A Practical Approach to Linux Clusters on SMP Hardware

Karim Yaghmour
Opersys inc.
www.opersys.com
karim@opersys.com

Abstract

Scalability is key to the success of any mature operating system. The classic approaches to scaling operating system kernels rely on increasing the threading of the kernel. Such threading has, however, led to maintenance problems for many kernels.

The Linux kernel development community is currently pondering whether to follow the path of extreme threading or try an entirely different approach. One such approach, which has been promoted by Larry McVoy and which is substantiated by the existing body of operating system research, is the implementation of a cluster of kernels on SMP hardware. This paper presents a suggested architecture for the implementation of such a cluster.

The architecture presented here is built around 5 components: the Adeos nanokernel, the Linux kernel, a set of virtual devices, a kernel-mode bootloader, and existing clustering solutions. In this architecture, the development of each component is independent of the development of the other components. The implementation of this system does not require any central development authority, though it does require the common definition of interfaces, and agreement on the architectural principles embodied in this proposal.

As is shown, most components required to build such an architecture already exist. Each component, nevertheless, requires some level of modification in order to be easily integrated in the architecture presented. The required modifications and enhancements are discussed and various research avenues are presented. So too are the caveats of the architecture and possible future enhancements.

Disclaimer

This paper is meant to encourage the discussion about practical implementations of Linux-based SMP clusters. It is by no means a complete design document, though it does outline a feasible approach. Comments and corrections are welcomed.

Main proposal features

- No changes to the kernel's virtual memory code.
- No changes to the kernel's scheduler.
- No changes to the kernel's lock granularity.
- Minimal low-level changes to kernel code.
- Reuse of many existing software components.
- Short-term accessibility.

1 Introduction

Recent discussions in the Linux development community have made it clear that increased threading of the kernel is likely to render the source unmaintainable. As pointed out by Larry McVoy, past attempts to thread kernels beyond the 4 and 8 CPU threshold have indeed resulted in very hard to maintain kernels.

Many have argued that smart locking schemes could be developed to facilitate the kernel's scalability. Others have argued that maintenance problems are unlikely to happen to Linux because of the peer review nature of open source.

Nevertheless, McVoy's assessment is clearly substantiated by those who have worked on SMP clusters including the team who worked on Disco [12, 15]. It is therefore not the purpose of this paper to convince you that SMP clusters are the way to solve kernel scalability problems. The assumption being made is that you are already convinced of this and are seeking implementation ideas.

The rest of this paper presents a scheme for implementing Linux clusters on SMP hardware using mostly existing software components and extending some of them for the purpose. As we shall see, most extensions are actually trivial and should not require any substantial development effort. In addition, the scheme presented here is rather intuitive and easily extendable. The addition of any features which are not currently part of this scheme should not require any modification of the basic architecture.

Because the scheme suggested reuses many existing components, our discussion mainly concentrates on the implementation details necessary to bring all the existing components together with as little effort as possible.

We start by reviewing previous work in 2. We then present the overall system architecture in 3. Each system component is then discussed separately. The Adeos nanokernel is discussed in 4. The kernel-mode bootloader is discussed in 5. Required changes to the Linux kernel are described in 6. The virtual devices to be created are described in 7. The clustering components to be used are discussed in 8. With all the components covered, we discuss how the work on each component can be carried out in 9. Before finishing, we review some of the caveats of the method suggested and possible improvements in 10. Finally, we conclude in 11.

2 Previous work

There are many areas of previous work that need to be investigated in order to get a good idea of what needs to be done and how it needs to be done in oder to get multiple Linux kernels running as a cluster on SMP hardware. Here is a summary of the areas of previous work covered:

- Efforts to increase Linux's scalability
- SMP clusters
- Kernel emulation/virtualization
- Nanokernels
- Conventional clusters
- Kernel bootloaders

The idea of running multiple production operating systems (OSes) in parallel on the same hardware in order to achieve scalability with minimum development effort is not new in itself. The most prominent implementation example is Disco [12, 15]. The Disco approach consisted of taking a production IRIX OS and running multiple copies of it in parallel using the Disco virtual machine monitor to achieve hardware sharing. The techniques developed during this work have actually been reused in the VMWare product [1, 24]. In both cases, key hardware components, such as the CPU, the memory, and basic I/O devices, are virtualized by the monitor in order to create the illusion of real hardware accesses.

Although the approach is indeed useful for mainstream OSes which are not available in open source form, it is not very well suited to Linux. As a matter of fact, many of the basic assumptions made by the Disco team do not hold if we accept the idea that basic OS modifications are possible and should indeed be carried out in order to facilitate running this OS on top of some hardware-abstracting software.

Instead of a virtualizing monitor, Linux lends itself quite well to the insertion of a small nanokernel beneath it. This nanokernel's purpose would be to provide mechanisms to allow client OSes to request hardware resources. Instead of implementing clever hardware virtualization techniques, the client OSes are aware of the nanokernel and can therefore best use its capabilities.

The existing Adeos nanokernel is a prime candidate for use as the basis of such work. Adeos was first introduced in [26] and aimed at providing a patchless nanokernel for Linux. Given that the techniques described in the original paper were not portable, an implementation was developed as a kernel patch [14, 2]. This latest release is easily portable to other architectures than the x86. Adeos, however, still requires some changes in order to accommodate multiple Linux kernels executing in parallel in separate address spaces.

As part of championing SMP clusters, Larry McVoy discussed his ideas both privately and publicly in many instances. [18] summarizes most of these ideas from a fairly high level and [17] covers some actual implementation details. Although the argument for SMP clusters is definitely convincing, some of the implementation ideas put forth require further evaluation. Mainly, many of the efforts to plan for single system image (SSI) components may actually be alleviated by reusing existing software components instead of creating new ones.

The approach taken in this paper, for instance, is to suggest the use of existing clustering solutions such as Beowulf [3, 22] and Mosix [4] to provide SSI components. By doing so, we reuse existing software which has been tested and proven to work.[1] Such clustering software is, however, often built with the premise that nodes are physically separated using high-speed networking hardware. There are probably simplifications which can be made to these softwares' algorithms in light of the tight coupling found in SMP systems.

Also, contrary to the suggestions made in [18], it is unlikely that RTLinux, or RTAI for that matter, may be of any use to the implementation of SMP clusters. Apart from the fact that these real-time micro-kernels run Linux as their lowest-priority task, hence imposing scheduling policies to the kernel that needs to scale, they do not provide an environment for running multiple operating systems concurrently on the same hardware, nor do they provide distributed services.

As described in [12, 15], SMP clusters require the use of a nanokernel which enforces hardware sharing among all existing OS instances and provides a set of distributed services. Adeos already provides basic nanokernel services for running multiple OSes on the same hardware and should easily extend to provide distributed services.

Another suggestion was to use User-Mode Linux (UML) [5]. UML, however, requires an existing userspace environment complete with virtual memory services and a C library in order to run. For best performance, we want each individual kernel part of the cluster to run on real bare hardware. UML is not adapted to this use.

An entirely different approach is taken by the Linux Scalability Effort [6]. Many parts of this effort aim at scaling the kernel's own algorithms and capabilities to handle N number of CPUs efficiently. As stated earlier, this approach may cause problems for the kernel's maintenance.

The porting of Linux to NUMA machines is part of these efforts [7]. Such ports are likely to reap the benefits of the approach put forth here. In addition to NUMA machines, however, this approach applies to any standard SMP box available on the market.

In investigating the running of multiple Linux kernels on the same hardware, one key issue is the booting of multiple Linux kernels on the same hardware. Most existing boot code which can load an extra kernel while a kernel is already running, such as the LOBOS LinuxBIOS loader [8], make the assumption that they can rewrite over the existing kernel's own structures. Of course, this is not useful in our setup. The code most likely to be useful to booting additional kernels on top of the one already running is the current SMP boot code (arch/i386/kernel/smpboot.c). Martin Bligh's work on booting Linux on NUMA-Q is also likely to be quite useful.

3 Overall system architecture

The software architecture presented here consists of 4 main runtime components:

1. Adeos nanokernel
2. Linux kernel
3. Virtual devices[2]
4. Clustering components

The layering of these components is illustrated in figure 1. The Adeos nanokernel provides a distributed hardware sharing layer. Instead of virtualizing the hardware, the Linux kernel is modified to make key hardware requests, such as interrupt allocation and manipulation, physical memory allocation, and device interrogation, to the nanokernel. The nanokernel on the root virtual node (vnode)[3] is the central Adeos arbitrator and is responsible for providing each vnode's nanokernel with the list of hardware components the vnode's kernel is entitled to.

During the boot sequence, for instance, the root vnode kernel will conduct PCI post or extract PCI post information from the BIOS. User tools are provided on the root node to enable the attribution of the various PCI resources to the vnodes. Once this is done, the additional kernels that boot on other vnodes will make their requests for PCI listing from their local Adeos. This Adeos will communicate with the root vnode to retrieve the list of devices belonging to its kernel.[4]

The devices could also be provided by the root node as boot parameters. The use of the nanokernel, however, allows for device migration once all

[1] This paper does not attempt to evaluate the pros and cons of existing clustering software nor its ability to effectively provide a single system image. The assumption being made is that this existing software is adequate and capable of providing a clustering API along with single system image capabilities.

[2] These are virtual devices very much like the virtual devices used by Rubini and Corbet to present their examples in [23]. They are **not** *virtualized* devices as implemented by VMWare.

[3] They are called *virtual* nodes because there need not be actual physical separation between the various nodes.

[4] The current scheme does not allow for the presence of ISA devices. Such devices would be complicated to manage, especially because of the way ISA DMA operates.

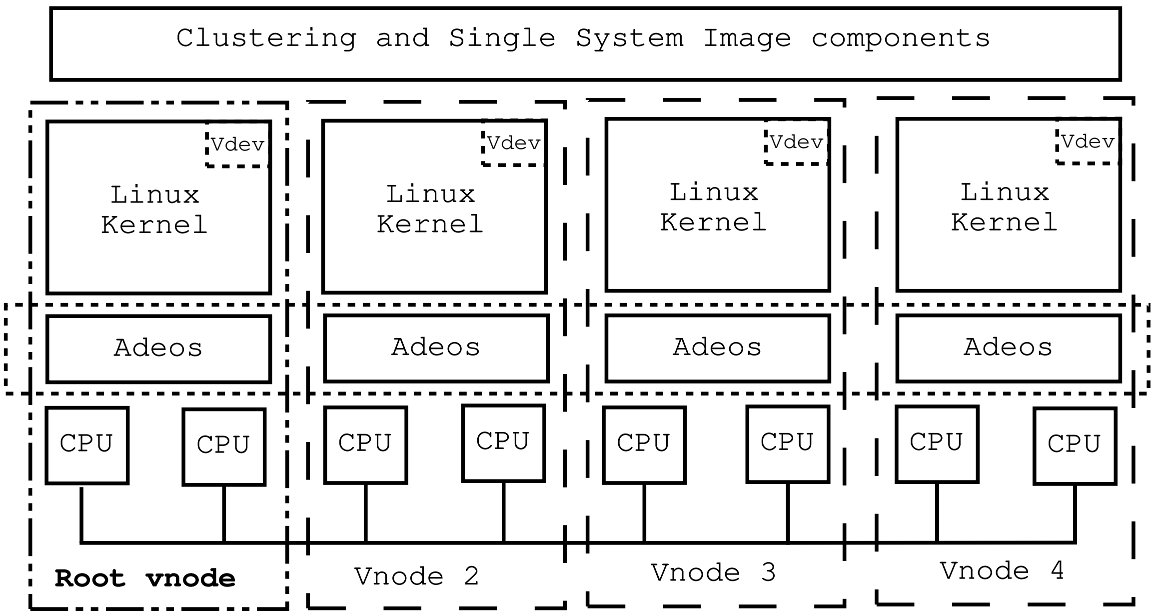

Figure 1: Overall system architecture.

devices have been allocated and the system is running. A SCSI controller can therefore be disabled on one vnode and then brought up on another vnode. Also, the nanokernel can be the cornerstone of fault-containment. When a vnode fails, the nanokernel notifies the root vnode which takes care of the cleanup and restart of the faulty vnode. Vnodes can also be debugged independently from one another. It should be fairly straight-forward to debug a kernel running on one vnode using another vnode.

Device allocation is exclusive in this scheme. A device allocated to a vnode is not shared by other vnodes. If an Ethernet card is given to vnode 3, then no other vnode can actually see this PCI device, much less use it. In addition, hardware accesses are left as-is. No changes are made to intercept or modify hardware accesses. Hence, all existing device drivers can be used as-is.

Figure 1 illustrates 2 CPU vnodes, but vnodes could be made of as many CPUs as desired. In the extreme case, 1 CPU vnodes are possible. Given the fact that there is one Linux kernel running per vnode, 1 CPU vnodes will waste physical memory resources with little advantage given that Linux already scales well up to 4 CPUs. Since Linux is already quite able to handle 4 CPUs, 4 CPU vnodes are likely to be the optimal setup.

Interestingly, vnodes can be made by an odd number of CPUs. A 3 CPU vnode, for instance, should be feasible. This could be potentially useful in the case where the system designer would prefer to allocate as little resources as possible to the root node. On an 8 CPU machine, for instance, there would therefore be 3 vnodes: the root vnode with 1 CPU, a second vnode with 3 CPUs, and a third vnode with 4 CPUs. The problem with this setup, however, is that the vnodes are not *equal*.

Each vnode runs its own copy of a kernel image common to all vnodes and has its own separate virtual address space, although all vnodes share the same physical memory.[5] It follows that each vnode has its own GDT, page tables, and interrupt table. This setup differs from the OSlet concept discussed in [17].

Communication with other vnodes is provided by the nanokernel using shared physical memory regions, which we will call *bridges*, and interrupts (inter-processor interrupts in particular), which we will call *portals* following the nomenclature developed for the SPACE nanokernel [21, 20, 19].

The allocation and management of bridges and portals is the role of the nanokernel. These basic abstractions can then easily be used to build more elaborate services using virtual device drivers. Since these devices would be implemented as separate components from the core kernel functionality, no changes are required to the kernel's code. Instead, these device drivers use the nanokernel's API

[5] The use of a single virtual memory address space for all kernels is a roadblock to scalability.

directly to interact with other vnodes. Prime candidate for such drivers are virtual network interfaces and a distributed filesystem. These services could also be used to provide SSI components, such as a unique /proc, or raw shared memory buffers.

Since these services are implemented using the nanokernel's API, Linux's own virtual memory code need not be changed in any way. Neither do the scheduler or the lock granularity. These are the key advantages of this scheme since maintainability of the kernel is not influenced in any way. Any elaborate service, such as process migration, can be implemented separately from the existing kernel code.

It follows from this that running other OSes than Linux as part of this cluster should be relatively easy. The root node could be OpenBSD, for instance, and would be the only node linked to a physical network outside the local cluster box. This will be all the more easier as the clustering and SSI components are close to userspace.

The last piece of the puzzle is the clustering and SSI components. Given that these issues have been intensively studied and worked on in the context of other projects, the best approach is to reuse existing components. Beowulf and Mosix are prime candidates for this because of their established user-base and proven reliability. Since virtual network device drivers can be implemented over Adeos' bridges, it is very likely that these packages may run unmodified on the setup described here.

It would be best, of course, to provide the capability of running any clustering solution on top of the current scheme. This would require a unified approach to clustering on Linux, however. This issue is beyond the scope of the current proposal, but Marowksy-Brée has discussed it amply in [16].

4 Adeos nanokernel

The existing Adeos code already manages interrupts on UP and SMP systems efficiently. It will require modifications, however, in order to satisfy the requirements of SMP clusters as described here.[6] The future architecture of the Adeos nanokernel is presented in figure 2. The following subsections discuss each part of the nanokernel. Note that the implementation of these services will also make the running of other kernels side-by-side with Linux much easier.

4.1 Interrupt management

Adeos' first and most important responsibility is to manage interrupts and implement a flexible scheme for interrupt handling. The interrupt pipeline described in [26] will continue to be used, as it is very well adapted to this purpose.

The interrupt pipeline will be used by some of Adeos' own components in order to implement various services. The distributed sharing protocol (DSP), for instance, is likely to use inter-processor interrupts in order to communicate with other Adeos instances. The DSP will then intercept these interrupts to receive the messages of other Adeos nanokernels.[7] Other "secret" interrupts may be used to enable userspace applications to communicate directly with add-on services. It follows from this and from the description provided in [27] that building a hard-real-time cluster should be feasible.

Instead of using the flat model for the various APICs, Adeos will need to use the cluster addressing model. Martin Bligh's work on the use of this addressing model in NUMA-Q machines should be helpful to this end. Chapter 7 of [13] should be the starting point.

4.2 Device management

The first vnode to boot is responsible for handing over its device management to Adeos. This vnode may be able to see all of the system's devices, but it still can't use any devices which have been handed out to other vnodes. Userspace tools will need to be developed to enable the administrator logged-in on the root vnode to manage the allocation of devices.

It should be feasible, for instance, for the cluster administrator to enter commands such as: Transfer device D from vnode X to vnode Y. The nanokernel will then have to bring the device down on vnode X and then bring it up on vnode Y. All such actions will interact with Adeos in order to complete and would use existing hotplugging capabilities extensively.

The programming of the various I/O APICs is modified according to the attribution of devices so that device interrupts go to the vnode which has been attributed the device.

4.3 Physical memory management

Adeos is responsible for providing all the vnodes, except the root one, with their initial physical memory.

[6] The assumption being made here is that Adeos is extended to provide additional functionality. It could be possible to leave the current Adeos codebase unmodified and build the required services on top of the basic services already provided.

[7] Most of the communication between the nanokernels is actually implemented using shared physical memory regions, as discussed below.

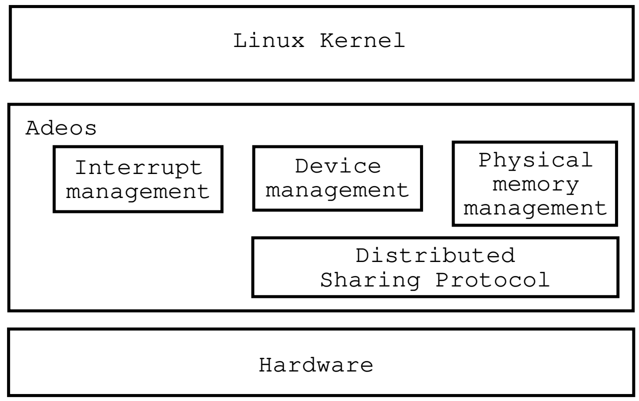

Figure 2: The Adeos nanokernel and its facilities.

The kernel-mode vnode bootloader requests physical regions from Adeos. Once the vnode is started, Adeos is then responsible for allocating and connecting bridges.

The Adeos physical memory management code will need to implement very smart policies. Specifically, it will need to be capable of defragmenting physical memory. Such defragmenting may requiring the temporary freezing of all the vnodes using a certain bridge. It follows that intensive allocation/freeing of bridges by vnodes will significantly impact on the entire cluster. This extreme situation is not typical of most cluster setups, however. In the most likely scenario, each vnode will allocate a certain number of bridges at boot time and will continue to use these bridges throughout its existence.

In order to alleviate the problems caused by transient bridges, Adeos could provide a separate allocation service for such bridges. The requester would then be required to provide a callback function for dealing with the relocation of the bridge to a different physical memory location.

Different bridges can be implement with various identification schemes. *Named bridges* would be recognized by all vnodes using a unique name. Such bridges could be local to a vnode or global to the cluster. Accessing a local named bridge requires the vnode ID, while accessing a global named bridge requires only the name of the bridge.

4.4 Distributed sharing protocol

In order to synchronize with one another, all the Adeos nanokernels need to communicate with each other using a distributed mechanism. This is the role of the DSP. The implementation of the DSP proper will require some additional research in order to implement a scalable and flexible protocol which has very low overhead. As with other components in the system, the DSP will be implemented over shared physical memory regions.

5 Kernel-mode bootloader

The kernel-mode bootloader (KMB) is in charge of the startup of the non-root vnodes. Its main interface is a command-line utility which takes the following arguments:

- Pathname of the kernel image to load.
- CPU ID[8] on which it has to start the kernel.
- Range of CPU IDs part of the vnode.
- Vnode ID.
- Size of physical memory attributed to this vnode.
- List of devices belonging to this vnode.

The kernel image provided to the KMB should actually be the same as the one running on the root vnode. Hence, all vnodes would be running the exact same kernel image. In order to automate the loading of multiple vnodes instantly, it is probably best to define a configuration file format for the KMB userspace tool. Such a format is outside the scope of this writing.

The parameters are then handed over to the kernel module component of KMB through `ioctl()` or a similar interface. The LinuxBIOS bootloader, LOBOS, passes a filename using an additional system

[8] It may be more appropriate to talk about the (logical/physical) APIC ID instead?

call. This is inconsequential since LOBOS is about to write over the running kernel. Because adding new system calls to a kernel which has to continue running is not a desired side-effect, another method is preferable.

Also, LOBOS reads the image as an executable image using `read_exec()`. This may not be appropriate for our current scheme. The kernel module indeed needs to use `lookup_dentry()`, as LOBOS does, but its subsequent operation differs from LOBOS. [9] Here are the steps carried out by KMB kernel module:

1. Request physical memory from Adeos for vnode. (This has to be a physically contiguous memory region).

2. Remap physical region into current kernel's virtual memory.

3. Copy kernel image and boot parameters to memory region. The boot parameters are: ID of CPU to boot on, list of additional CPU IDs that image has to boot, vnode ID, and allocated physical memory range/size (`mem=`).

4. Set region as executable.

5. Inform Adeos of the initial resources the upcoming vnode will be allowed to use. (PCI devices).

6. Jump to the loaded image's `startup_32` function.[10]

7. Unmap physical region from current kernel's virtual memory.

For its part, `startup_32` is modified to check very early whether paging is already enabled. If it is so, then its action differs from the existing procedure. The modifications to the kernel's initialization will be described in detail in section 6.1.

6 Linux kernel changes

In order to run in the scheme described here, the kernel needs slight modifications. The role of the kernel and most of its existing functionality remains unchanged, however. Mainly, the kernel knows of a set of PCI devices it can probe for and a single flat physical memory region it is entitled to manage freely. These assumptions are left intact by the current proposal.

There are 2 reasons for implementing changes in the kernel for the current scheme:

1. Allow insertion of nanokernel beneath Linux.

2. Enable Linux to boot differently.

3 areas of the kernel need to be changed in order to allow the insertion of the nanokernel: the PCI subsystem, the interrupt management code, and the locking primitives. The last 2 have already been successfully modified as part of the current Adeos work. The following subsections detail the required kernel modifications, starting with modifications to kernel initialization.

6.1 Initialization

At startup, the kernel usually assumes that it has total control over any available and visible piece of hardware. Linux here is no different from the majority of existing OSes. Such an assumption will have to change, however, if Linux clusters are to be possible on SMP hardware.

First and foremost, Linux will have to be able to boot in very strange physical addresses, sometimes very high-up in the physical address space. It may already be capable of dealing with such situations, but this needs to be confirmed.

Hardware initialization will also need to change in light of the fact that the kernel will only have access to the hardware the Adeos PCI proxy allows it to see. ISA-style probing, for one, should not be carried out during the booting of any non-root vnode. If necessary, then different kernel images should be used for the root vnode and the non-root vnodes.

For its startup, the root vnode should be booted with a vnode ID of 0, the list of CPU IDs part of the root node, and a `mem=` parameter equal to the amount of physical memory it should use. The rest of the physical memory actually available will be handled by the nanokernel.

The boot code in the kernel differs when it comes to the `startup_32` code. Instead of immediately setting up the CPU's table and enabling paging, the code should first check if paging is already enabled. If so, then the current image was loaded by KMB. Otherwise, the normal code follows its course.

[9] Interestingly, nevertheless, LOBOS comes with a small script which uses `objcopy` to strip the kernel image from all its headers and turn it into a bare executable. This script may not be very useful, however, since we need to be able to locate the `startup_32` symbol in the image to be started.

[10] All references to `startup_32` refer to the function in arch/i386/kernel/head.S.

In the case where paging is enabled, then the code branches off and conducts an operation similar to that found in arch/i386/kernel/smpboot.c but on itself. Essentially, the code sets the startup EIP for the vnode CPU ID to be the physical address of the beginning of the trampoline routine[11] and sends the appropriate IPI to the APIC of the vnode's first CPU. The code then returns and the KMB can continue where it left off.

Meanwhile, the other CPU has booted with a fresh copy of Linux which will proceed through the normal boot sequence since paging will not already be active for it. It will load its own GDT, its own IDT, and its own page tables. Anything it does afterwards is conducted in an entirely different address space from the first kernel.

The main references for implementing this scheme are the current SMP boot code and chapter 7 of [13].

It could be argued that the setup and boot operation of the vnode's first CPU should be the responsibility of the KMB instead of that of the kernel being booted. There are, however, advantages in having the booted kernel take care of setting up its own CPU prior to booting. Mainly, the kernel's booting on a vnode is made independent from the bootloader. A distributed fault-detection mechanism, for instance, could have kernel images ready to go in kernel space. When detecting a failure, the mechanism would fire the kernel image without requiring the intervention of the KMB. The developers of such a mechanism would need to know very little about vnode kernel booting. Also, various kernel versions with different boot mechanisms can be supported. So can other OSes.

The bootup of the additional CPUs in the vnode is done very much the same way as the SMP bootup is currently done. Some small changes may be required nevertheless.

The root filesystem could be provided either as an initrd or mounted via NFS using the virtual NIC described below. Of course, each vnode that has a SCSI controller with an attached physical HD can mount its own device. It can also share it with other vnodes using NFS or a distributed filesystem as discussed in section 7.

6.2 PCI subsystem

Linux's current PCI subsystem can either retrieve its data from the BIOS or conduct its own PCI post. In the case of a non-root vnode, PCI posting is out of the question and so is probing the BIOS to obtain the device entries.

The PCI subsystem must be modified to make its device listing requests to Adeos instead of doing them directly. In the case of the root vnode, these requests actually end up being calls to the kernel's own existing functions. On other vnodes, however, these calls are routed to the root Adeos which then hands over a list of visible devices to the vnode's Adeos. It is this list which is provided to the kernel running on this vnode.

Device transitions require the dynamic addition and removal of entries in the kernel's PCI tables. The hotplugging work comes in as being very useful here since it should be easy to reuse this functionality to implement device migration as described above.

6.3 Interrupt management

Easy access to a vnode's interrupt flow is essential for implementing many of the capabilities described here. Portals are the most basic service for which taping in the interrupt flow is necessary. Fault containment and kernel debugging are yet other reasons why interrupt interception by the nanokernel is essential.

To achieve this, we need to modify the kernel in order to divert all interrupt allocation to the nanokernel. This has already been implemented and is available in the current Adeos implementation. See [14] for more details. Basically, `set_intr_gate()` is modified to call on Adeos' interrupt allocation services instead of manipulating the interrupt table directly.

All the modifications required for this are easily surrounded by appropriate `#ifdef/#endif` statements.

6.4 Locking primitives

Since we need to receive all interrupts that come from other processors, including ones which don't belong to Linux, we need to modify the kernel so that it doesn't disable the interrupts in hardware. Instead, the interrupt pipeline's properties are put to use here. In essence, Linux's requests to disable interrupts result in a pipeline stall. Conversely, requests to enable interrupts unstall the pipeline. Remember that interrupts are not propagated in the pipeline beyond a stalled stage.

As above, the modifications required to the kernel sources are relatively small and are all conditional to

[11] This address could be passed as a kernel boot parameter if virtual addresses cannot be easily converted into physical addresses by this boot code.

enabling the Adeos nanokernel in the kernel's configuration. The complete details are provided in [14].

6.5 Timer

This is not so much a change to the kernel as it is a change of policy. There's only one 8253 in the system and it becomes very inconvenient in the current setup to try to have all vnodes use this timer. Instead, it is much more viable for each vnode to use the timers found in each CPU's APIC. Clock synchronization then becomes the job of the clustering and SSI components. It is, however, conceivable that Adeos may be modified to allow very low-level clock synchronization through its DSP.

7 Virtual devices

The virtual devices are the glue that enables all the vnodes to communicate together. These devices export known abstractions to userspace. These abstractions are themselves used by the clustering and SSI components to form a cluster. In order to best communicate with other vnodes, the virtual devices use the nanokernel's bridge and portal services directly. The following subsections discuss the key virtual devices to be implemented and provide an outline of each device's operation. Note that, as stated in [12], such devices are fairly easy to implement.

7.1 Network device

The network device most likely appears as a normal Ethernet device to the rest of the kernel. It starts by opening a named bridge, the "Ethernet" bridge, and maps this bridge into its kernel's address space. This bridge is the Ethernet "wire" that connects all the vnodes. Obviously, these devices will have to include a software implementation of CSMA/CD in order to synchronize accesses to the virtual wire.[12]

Because the virtual NICs are free, a vnode can have many such devices, each connected to a different network. Since each vnode in the cluster is connected to all such existing networks, efficient load-balancing can insure that no one network is saturated.

A similar device has been implemented as part of the examples presented in [23]. That device is rather simple, but it is certainly a good start for whoever wants to implement an Adeos-aware virtual NIC such as the one described here.

7.2 Distributed filesystem

A distributed filesystem is an important SSI component. Many such filesystems already exist. Often, they are implemented over existing network services. NFS, for instance, could easily be used on top of the virtual NIC described earlier. Given the architecture described in this proposal, however, it is likely that other, more efficient, distributed filesystems can be implemented.

In this case, a filesystem could use Adeos' services to communicate with the other filesystem components on the other vnodes. In the simplest implementation, the root vnode could be the central repository with all the other vnodes making requests to it. This is similar to NFS' operation but would involve only one software layer instead of two.

The necessity of having a unique /tmp directory was discussed in [17]. In this scheme, one instance of the distributed filesystem could be mounted at /tmp on each vnode, hence enabling a cluster-wide unique /tmp. There are, of course, other directories which can use such a filesystem. In the extreme, the entire cluster could have the same root directory.

It is also possible to implement a virtual distributed block device, but as with the MTD work where JFFS2 is more efficient than ext2 over NFTL [25], it is preferable to have an Adeos-aware filesystem than an Adeos-aware block device.

7.3 Console device

Since there is only one real console on the entire system, there needs to be a way for system administrators to communicate with the console of any vnode. This too is a service provided on top of Adeos. Each vnode would have a line discipline implemented over a local named bridge. This line discipline would serve as the vnode's console. The root Adeos would therefore be able to selectively communicate with the console of any vnode. Such a line discipline should be able to serve any vnode, including the root vnode.

7.4 Userspace shared memory regions

Some applications may be designed to make efficient use of the current setup. It may be useful for such applications to have access to the bridge facilities in user space. Although it could be possible to export

[12]This is a very good illustration of the use of inter-vnode locks. Interestingly, the scalability of the algorithm used to implement such a lock will most likely be independent of Adeos' or Linux's or even the clustering and SSI components' own scalability.

Adeos' services to user space using a different software interrupt, it may be more appropriate to encapsulate Adeos' services in a kernel device driver. Such a device driver would implement all the security and sanity checks Adeos isn't meant to take care of.

8 Clustering and single system image components

Any in-depth work on Linux clusters on SMP hardware will have to take into consideration the very large amount of work already carried on classic clusters since much of this work can be reused as-is. As a matter of fact, issues such as providing a single system image, distributed shared memory, and distributed filesystems have already been amply discussed and have been implemented many times over. As stated above, Beowulf and Mosix are prime candidates, but have a look at the Linux High-Availability project for a more detailed list [9].

As in other clustering situations, each node sees its own local kernel and a set of network services it can use to connect to other nodes.

That being said, clustering packages may make assumptions that do not hold in the current architecture. Primarily, by having nodes so close together, physical network latencies and problems disappear. Other issues such as node failure can be detected using other means than a keepalive signal.

In the simplest configuration, the root vnode can be used as the central cluster node. Each node could then export its own local process table using a local named bridge, and the root vnode would use this information to provide a unique /proc. Other configurations may be more appropriate.

It is not the purpose of this paper to try to discuss all the details of how existing clustering solutions can best use the architecture presented. Rather, the designers of such solutions are encouraged to analyze this proposal and comment on the applicability and usefulness of their systems to it.

9 Work ahead

Most of the work presented here can be conducted by independent teams. There will be a need to synchronize in order to agree on interfaces, but there is little need to start a complex project for this work. This is yet another advantage of this proposal: if most developers agree on the best way to proceed, then each team can head its way with little need for a single authority to head the rest of the effort.

It may seem that each developer would require very expensive hardware to participate in this effort. A 4 CPU machine, with 2 vnodes of 2 CPUs each, should be sufficient, however, to develop most of the system's components. Further testing will certainly be required on machines with a greater number of CPUs, but such testing can be carried out by a relatively small number of developers taking part in this effort.

Here is the list of system components in decreasing order of the amount of work required for implementation:

1. Adeos nanokernel
2. Virtual devices
3. Linux kernel
4. Kernel-mode bootloader
5. Clustering and SSI components

The nanokernel is clearly the component which will require the largest amount of work. It will have to be extended to support device management, physical memory management, and distributed operation. Device management is relatively simple since it mainly requires each vnode's Adeos to retrieve a device list from the root vnode. The physical memory management unit can probably reuse existing memory management algorithms that already implement memory defragmentation. Finally, the distributed communication subsystem can reuse existing distributed resource sharing algorithms. In the case of the current proposal, the operation of this subsystem is likely to be simplified because of the use of a root vnode.

The virtual devices can be developed independently from one another. Once the Adeos bridge API established, such devices should be relatively straight-forward to implement. For someone already familiar with the implementation of drivers for the equivalent real devices, this probably amounts to less than 2 weeks worth of work, granted the software CSMA/CD is easy to implement. The distributed filesystem may require some more work, however. A look at the state of the art in distributed filesystem research is probably warranted.

Most kernel modifications presented in section 6 are fairly trivial. Less trivial, however, is the kernel initialization. This will require someone already familiar with SMP booting. This work may require an in-circuit emulator (ICE). Alternatively, a set of ad-hoc development tools could be implemented to help in the development. It may facilitate the process to start off with 2 different kernel images. The first one would have serial port support disabled while the second would have it enabled. Work on getting the second vnode booting could then continue using another host to debug the second vnode through the serial port. Whichever scheme is chosen, note that the development of the boot process does not require the availability of the complete nanokernel. Vnodes will, of course, not be able to communicate without it, and vnodes will therefore be isolated in its absence, but this should not impede on the development of the boot procedures.[13]

The kernel-mode bootloader too should be fairly simple to implement. See the references mentioned in section 5.

Finally, the existing clustering and SSI components should be used as-is in the first generation of this scheme. Simplifications and enhancements to make full use of the locality of the cluster nodes could probably be added in future iterations of this scheme.

10 Caveats and future work

The reliance on a central root vnode is this scheme's main weakness. Any problem on this central vnode will bring the entire cluster down. Yet, it is the use of such a central vnode that makes the entire scheme very simple to implement. It would be beneficial to plan for future iterations of this scheme where all vnodes are entirely independent of one another. If any vnode fails, then it could be brought up again by any other vnode. Great care will have to be taken to ensure that such a distributed scheme does not add additional overhead, however.

Another possibility is to look into bullet-proofing the root vnode. Running additional error-handling domains on the root vnode's Adeos is one aspect that should be looked at. Running as few device drivers as possible in the root vnode is likely to diminish the risk of failure too.

Although the current approach is unlikely to be influenced by illegal virtual memory accesses on any vnode, it is definitely very fragile to raw physical memory accesses. Indeed, any illegal physical memory access by any vnode is likely to bring the entire cluster down. Properly written drivers and clean coding are the only immediate remedies. If hardware were able to cooperate in one way or another in checking for such illegal accesses, then it would be all the better.

Currently, all CPU and physical RAM resources allocated to a booting kernel are never reclaimed. Hence, contrary to Disco, there is no CPU migration possible or physical memory load-balancing. Such functionality can probably be added, nevertheless, by using the existing work on CPU and memory hotplugging [10, 11]. Actual physical memory and CPUs could then migrate between vnodes. Adeos would certainly need to be modified to take in account such migrations.

Also, only physically contiguous vnodes can exist in the current design. It may be desirable to augment Adeos to handle vnodes spanning many different physical machines. Having appropriate hardware to facilitate the mapping of physical memory regions between separated machine would certainly help in this regard. A cluster of clusters could then be feasible.

11 Conclusion

This paper has presented a scheme for implementing Linux clusters on SMP hardware. An architecture was presented to enable many existing components to interact together in order to provide this functionality. In particular, the approach suggests the use of the Adeos nanokernel to enable multiple Linux kernels to exist in parallel on the same hardware, each in a different virtual node. To this end, the kernel requires minor modifications in order to use Adeos' services instead of directly accessing key hardware resources.

Virtual devices, such as network devices, a distributed filesystem, and a line-discipline, are implemented as kernel modules. Instead of obtaining their resources from Linux, however, they use Adeos' services directly. In turn, these devices enable existing clustering and single system image components to be used with little or no modification.

In addition to being modular, each component in this scheme is relatively simple to implement. The clear advantage of this scheme is that Linux's capability to scale is unrelated to Adeos' or the clustering and single system image components' ability to handle large numbers of virtual nodes. Also, the kernel can continue to be developed independently from all the other components in this scheme.

This paper was written with the purpose of encouraging discussion on building clusters using the Linux kernel and SMP hardware. You are encouraged to make suggestions and participate in this effort in order to provide "a cluster in a PC" for the masses.

[13] ? Implement boot procedure in arch/i386/kernel/vnodeboot.c ?

References

[1] VMWare, http://www.vmware.com/.

[2] Adeos: http://www.freesoftware.fsf.org/adeos/.

[3] Beowulf: http://www.beowulf.org/.

[4] Mosix: http://www.mosix.org/.

[5] User-Mode Linux: http://user-mode-linux.sourceforge.net/.

[6] Linux Scalability Effort: http://lse.sourceforge.net/.

[7] Linux Scalability Effort: NUMA, http://lse.sourceforge.net/numa/.

[8] LinuxBIOS: http://www.linuxbios.org/.

[9] Linux High-Availability: http://linux-ha.org/.

[10] Linux Hotplug CPU support: http://sourceforge.net/projects/lhcs.

[11] Linux Hotplug Memory support: http://sourceforget.net/projects/lhms.

[12] E. Bugnion, S. Devine, and M. Rosenblum. Disco: Running Commodity Operating Systems on Scalable Multiprocessors. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, October 1997.

[13] Intel corp. *Intel Architecture Software Developer's Manual, Volume 3: System Programming Guide.*

[14] P. Gerum and K. Yaghmour. Adeos nanokernel for linux kernel, June 2002. http://lwn.net/Articles/1743/.

[15] K. Govil, D. Teodosiu, Y. Huang, and M. Rosenblum. Cellular disco: resource management using virtual clusters on shared-memory multiprocessors. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles*, December 1999.

[16] L. Marowsky-Brée. The Open Clustering Framework. In *Proceedings of the 2002 Ottawa Linux Symposium*, June 2002.

[17] P. McKenney. Larry McVoy's SMP Clusters, November 2001. http://lwn.net/Articles/4536/.

[18] L. McVoy. Scaling linux with (partially) cc clusters, July 2002. http://www.bitmover.com/cc-pitch/.

[19] D. Probert and J. Bruno. Efficient cross-domain mechanisms for building kernel-less operating systems.

[20] D. Probert and J. Bruno. Building fundamentally extensible application-specific operating systems in space. In *Technical Report TRCS95-06, Computer Science Dept., UC Santa Barbara*, March 1995.

[21] D. Probert, J. Bruno, and M. Karzaorman. SPACE: a new approach to operating system abstraction. In *International Workshop on Object Orientation in Operating Systems*, pages 133–137, October 1991.

[22] D. Ridge, D. Becker, and P. Merkey. Beowulf: Harnessing the Power of Parallelism in a Pile-of-PCs. In *Proceedings of IEEE Aerospace*, 1997.

[23] A. Rubini and J. Corbet. *Linux Device Drivers, 2nd ed.* O'Reilly, 2001.

[24] J. Sugerman, G. Venkitachalam, and B.-H. Lim. Virtualizing I/O Devices on VMWare Workstation's Hosted Virtual Machine Monitor. In *Proceedings of the 2001 USENIX Annual Technical Conference*, June 2001.

[25] D. Woodhouse. JFFS: The Journalling Flash File System. In *Proceedings of the 2001 Ottawa Linux Symposium*, July 2001.

[26] Karim Yaghmour. Adaptive Domain Environment for Operating Systems, February 2001. http://www.opersys.com/ftp/pub/Adeos/adeos.ps.

[27] Karim Yaghmour. Building a Real-Time Operating System on top of the Adaptive Domain Environment for Operating Systems, February 2001. http://www.opersys.com/ftp/pub/Adeos/rtosoveradeos.ps.