as a target hardware architecture for Linux (in the same way as real hardware architectures such as Alpha, i386 and PowerPC do). This requires implementing the basic functionality required of architecture-specific code in Linux (e.g., assembly-level constructs, definition of locking mechanisms) in an "emulation layer" that can be linked with the individual Linux-kernel components to be used in K42. This approach can be considered from the K42 perspective as providing a "Linux target-hardware" personality to those processes that utilize Linux-kernel components.

Additionally, for each type of Linux-kernel component to be supported in K42 an interface layer must be developed (e.g., a layer that provides entry points for K42 to perform operations on the Linux TCP/IP stack, comparable to system calls such as `sendto`). Finally, there is a number of generic services (such as dynamic memory allocation) within the Linux-kernel that must be emulated. Providing such services is a simple task of translating Linux-kernel interfaces to K42 interfaces and will not be discussed in detail. Finally, in order to use the K42 implementations of such services and to avoid using unwanted Linux-kernel code the build procedure must be modified to build only the desired components.

3. Linux Execution Environment

In order to understand the target-architecture emulation layer that is used to present K42 as a hardware architecture to Linux, it is essential to discuss some basic aspects of how the Linux kernel works and its expectations of the environment it runs in. The chosen approach for defining what Linux code sees as a "processor" makes most of these issues trivial to handle, thereby increasing confidence in the correctness of this approach.

To begin, note that Linux-kernel code (when running on a real hardware target such as i386) is in one of three "execution contexts" referred to as:

Top Half

This is the highest priority execution mode, encompassing low-level interrupt and exception handling routines. In this mode, code cannot block and should minimize use of locks (and any locks must be spin-locks). In general, "Top Half" interrupt handlers should not do much work, only enqueue work to be done by lower-priority execution modes.

Bottom Half (aka SoftIrq)

This is where the most work is done in response to interrupts (e.g. handling TCP packets). Typically, once a "Top Half" has terminated it will invoke "Bottom Half" handlers on the current CPU. This code will then work on whatever work has been generated by any past "Top Half" interruptions or any "Top Half" handlers that may interrupt it. Once there is no more work to do the original execution context that was in effect before the first "Top Half" preemption is restored. Note that there is at most one "Bottom Half" context in existence on any CPU.

Thread Mode

"Thread Mode" This mode executes kernel threads that implement kernel-internal services, or are performing system calls on behalf of user-processes. Code in this mode can find a run-queue element that represents it, remove itself from the run queue and then block by yielding.

There are several assumptions that Linux-kernel code makes about these execution modes that have far reaching implications. In particular, how these execution modes interact has a significant impact on how locking and synchronization is performed inside the Linux-kernel. These assumptions are:

- Code running in Thread Mode is subject to being preempted by Top Half code.
- Top Half code that has preempted Thread Mode code may convert itself into Bottom Half mode. Thus Bottom Half code may be running in preference to Thread Mode code, but the actual preemption is always accomplished by an entry into Top Half code.
- In case of preemption the code that is doing the preempting may do so without switching stacks, thereby preventing the code that has been preempted from running. Thus Top Half and Bottom Half code must never attempt to grab a spin-lock that may be held by the Thread Mode code they have preempted. This is because the Top Half and Bottom Half code must run to completion to free the stack of the Thread Mode code so that it may run. (This stems from how Linux handles interrupts on i386 processors.)
- While Top Half code is running it cannot be preempted by itself. However, it may be preempted by a different Top Half handler; perhaps an interrupt handler for a different device.
- Bottom Half mode can only be preempted by Top Half code.
- Thread Mode code is never preemptively rescheduled in favor of other Thread Mode code. That is, when running inside the Linux-kernel, in Thread Mode, no other Thread Mode code will run (on that CPU) unless the currently running Thread Mode code explicitly calls a scheduling routine. In other words, the Linux-kernel is not fully preemptible. The scheme used in K42, where Top Half and Bottom Half code runs on threads, allows for full preemption even to the extent that real-time tasks may run in preference to interrupt handlers.

4. K42 Architecture Emulation Layer

This section explains the emulation layer that presents K42 as a hardware architecture target for Linux-kernel code to be compiled against. In the next section some of the issues faced in developing individual interface layers for various Linux-kernel components used in K42 are presented.

The model of treating K42 as a hardware platform that Linux code is compiled to requires an environment that mimics a real hardware target for Linux. This is accomplished by using K42 threads to simulate hardware processors. Consequently, a thread executing Linux-kernel code is said to present a "Logical CPU" (LCPU) to that code (this term should not to be confused with the term "virtual processor" already in use in K42 terminology).

Any K42 thread about to execute Linux-kernel code is assigned an LCPU identifier. This identifier is returned when the Linux-kernel code queries for the identifier of the processor it is running on (`smp_processor_id`)(e.g., when an ethernet driver attempts to determine which per-processor packet queue on which it is to enqueue a received ethernet frame onto). Thus, this LCPU binding determines the processor-specific Linux-kernel data structures that will be used by that thread.

It is ensured that a thread using an LCPU will be the only thread assigned to that LCPU until it leaves the Linux-kernel code space. This approach takes advantage of the fact that K42 as a Linux target can provide a large number of LCPU's for the Linux-kernel code to use. Satisfying the assumptions Linux code has about its execution environment is easier with this model as there are guarantees that only one execution context can be active on any processor/LCPU. A caveat to this approach is that care must be taken to ensure that the appropriate sequence of execution contexts is run on each LCPU (e.g., a Top Half interrupt handler enqueueing a packet on a pre-processor packet-queue requires that a Bottom Half handler on that LCPU be run to consume that frame).

The number of LCPUs in the system is determined in the Linux-kernel code at compile time — by defining a constant that determines the sizes of various arrays whose elements determine per-processor data structures. If needed, additional LCPUs can be created (at compile time) with minimal cost.

Implementation of these semantics is accomplished by requiring that any K42 code about to call any Linux-kernel code (specifically any code compiled with Linux-kernel headers) create a "LinuxMode" object on its stack. A pointer to this object is maintained as thread-specific data that can be easily accessed by a thread at any time (it is always pointed to by well-known register). The LinuxMode object maintains the thread's Linux-specific environment (such as its LCPU binding – configured by the LinuxMode constructor) while it is executing Linux-kernel code and can be efficiently queried for this information. LCPU's are chosen in a first-fit manner with some consideration given to processor locality (LCPU's have a preference as to which physical processor they are used on).

In addition to the LCPU binding information, the LinuxMode object keeps track of the execution context that the thread is using. When the K42 thread has finished its call to the Linux-kernel space it destroys the LinuxMode object. The destructor of the LinuxMode object enforces the rules of execution context interactions (executing Bottom Half code if necessary).

A LinuxMode object thus represents a unique execution environment that exists while the object is in scope. This property is useful as there are many occasions when there is a sequence of calls to Linux-kernel code (perhaps with K42 logic in-between) that must be executed in the same environment, on the same LCPU, without other activity on the LCPU in between (e.g., without the network stack having processed any other packets on the particular LCPU).

4.1. Interrupt Handling

The individual components or properties of the emulation layer can now be considered in the context of the larger system. In particular it is useful to see how these components interact to allow Linux interrupt handling code to run within K42.

Like any other Linux target-architecture, the K42 target must implement interfaces for registering interrupt handlers. The K42 implementation records the interrupt-vector to handler-function binding and registers as the interrupt handler for that particular interrupt with the K42 kernel.

When K42 invokes its native interrupt handlers, it does so by creating a new thread on which handler routines execute. Thus when the emulation layer receives an interrupt to handle using Linux-kernel code, the handler is already running on a thread. On this thread the emulation layer creates a LinuxMode object configured for Top-Half mode, remembers the set of interrupts it is to handle, re-enables interrupts and executes the appropriate Linux interrupt-handlers. These are the same steps that any other Linux architecture-specific code must do in invoking an interrupt handler routine.

The important aspect of this is that the emulation layer has complete freedom in assigning the interrupt to any unused LCPU. From the point of view of the Linux-kernel code, it finds itself to be exceptionally fortunate to be running in an environment with many CPUs and all interrupts being sent to CPUs which are not doing any work thereby precluding the possibility of same-stack preemption.

Once the appropriate Top-Half code has been executed, the LinuxMode object is destroyed and its destructor invokes Linux's generic Bottom-Half mode entry function (the object is converted from Top-Half mode to Bottom-Half mode). This results in the appropriate Linux Bottom-Half code executing (e.g., TCP/IP processing in response to an interrupt on an NIC).

Such an environment is similar, though not identical, to what typically occurs inside the Linux-kernel. The most obvious difference is that Top-Half code is executed as threads. This may result in higher latencies in invoking Linux interrupt handlers because the interrupt must pass through K42's interrupt management, thread creation and thread scheduling code. However, thread creation in K42 is fast and Linux Top-Half code is expected to be short (and thus likely to execute entirely within a scheduling quanta) and so it is not expected, nor has it been observed, that these issues pose a problem to running Top-Half code as is done in this manner. There has been no indication that this latency issue is an issue of correctness rather than performance. On the other hand, a stock Linux-kernel is limited as many actively running Bottom-Halves as there are processors. This is not a limitation in K42 and as a result it is likely that latencies that involve Bottom-Half processing may be improved (for example, a worst case scenario where a network packet consumes a lot of time in Bottom-Half would not delay other packets since those packets can be processed by other LCPU's).

This scheme for interrupt handling is fairly simple allowing for possible optimizations to be considered. For example, it may be worthwhile to execute several Top-Half contexts on an LCPU before a Bottom-Half context is invoked. This would be useful in a high network-load environment where each Bottom-Half context typically has several packets to process (from the several Top-Half handlers that ran before it). Such a scheme may reduce Bottom-Half overhead, improve locality and reduce lock contention for global data structures within the TCP/IP stack.

4.2. Locking and Scheduling

Linux leaves it up to architecture-specific code to define low-level locking constructs (i.e. spin-locks). This allows the code that defines the K42 architecture target to define the implementation of locks within the Linux code to correspond to standard K42 locks.

All code running in K42's Linux-kernel environment is running on a thread. This is typically not the case with Linux-kernel code, which must be prepared to deal with the possibility that interrupt handlers prevent threads from running by re-using their stacks (and such interrupt handlers are not considered threads). By virtue of K42 implementing all Linux execution contexts using threads, K42 "spin and block" lock implementations can be substituted for all low-level locking mechanisms in the Linux-kernel code space. This is not possible in the normal Linux-kernel environment since the preemption assumptions force Bottom-Half and Top-Half code to use only spin-locks and interrupt disabling/enabling.

Linux leaves it up to the architecture-specific code to define locking mechanisms. Therefore the emulation layer defines these locks as opaque structures of the same size as standard K42 locks. Any operations on these locks (including initialization) are implemented as functions that understand and operate on the underlying K42 locking mechanisms. This technique applies only to the fundamental low-level locking facilities (i.e., spin-lock interfaces). High-level locking and synchronization facilities (e.g. "big reader-writer locks") that rely on the basic low-level locking functions are not modified except for the swap of underlying low-level locking implementations.

The implementation of locks for the Linux code-space as described above allows for easy integration into the K42 threading infrastructure. Besides locking mechanisms there remain only a few scheduling-related interfaces inside the Linux-kernel code that must be supported by the emulation layer to complete the emulation the Linux-kernel scheduling infrastructure.

4.3. Selective Building

The build process is an important aspect of the K42 emulation layer. The goal here is not to maintain a full running Linux-kernel inside K42 but rather to use individual components. Thus, a modified build procedure is necessary. The build process needs to isolate individual components and link them with the appropriate Linux-kernel objects, interface-layer objects and emulation-layer objects. This allows for support facilities used by the targeted components to be provided by Linux-kernel code or K42 code as appropriate. The ultimate goal is to allow for a K42 user to build an application that includes a Linux-kernel component, in the same was that Linux users build modules for insertion into the kernel at run-time. For example, a Linux ethernet driver may be compiled into an application that runs as an "ethernet server" outside the K42 kernel. In contrast on a standard Linux system, such a driver may be inserted into the kernel as a module.

5. Component Interface Layers

The emulation layer makes it is possible to build and run Linux-kernel components inside K42. However, this alone is not sufficient because the components must be adapted to interface with the K42 environment. K42 does not have system-call entry points like Linux and K42 expects potentially different interfaces than those provided by the Linux system-call interfaces. Consequently, it is necessary to create an interface layer that allows for Linux-kernel components in K42 to interact with the rests of the system. This section describes the various interface layers that are being developed.

5.1. Hardware Drivers

Allowing K42 to use hardware drivers from Linux requires that the interface layer provides mechanisms to allow drivers to register handlers for interrupts and for interrupts to be correctly routed to those handlers. Currently this involves a simple translation between K42 and Linux interfaces. This is sufficient because Linux-kernel drivers are run within the K42 kernel. A longterm goal is to allow for the possibility of running drivers as applications outside the K42 kernel (requiring development of more robust interfaces).

Most Linux hardware drivers expect rely on several interfaces to interact with the PCI-bus layer to facilitate device detection and initialization. Consequently the functionality of these interfaces must be provided by the K42 run-time environment. To this end K42 presents the generic Linux PCI layer with a virtualized PCI controller that presents the actual hardware the system is running on. The generic Linux PCI layer can then use this virtualized PCI controller to expose PCI resources to drivers. This approach allows drivers to use the Linux PCI interfaces as they normally would with a minimal amount of effort.

5.2. TCP/IP Stack

The Linux TCP/IP stack interfaces with low-level NIC drivers below and the Linux system-call interface above. At this time it is sufficient to use the existing interfaces to the NIC driver layer (as it resides in the same address space as the TCP/IP stack).

To interface with applications, a new set of high-level interface functions has been developed. These interface functions present an object-oriented view of sockets and export the appropriate IPC interfaces. The job of these functions is to translate calls to a K42 socket object into operations on data structures inside the TCP/IP stack. The high-level system-call entry-point functions in Linux (i.e., the first-level implementations of `sendto`), are not appropriate as there is some functionality in these functions that is not necessary for our purposes. For example, in the K42 environment, pointers to socket structures, not file-descriptors are passed to the high-level TCP/IP interfaces (file-descriptors are managed by applications). Providing an interface layer between K42 and the Linux-kernel component simplifies the integration process.

5.3. Block Device Layer

The Linux block-device layer provides K42 with simple, device-independent interfaces for performing block-device I/O. K42 uses Linux-kernel interfaces to perform block I/O (e.g., `ll_rw_block`). This is done in a straightforward manner, though it is important to note that Linux-kernel buffer and page caching functionality is avoided. This is accomplished by formulating calls to Linux-kernel code , from the block-device interface layer, in a manner that explicitly removes buffers from the cache thereby keeping it empty (this does not impose any extra costs). The Linux kernel's caching mechanisms must be avoided to ensure that caching functionality remains solely within K42.

5.4. File Systems

Using Linux file system code requires translating calls to native K42 file system interfaces into the appropriate operations on Linux file system data-structures. Furthermore, to allow for file systems to be implemented outside the K42 kernel, calls from the Linux file system code to the Linux block-device layer must be translated into calls on K42 block-device interfaces. These calls may communicate with the another process that contains the block-device driver.

File systems pose a unique problem since Linux file systems are tightly integrated with the Linux memory-management sub-system. In the case of the TCP/IP stack's interactions with low-level NIC drivers, the interactions are simple (packets are passed back and forth between the two layers). File systems are more complicated because file systems talk to block-devices using paging/memory-mapping interfaces through the Linux-kernel's caching facilities and these are integrated with the memory-management sub-system. These problems result in a more complicated interface layer, but do not prevent the desired functionality from being attained.

6. Conclusion

The mechanisms and techniques outlined in this paper have been used to date to enable a wide range of functionality within K42 running on 64-bit PowerPC hardware. K42 uses Linux's pcnet32 ethernet driver, the sym53xxx SCSI adapter driver, the TCP/IP stack and the ramfs file system. All this has been accomplished with only a handful of cosmetic changes to architecture-generic Linux-kernel code (to facilitate compilation of the code on AIX, with an AIX assembler and to gain access to several "static" functions). Current development efforts are focused on the FAT and EXT2 file systems as well as enabling the same functionality on the 64-bit AMD platform.

Above all, this exercise demonstrates the flexibility of Linux-kernel code. Most importantly it should be noted that model that is used to create the run-time environment (binding LCPUs to K42 threads) is feasible only because the Linux-kernel supports multi-processor hardware. The emulation layer takes advantage of this and in fact disables all mechanisms that the Linux-kernel normally uses to multiplex a single CPU among multiple execution contexts. In doing so, the Linux-kernel run-time environment in K42 is fully preemptible and supports a greater degree of possibly concurrency.