The Linux/SPARC port

David S. Miller Miguel de Icaza

December 19, 1996

1 Introduction

The SPARC port of Linux is an interesting example of how the free software development model can be made to work. The port was acomplished by a loosly coupled team of progammers over the Internet. The project delivered a commercial grade distribution of the GNU/Linux operating system at the end of the project.

The purpose of this paper is to give the reader an overview of the entire project. Some subjects are treated lightly, while the paper does go into depth where the authors found it necessary.

Topics include everything from Linux kernel issues, to userland, and even issues concerning our boot loader.

2 The Linux/SPARC time line

1994: Work begins.

Dec 1994 - sparclinux@vger.rutgers.edu mailing list, kernel is on the 1.1.xx generation.

Dec 1994 - David's first patches in the Linux sources for the SPARC.

Dec/11 1994: Bunch of mail exchange between David and Miguel for getting information on the SPARC architecture.

1995: It is starting to look like a system.

Mar/29: SPARC kernel crashes on schedule()

Apr/15: ELF to a.out conversion program, from Peter Zaitcev. Allowed hackers to work on Solaris.

Apr/26: Kernel can do 3 context switches.

May/1-6: Linux/SPARC kernels are booted from the net; Support for V2 PROMs; many bugs fixed in a rush of e-mails.

May/29: Support for debugging the kernel with kgdb is added.

Jun/2: Started coding the Lance driver; DMA and SBUS support is coming along.

Jun/9: Fork works, machine still crashes.

Jun/13: Floppy driver is on the works.

Jun/21: Code for SRMMU has been written; many implementation bugs on those chips were found and documented

Jun/24: Kernel forks 17 processes and runs them without crashing.

Jul/17 Floppy driver works.

Aug/3: We get documentation on 4m interrupts; SCSI docs; NCR Lance chips; NCR Zilog chips; Floppy documentation; bpp docs.

Aug/15: Linux kernel received his first network packet.

Aug/18: Networking stack can reply to an ARP.

Aug/22: Serial driver working with keyboard and mouse support.

Aug/27: First release of the Lance driver.

Sep/12: kernel gdb works.

Sep/15: Linux/SPARC can be pinged.

Sep/25: Linux/SPARC executes it's first system call from userland.

Oct/1: Shell prompt.

Oct/2: Kernel released with an ext2fs image for floppies that can run /bin/sh. LANCE code released; checksumming is there.

Oct/10: Merged in Peter Zaitcev's display driver.

Oct/16: Sun disk label handling; the beginning of the ESP SCSI driver.

Oct/20: Linux/SPARC runs ifconfig; route and was pinged all night long.

Oct/23: SunOS mount program works; NFS mounting is working.
Oct/29: Developers start using CVS and have accounts on vger.
Nov/1: Dynamically linked SunOS programs work.
Nov/5: Signal handling is getting into shape.
Nov/6: NFS boot changes have been merged into the Linux/SPARC kernel. Work has started on the first Linux/SPARC libc based on the Linux Libc 4.x.
Nov/10: New console code based on Jay Estabrook's code for the TGA is working; VC's work.
Nov/14: First release of the Linux libc 4.x.
Nov/16: Massive MMU fixes; Debian File utils works.
Nov/20: Ext2fs on the SPARC is using little endian file systems.
Nov/22: First release of the Linux kernel with NFS root enabled.
Nov/23: SparcLinux mailing list is open to the public. The system is self hosting (with SunOS binaries). First native a.out userland released.
Nov/29: David starts using Crashme on the port.
Dec/4: Linux Init works.
Dec/10: SunOS netscape works.
Dec/13: Work started on the devices required to get X11 to run. Work resumed on SPARC 4m support.
1996: We mature.
Jan/2: X11 booted.
Jan/6: X11 works with cg6 driver.
Jan/11: bwtwo driver.
Jan/19: First document that describes how to setup a Linux/SPARC system.
Feb/16: 4m boots. Shell prompt on SPARC Classic.
Feb/20: SPARC Classic goes multiuser; cg3 driver.
Mar/8: New SILO working
Mar/15: Linux/SPARC runs on the AP-1000+
Mar/18: First Red Hat RPMS based on the libc 4.
Apr/3: Dynamic linker works.
Apr/19: Dynamic ELF Hello world works.
Apr/24: ptrace works.
Apr/30: ELF developement kit released.
May/5: cg14 driver.
May/7: Eddie's native binutils and gcc for Linux/SPARC ELF.
May/10: Linux/AP-1000+ runs on a 64 nodes machine.
Jun: SILO is released.
Aug: Red Hat releases a beta distribution of Linux/SPARC.
Oct: Red Hat releases Colgate 4.0 with SPARC support.

3 Linux kernel changes for the SPARC.

3.1 Ext2fs changes

We kept track of the main Linux kernel while developing the SPARC port. At the time we started the SPARC port, the Linux kernel was running on the Intel 80386 class machines and on the Alpha as well. Those computers are little endian machines, while the SPARC is a big endian computer. Making the kernel endian-clean was most of the time a straightforward thing: the only real problems were the file system code, especially since we wanted to allow people to share filesystems between all Linux systems.

The file system code could have been used without changes, but the file systems would not have been compatible with file systems created on other Linux systems, so we had to make changes to the native file system code to dynamically byte-swap the on disk meta-data just to be compatible. With this change, an administrator can freely move his drives between different Linux systems without any problem.

3.2 Memory management

SPARC based machines come with a proliferation of memory management units (MMU) and cache hardware.

The initial systems were called sun4. These machines had the first incantation of the SPARC instruction set, the page size was 8K bytes, plus it has a virtually indexed, virtually tagged unified cache. This architecture is not yet supported by Linux/SPARC, but people are steadily working on it.

The second generation of the architecture released by Sun is what we know as the Sun 4c. It has a 4K byte page size. This type of machine was the first

that Linux ever ran on, specifically a small SPARCStation/SLC with 8MB of ram. Some integer math instructions found on later SPARC models (Version 8) had to be implemented in software (as part of the C library or simulated by the OS by catching the unimplemented instruction trap). The MMU on these machines is a software managed Translation Look-aside Buffer (TLB). The TLB acts as a cache of the kernel software page tables. When this cache of translations is missed, a fault occurs and the kernel must make the translation available in the TLB, possibly by replacing an existing entry. To speed up context switching between processes, the MMU keeps a number of contexts that decrease the amount of misses in the TLB due to context switching. Sun4c has a Virtually Indexed, Virtually Tagged (VIVT) on-chip cache just like the sun4. However, note that the virtual cache on the sun4 and the sun4c have knowledge of contexts just as the TLB does.

The third generation of the SPARC architecture was the Sun4m series. The 4m systems are compliant with the SPARC version 8 architecture specification from SPARC International. The page size is 4K bytes and most of these systems fully support integer math instructions (specifically, signed and unsigned multiply/divide/remainder) that previously were not (ie. on sun4 and sun4c as mentioned above). Most sun4m machines use the SPARC Reference Memory Management Unit (SRMMU) as described in the SPARC Architecture Manual [1]. This is basically just a 3-level MMU page table scheme, again with contexts just like the sun4c. The difference here is that the MMU directly reads the kernel software page tables in physical ram to perform a translation. So people familiar with the MMU found on the Intel x86 processors can think of the SRMMU as a ”3 level” Intel MMU.

Besides all these differences, SPARC systems ship with different types of address caches. The variations are great, you can get anything from a pure physical Physically Indexed Physically Tagged (PIPT) cache, to a Virtually Indexed Physically Tagged (VIPT) cache. Some of the configurations on sun4m are multi-level: split instruction and data caches (”Harvard Architecture” cache configurations), plain unified level 2 caches (which exist in isolation, and combinations of previous two). On the Viking (also known as the SuperSPARC) we have both a Hardvard split I/D configuration plus a 1MB PIPT level 2 cache. Both have advantages and both have disadvantages and all of the possible configurations needed to be supported by Linux.

The VIVT cache offers the advantage of not incurring a TLB lookup just to detect a hit. However, knowledge of contexts must be placed into this hardware so that multiple tasks can share entries in the cache. Furthermore, such a design can produce undesirable situations, even with the knowledge of contexts. Consider a page in a shared library common to two tasks. The cache knows of this piece of data by a virtual address, context pair. Since the contexts are not the same, the two tasks must have their own lines in the cache to represent and use this single piece of data. This is highly inefficient.

The next permutation of cache architecture is the virtually indexed, yet physically tagged cache. Here the virtual address decides which line the data is to be found in, while the physical page to which the virtual address maps to in the MMU decides if there is a hit or not. With this scenario, one can get away without putting knowledge of contexts into the cache subsystem. However, now a full match in the cache incurs a TLB translation to verify the tags. But in the example we gave above for VIVT caches and shared libraries, two processes can now share that one page mapping in the same cache line. Another advantage of this design, is that a physically tagged scheme lends itself more to an efficient snooping bus mechanism for Direct Memory Access (DMA) and Symmetric Multi Processing (SMP) cache coherency. The snooping mechanism needs to only watch the physical addresses on the bus for requests to the memory controller to detect hits in it’s own cache.

Finally, we have pure physical caches. A TLB translation must be incurred before even the appropriate line in the cache can be determined. One large advantage of this scheme is that a piece of data residing in a given physical page can be shared by numerous processes in the system, no matter where it happens to be mapped virtually for them. Also, the problem of virtual address aliasing which can happen if the lines are virtually indexed, cannot possibly occur with a pure physical cache.

With this in mind, the coding for the memory man-

agement routines was done in such a way that allowed the kernel to easily support different SPARC MMU chips as well as providing the hooks needed to implement workarounds for some buggy SPARC chips.

So we arranged that all of the Linux standard routines for manipulating the architectures MMU are function pointers, and all of the configurable page protection parameters are variables, not constants. These function pointers and system configuration variables are set early during the boot stage, not at compilation time. This is a good thing, because the same kernel that is used to boot the 4c class machines can be used to boot all of the supported 4m machines, something that not even Sun has added the capabilities for in their operating systems.

These changes described are transparent to the rest of the kernel, we did not have to make changes to the generic kernel code to make use of this feature. The performance penalty by doing so is minimal: on the SPARC architecture, calling a function by its pointer is a cheap operation. Also, the overhead of the code for architectures other than the type on which the kernel is currently running can be completely eliminated by using some clever ELF tricks.

This amount of flexibility is quite remarkable, especially considering the fact that a few months before this code was implemented, the memory management support code in the Linux kernel was still extremely Intel centric.

4 Input/Output

SPARC based workstations and servers have a somewhat unique method by which direct memory access is implemented. Devices do reads and writes to main memory using virtual addresses. Most systems utilize the physical addresses to perform DMA reads and writes. This scheme implies that a translation must occur somehow, and this differs from one architecture to the next.

On the sun4c, the MMU of the processor in the system is used. If a translation is valid in the processor TLB for a given virtual address, DMA can use that address as well. One of the problems here is that commonly the TLB entries are allowed to be replaced at any given time during TLB miss processing. This cannot be allowed when a device is using a set of translations. Therefore, the kernel locks down this set of translations while the device is performing the memory accesses. Under Linux we have a fixed address range in which such locked mappings are created and destroyed later when no longer needed. To handle as many situations as possible and to use resources as efficiently as possible, the true size of this area is dynamic. As more devices need more areas locked down for DMA, the area expands. And when this rush of requests subsides the area becomes smaller again.

Things are a bit more sophisticated on the sun4m class of machines. At least logically, the DMA has it's own dedicated MMU and TLB to process the translations. On the larger sun4m machines, such as the SPARCStation 10 and the SPARCStation 20, it is truly a separate MMU on the motherboard. However on some smaller systems the functionality is implemented within the processors real TLB, but this is invisible to the programmer. It uses a separate page table, and the so called IOMMU directly reads the page tables from physical memory when satisfying an IOMMU TLB miss.

One implementation strategy could be (and initially was under Linux) to keep a resource map within a fixed address range and dynamically setup mappings as they were needed. We had already implemented the abstraction necessary for this to work for the drivers and such which needed it. But the hardware could be taken advantage of much better, so we found. Linux makes every physical page in the system available from kernel space at all times. The IOMMU has the capability to do this as well, so we simply made the SRMMU and IOMMU mappings match up exactly as far as the kernel was concerned. When a DMA operation is required, it just uses the exact same address in kernel space you would use to perform loads or stores via the main processor. No resource maps, or messy things like that. The translations are there all the time ready to be used with zero overhead for setting up these translations at run time.

5 Drivers for undocumented hardware

Linux was first developed on the Intel x86 platform The specifications and information on hardware that runs on the Intel PCs is widely available and usually hardware vendors are willing to give out the information on how the system software can take advantage of it.

Actually, this was not completely the case early in the life of Linux. A significant set of hardware vendors were unwilling to release the necessary documentation to the Intel Linux developers, so those things ended up not being supported. But as time went on, these "hardware documentation hoarders" learned that having their hardware supported under Linux translated into sales. Thus the information began to flow more freely and this is less of an issue for the Intel port.

On the Sun SPARC platform unfortunately this is not the case for the most part. Sun usually does not give away the programming information for most of its hardware, so we had to use other information sources (or lots of hours spent reverse engineering the hardware via random pokes into the I/O address space and other techniques which we explore later on).

It is hoped, that what happened to the Intel port will happen on the SPARC as well. And at the present time the trend seems to be in this direction. Already many of SparcLinux developers are not only receiving full documentation for vendors expansion cards, but they are also receiving loaner hardware for the purposes of writing a driver as well.

5.1 Existing SPARC ports of other operating systems.

We consulted the source code from a couple of publicly available operating systems that ran on the SPARC at the time: the NetBSD port, the Sprite port and the Xinu port.

The Sprite operating system has some very well documented drivers for the SPARC architecture and throws some light in the darkest parts of the architecture. These sources are a pleasure to read.

At that time NetBSD had a port to the Sun4c architecture. They had drivers for common Sun hardware like the SCSI driver, the Ethernet driver, the frame buffers and the serial ASICs (on the SPARC, the Zilog serial drivers control also the keyboard and the mouse).

Theo de Raat from the OpenBSD project was specially helpful in this stage of the project while answering our questions on the SPARC architecture.

The main problem with the existing operating system ports is that those were for the Sun4c architecture, an architecture that Sun stopped producing. When we started the project, the SPARC 4m was a more prominent machine at most sites (and the UltraSPARC based machines were just around the corner!). The 4m series of machines had a documented MMU page table format, however peculiarities such as the IOMMU had to be determined before a functional port could be done.

5.2 Poking.

We did not have documentation for handling most pieces of Sun hardware, so we had to rely on the Sun header files. This, plus some poking at the machine and common sense got us most of the information.

The cgfourteen and the TCX drivers were written with no hardware information from Sun. The information for using them comes from running Sun's X server on SunOS and watching what they were requesting from the kernel (X servers usually require little support from the kernel: setting the resolution, loading a color map, querying the hardware and mapping the frame buffer and the control registers into user space).

Supporting those cards at their full potential was not easy, but by careful usage of our tools and by poking on the PROM and looking at the OS page tables with crash dump programs we were able to deduce most of the inner workings of those cards.

Discovering the workings of the cards usually involved looking at what did the PROM map when the machine booted, then poking with the PROM the registers that looked interesting, and using the PROM commands to determine where the frame buffer was. Once this was done we could write a basic driver for those cards and at least get the console driver functional.

The Sun include files for the video card sometimes gave us some ideas on what were the values that could be used on the video cards on the registers and their layout.

The cgfourteen video card uses a powerful MDI chip designed from Sun. It provides lots of features for color control, currently Linux/SPARC can only drive the video card in 24 and 8 bit modes, without taking advantage of the more advanced features on the chip.

The TCX driver (written by Jakub Jelinek) can currently run in 8-bit and 24-bit depth mode and takes advantage of the features on the cards and was also written this way.

We have also made the required modifications to MIT's X server to support our new drivers. This code is available as part of the Linux/SPARC distributions.

We plan on adding the support for the most useful features in the chips in both the kernel and the X server sources when we find out more about them. We will then commit the work back to the X Consortium.

5.3 Public information on Sun hardware.

The Sun SPARC machines used standard industry components for the most part. The Ethernet is essentially a stock AMD Lance, the SCSI is a fabrication of the NCR53C9X series by Emulex with a modified DMA interface, and finally the serial chips on those machines are stock Zilog SCC controllers. Luckily most of these vendors were more than happy to send programming information to us.

One nice thing about most SPARC machines it the OpenBoot firmware. A running joke amongst SparcLinux team is that Sun's "The Network is the Computer" catch phrase should really be "The PROM is the Computer".

The PROM firmware provides an effortless means to probe for devices present in the system. It provides a tree structured layout of the system from the bus down to the devices themselves. Each node in this device tree contains all the necessary information one would need. For example, for a device such as the Lance Ethernet controller, the PROM device node contains the address at which the I/O registers reside and the interrupt priority level this Lance uses.

6 Compatibility with other SPARC operating systems

Linux/SPARC can run binaries of the most popular SPARC operating systems: SunOS 4 and SunOS 5, this is done trough an emulation layer that makes the Linux operating system act as any of those operating systems. While SunOS 4 is a Berkeley based operating system, SunOS 5 is a System V Release 4 (SVR4) based operating system.

They are radically different and so Linux/SPARC has to use two different approaches to run binaries for those platforms. Fortunately, each operating system reaches the kernel trough a different trap, thus allowing us to provide different entry points into the kernel depending on the type of binary we are running. (Contrast this to the MIPS for example, where a single exception entry point represents a system call, however on the MIPS this problem is eliminated since different OS's use a different range of system call numbers which are in fact unique).

6.1 Common changes

Most SunOS and Solaris binaries use configuration files from /etc as well as requiring their shared libraries in /usr/lib. Unfortunately, those libraries and configuration files clash with the ones found on a GNU/Linux system. To alleviate this problem, the kernel translates all pathname lookups from a non-native binary. It does this by prepending the /usr/gnemul/sunos or /usr/gnemul/solaris prefixes to the pathname (depending on the binary being executed), and then the lookup is done; if it fails, then the regular lookup is performed.

With this, normal /etc and /usr/lib and any other directories that may be required to have non-native information can be kept clean by making non-native applications get their information from /usr/gnemul first (gnemul stands for: Gnemul is Not an EMULator).

For example, SunOS 5 programs that read the file /etc/nsswitch.conf will read instead the

/usr/gnemul/solaris/etc/nsswitch.conf. If they access for example /etc/hosts and there is no corresponding file under the compatibility directory, then the normal /etc/hosts file will be used in such a case.

The idea is to put any system specific files that are not compatible with Linux under the /usr/gnemul hierarchy, thus acting as files that override the real ones when running non-native binaries.

6.2 SunOS 4.x emulation (BSD)

Linux/SPARC has the ability of running most SunOS binaries. Stock SunOS programs like Oracle, SPSS and Netscape run without a glitch. We hope this will allow users to easily migrate to the Linux/SPARC platform.

SunOS 4 emulation is built into the Linux/SPARC kernel, yet, the cost of the emulation code is just 10 kb of object code (1200 lines of C code). SunOS 4 emulation was designed into the Linux/SPARC kernel since the beginning, this capability was used to bootstrap the system at an early stage, without a userland base. Since its very first days Linux/SPARC was running a SunOS 4 /bin/sh instead of a native Linux/SPARC binary.

This is possible thanks to a rather clean scheme of the Linux kernel used by Linus Torvalds when the kernel was being ported to the Alpha: the kernel include files were split so that all of the information (constants and kernel structures) that was exported to userland had the same values and layout as the ones presented by OSF/1 on the Alpha. This means that Alpha code will be just as happy to talk to the Linux kernel as it would be talking to the OSF/1 kernel.

Linux on the SPARC does the same for SunOS binaries: all of the userland visible constants were chosen so that they were compatible with the constants provided by SunOS. The same was done with system call numbers, with the data structures, and with the stack layout when signals are delivered to a process.

With this scheme, there is no need to translate error codes returned by system calls nor the need to convert the information within the internal data structures to satisfy existing SunOS programs.

Just to contrast: NetBSD/OpenBSD keeps all of its structures and constants consistent among ports, they have to provide emulation wrappers for most functions as well as code to translate error codes and accommodate data structures to be compatible with SunOS, thus using up to 4000 lines of code.

Even when this allowed us to start running most SunOS applications, the subtle differences between Linux and SunOS had to be dealt with a small emulation module and we had to deal with those small differences as we ran different SunOS programs: in those cases, the Linux/SPARC kernel provides wrapper routines that have to deal with the problem.

The kernel can easily distinguish the kind of binary it is running by the software trap used by the program to enter the kernel. The kernel keeps several system call dispatch tables for this very purpose:

- A Linux dispatch table that provides system calls with Linux semantics
- A SunOS dispatch table for SunOS system calls. Most of this table points to regular Linux system calls except on those cases where the semantics for the system call are different, in which case it points to the SunOS system call wrapper.
- NetBSD/OpenBSD dispatch table: not currently used.

Our convention was to put a sunos_ prefix to all of the system calls wrappers for SunOS (Linux system calls entry points have a sys_ prefix).

6.2.1 Wrappers

- termios handling: Termios as provided by SunOS is overloaded and the data structure used to pass information from the kernel to userland should fit more information that the one required by the Linux kernel, so we had to modify Linux's kernel to deal with this.
- System V IPC: SunOS has a single entry point per System IPC family of routines, we multiplex it and call the appropriate Linux system call.

- Some other wrappers that were required: The sigaction size isdifferent; mmap as provided by SunOS (modulo the MAP_NORESERVE); mount; brk does not exist on Linux; select semantics is different (actually, all BSD-derived operating systems have documented that as a bug).
- read procedures (read, readv) to get EWOULDBLOCK, EAGAIN semantics.
- Poll code was taken directly from the iBCS2 Linux project andused in our kernel.
- Multicast routines.

The idea is that whenever we had not contemplated something in our original design of the SunOS compatibility code, we just dropped a wrapper routine that did the proper translations back and forth (all of these wrapper routines are isolated in two files).

From the NetBSD port we also learned a bit about what did SunOS 4 applications expect from the kernel; the other bit we learned it from the manual pages and trough experimenting with small user level test programs that compared what SunOS4 did provide and what it required from user level applications, then we just had to code that behavior in our kernel.

6.3 SunOS 5.x emulation

Once we had picked up the operating system we were going to emulate we went ahead and did it. This is good because at the time SunOS 4.x was the most widely available SPARC operating system and is was also simpler to emulate than the SVR4 kernel.

SunOS 5.x is radically different from SunOS 4.x and the trick of keeping the same constants, structures and so on was already used to emulate SunOS 4 and could not be reused again. We followed an approach similar to OpenBSD/NetBSD in this area, these are the requirements for SunOS 5.x emulation:

- 1. Support for ELF executables.
- 2. Networking with a STREAM based networking stack.
- 3. Wrappers for most OS system calls to translate error codes and structures information.
- 4. Special stack layout at frame delivery time.
- 5. New system calls not available on Linux.

The support for Executable and Linking Format (ELF) executables was easy, the Linux operating system was already an ELF-aware operating system, and on the SPARC it is by default built with ELF support. The wrappers could be easily coded as well as making the special layout for frames, but the networking stack was not a trivial task.

Instead of coding and putting all of that code into the Linux kernel for running SunOS 5 applications, we ported the iBCS2 package from Linux/i386 to the SPARC.

The iBCS2 package provides a framework for emulating other Unices on the i386, including SVR4 variants of it. iBCS2 already has support for the system call wrappers plus error code translation, and provides a small STREAM emulation package that allows common networking application to work.

The way the iBCS2 package works is as follows: the iBCS2 package is compiled as a kernel module and it is dynamically linked to the kernel at run time with the module manipulation utilities. This allowed us to quickly test and extend the iBCS2 code since most of the time it did not require a system reboot: just unload the module, make changes to the module and reload the module. We managed to avoid most reboots if we were careful with the module during the development phase.

SunOS 5.x emulation thus required some minimal changes to the SPARC kernel:

- Making a system call entry that would transfer control to the iBCS2module when a SunOS 5 system call was issued by a userland program (there is no SunOS 5 dispatch table in the kernel, the system call de-multiplexing job is left to the iBCS2 package).
- Adding signal dispatching code that would present the stack layoutthe Solaris signal handlers expected.

- Adding low level code to the kernel that could not be placed on theiBCS2 module.

SunOS 5.x emulation as of this writing is working but it not finished. Notably SunOS 5.x Netscape does not work. Most command line Solaris programs work, as well as some simple networked programs like telnet and ftp.

6.4 Device drivers.

The device drivers on the Linux/SPARC port have been modified to provide two programming interfaces: the Linux interface (this one implements the same programming interface that Linux has on the Intel) and a SunOS interface.

The SunOS interface is present on the the keyboard, mouse and frame buffer device drivers. By providing a SunOS interface we can run SunOS binaries that need to talk directly to the device drivers (like the X11 servers) and some other Sun-specific software.

This also eased the porting of the X11 server software to Linux: The Linux port uses the same code source code as the SunOS code provided in the X11R6.1 distribution. The code was just recompiled under Linux with minor changes on the configuration.

The X server has been extended to provide Linux users with the familiar virtual console switching: this means that you can run the X server in one console and run console application on other consoles and in the future it would be possible to run several copies of the X server on the same display (this is usually done for running an 8 bpp server and a 24 or 16 bpp X server on the same machine).

7 Userland

7.1 The Linux/SPARC C library

Once we had a somewhat working kernel, work started on porting the C library that would let us provide a complete operating system environment for Linux/SPARC. At the time the work started, we had only tested the a.out executable format (SunOS compatibility required this) so this was a constraint in the first C library port.

We had to choose between three C libraries for our port:

- The GNU C library: a highly portable C library developed by the GNUproject. It was already ported to Linux on the i386. The library however was designed to provide all of the include files required by user programs and thus came with their own definitions of the errno constants, ioctl constants and data structures. This did not fit well with the SPARC port since we were not compatible in this area with the C library.

  At the time of this writing the problems with the GNU C library and the way Linux adapts to a new architecture have been fixed. Plugging GNU libc into Linux/SPARC should be an easy task now.

- The Linux C library 4: This is a port of the GNU C library to theLinux operating system that diverged back when Linux required a working C library on the i386 platform. This version of the C library is just intended to be used with the a.out executable file format and it supports the jump libraries approach for doing shared libraries. It is not being developed any further and it's use is deprecated.

- The Linux C library 5: This was the latest version of the Linux Clibrary and at that point, the support for building a.out style libraries was dropped in favor of ELF. The library is maintained and keeps track of recent kernel additions to support new features of the Linux kernel (this is also the case with the GNU C library nowadays). Both versions of the Linux C library support the adaptation of the kernel to different data structures and constants. All of the include files provided by the library end up including a kernel include file to provide the proper definitions.

Thus, we settled on porting the Linux C library version 4 to the SPARC. Changes to this library would be easily merged in the version 5 library later and we would know where to look for bugs (so, we would not be hunting bugs in the ELF loader and in the C library at the same time). The port was finished a couple of weeks

ago, and we benefited a lot from using code from the GNU C library into our port.

We provided a small freely redistributable native userland base at that point: It was completely a.out based; we did not have shared libraries support, thus we had giant executables; and there was even people interested at this point in making a distribution based on the a.out support.

At this point some minimal testing for supporting ELF executables was in the kernel, so we had to choose between using the GNU C library or the Linux libc 5 for the port. We wanted to use the GNU library but as we found out (and explained in a paragraph before), it was not ready to support our ported Linux.

Finally, we settled down to port version 5 of the Linux library with ELF support. Statically linked binaries were running two weeks after we started and then work started on providing dynamically linked executables in Linux/SPARC.

We needed to make some changes to the kernel (bug fixes, assumptions we had made), as well as changes to the Linux dynamic linker (mostly bug fixes in the SPARC port) and to the C library to support position independent code (PIC) and to consider the fact that the SPARC 4c architecture has a hole in the vm area.

The Libc changes were mostly related to the assembly coded files and since most of them were taken directly from the GNU C library, that code was contributed back to GNU.

7.2 Programs

We currently support all of the features found on the Linux/i386 port: ELF executables and libraries; dynamically linked executables; a dynamic linker that can load code dynamically and the clone() based thread model.

Porting application code written in C or another high level language code from Linux on the i386 to the SPARC is usually accomplished by just recompiling the code (at this point the SPARC port is probably the port that has more in common with the i386 version of Linux) there are a couple of caveats though:

- The SPARC is a big endian processor, while the i386 is a littleendian processor, this means that your variables are stored in memory with a different bit pattern. You should keep this in mind while porting code from Linux on the i386 to the SPARC. Most code that has been ported to the SPARC or any other big endian processor will not require any work at all.

- Intel i386 assembly language code should be rewritten: There aresome Linux userland programs that use i386 Intel assembly language, those routines will have to be rewritten in C or in SPARC assembly.

- On the SPARC, access to half-words, words and double-words on memoryrequires that the values are aligned. On the Intel, accessing aligned half-words, words and double-words is faster, but accessing non-aligned chunks works as well. On the SPARC, accessing something what is not 16, 32, or 64 bit aligned generates an exception.

  There is a exception handler in the kernel which handles such accesses, but it is much slower than accessing non-aligned chunks, so by default, for userland programs this handler is disabled and programs which do unaligned accesses get a SIGBUS signal. Programs that assume they can just fetch unaligned information need to be fixed to take this in mind. Programs already ported for SPARC or any other processor which requires aligned accesses will work without changes.

7.3 Distribution

From an historical point of view, the Linux/SPARC operating system started using a SunOS userland, later we switched to our own native a.out based distribution while using some SunOS programs and finally we settled on an ELF based distribution.

During this time, we ported and compiled several programs as we required (debuggers, system utilities). It came a point where we had the binaries but did not know how they were compiled nor which changes were made and thus the code had to be ported again for the next release and again if we were sending changes back to the maintainers of the software.

At this point we agreed to use Red Hat's package manager (RPM) to keep track of the changes required. Rpm is a tool that allows the user to drive the compilation and configuration phases of the software starting with the pristine sources.

A source RPM file contains:

- The original package as it is found in the Internet;
- Patches required to make the package configuration file andinstallation paths adhere to the Linux File System Standard (FSSTND);
- Any possible patches that fix know bugs, security holes or specificfixes for an architecture.
- Instructions for compiling and patching the original source file andthe possibility of selectively patch or run commands depending on the architecture where the RPM is being compiled.

The RPM program is used to drive the compilation and it makes sure the compilation is started all afresh each time. It won't let you package an instalable binary file if the compilation and installation did not succeed.

All of the binaries that were shipped were produced with a reproducible environment and it allowed us to keep the Linux/SPARC specific patches in one spot, thus making it easier for us to keep track of what needed to be committed back to the package maintainers.

In early August 1996, Red Hat Software released the first complete BETA Linux/SPARC distribution on Internet, it was a state of the art GNU/Linux distribution. A Debian GNU/Linux distribution is expected too. In early October, Red Hat released the first non-beta release of the Linux/SPARC port in a bootable CD with the help of the SparcLinux team.

7.4 The Linux/SPARC boot loader.

The boot loader employed on the SPARC port loader is a two stage boot loader called SILO (The SparcLinux Loader). Instead of doing a direct port of the existing boot loaders for Linux (LILO on the Intel and MILO on the Alpha), SILO takes advantage of the existence Ext2Fs library.

When SILO is installed, the first stage loader (a 512 bytes program) is written with a list of the blocks that hold the second stage loader. The second stage loader is a program that is linked against the ext2fs library. The library provides a way for applications to define the IO interface that it will use, thus making it very easy for boot loader to employ it. The SILO code is very small compared to other boot loaders for Linux and was written in a very small amount of time.

By using the ext2fs library, the loader can lookup and load any file in the file system without having to reinstall the boot loading software every time a different kernel is installed on the system.

8 Keeping up with other operating systems

One of our concerns was how fast was the Linux/SPARC port compared to the other Linux ports (the Alpha and the Intel trees were the only two ports in the main tree at that time) and how fast it was compared against the other operating systems running on the SPARC from Sun.

The reliability of the Linux/SPARC port was also important: how reliable would the port of a free operating system to an architecture with a little public information on the hardware. Linux on the SPARC suffered from the fact that we did not have the help from Sun (a completely different scenario compared to the Alpha port of the kernel).

8.1 Speeding up Linux/SPARC

We used the fine lmbench program by Larry McVoy [2] to test the performance of the operating system. It was not just used as a benchmarking tool, but also as a tool to pinpoint the weakness in the port. Those areas where the port was behind the other operating systems on the SPARC meant that we were doing something wrong, while those where we were better at showed Linux strengths.

From the figures we got from lmbench we were able to optimize those parts of the port that were not spectac-

ular, up to the point of being faster than both SunOS and Solaris on every single test.

After several passes at optimizing the kernel code, the Linux/SPARC port was able to outperform in every single lmbench test both the SunOS 4.x and the SunOS 5.x operating systems on the SPARC hardware (at least, this is the case with Linux 2.0.22 against SunOS 5.5.1).

In the same way that lmbench was used to measure the weakness in the port, the bonnie test suite was used to get the most out of the SCSI driver on the SPARC.

Linux on the SPARC is faster than any Sun Operating System release because of a number of reasons:

- Linux is a light weight operating system One of the most critical things that contributes to performance is the cache and translation look-aside buffer (TLB) footprint of the operating system. Linux being small solves the cache footprint problem in a big way. The TLB footprint problem has been worked around because of Linux's small size and a SPARC specific trick.

  The MMU's present on the sun4m/sun4d line of Sun machines possess a three level page table scheme. Using this, one has the capability to use the normal 4k sizes pages, and also larger 256k and 16MB sized pages. The average TLB on these machines has 32 or 64 entries to cache these pte's, if the entry is not in the TLB hardware has to go out to the memory bus and walk the software page tables to reload the TLB so that the translation can be satisfied.

  This miss processing is very expensive. Under SunOS and Solaris, they do not take advantage of the 16MB and 256k sized pages to map the operating system. Therefore those two systems take many misses in the TLB during even the most rudimentary trap into the kernel. However under Linux the TLB misses for the OS are quite minimal.

  For example, an average SPARCClassic with a 32 entry TLB with 24MB of memory installed: the whole kernel can be mapped (without I/O device register mappings and Lance Ethernet DMA) in three TLB entries. These 3 entries are enough to allow the kernel to access an arbitrary physical page from kernel space. [FIXME: should we remove the comparission with SunOS from here?] Compare this to Sun OS's: 3 + (24MB / 256K) + (24MB / 4K) TLB entries to map this same amount of space. For a great many number of operations, it is quite easy for an OS with this page table strategy to blow the entire user context out of the hardware TLB. Which in turn means more processor stalls (in fact many) for both the user level processes and the operating system.

- A system call entry sequence that takes advantage of thearchitecture.

  Linux takes advantage of the procedure call conventions on a particular architecture so that it can process system calls in the most expedient way possible instead of the regular UNIX way of doing things: Linux passes parameters in the registers as if it were a mere subroutine call. Other UNIX implementations instead allocate a block that holds the registers, packs them up there and then invoke the system call with this information. Then, the system call routine, needs to retrieve the parameters from this packed up block before it can use them. This is a huge advantage: much less work involved in this same operation.

- Linux takes advantages of the GNU compiler tricks and SPARC ABIspecification.

  The ABI on the SPARC reserves a couple of registers that the compiler is not supposed to use (formerly, the %g6 and placed some of the most common used variables in the kernel in those registers and the GNU C compiler takes care of fetching those registers whenever we reference that variable. A simple trick possible to the GNU C compiler.

- Linux has been extensively profiled and most often used routines wereoptimized for every single CPU tick. Especially speeding up the IP checksumming and memory copying speeded up a lot of things, especially networking latencies.

8.2 Stability

Testing the stability of an operating system is not an easy job. We used the fine Crashme program to automate the process of finding problematic areas on the kernel. The SparcLinux team did a hard work at answering the question from users and responding as quick as possible to the bug reports from the users that were either beta testing the kernel or those users that had an installed Linux/SPARC system.

In short, Crashme is a program that tries to execute random garbage code over and over. This program is known to bring down most commercial Unix operating systems (including both of the Sun operating systems).

The routine use of Crashme on the Linux/SPARC port became part of the developement cycle (up to the point of having the team leader starting Crashme from his init scripts). The Crashme helped to find lots of problems in the port that were fixed as soon as Crashme spotted them.

Thanks to having an international team of developers and support people, when the first Linux/SPARC distribution on CD went out we had a very strong port: a port that had taken only 22 months to engineer and complete (starting from scratch up to releasing the operating system on a bootable CD-ROM).

9 The AP1000+ port

Linux/SPARC was ported by the a team at the Australian National University (Andrew Tridgell, Paul Mackerras, David Sitsky and David Walsh) to the Fujitsu's AP-1000+ computer, a distributed-memory parallel computer based on SuperSPARC processors.

The work concentrated on porting Linux/SPARC to this machine and adding the support to execute parallel programs on it. Before this port was done, the machine was only able to boot with CellOS.

10 Future plans

We want to support more hardware on the SPARC architecture and support the new generation of UltraSPARC computers as well as the multi processor server class machines.

We want to add more speed improvements to our kernel and making it release the memory it is not using once it has booted.

We have also started the work to switch to the new GNU libc to be in sync with the other Linux ports.

11 Conclusions

We have achieved our goal of being as compatible as possible with the i386 version of Linux.

By providing the same set of services and APIs across different platforms, Linux will enable system integration, ease system administration tasks and promote the use of free software.

Linux/SPARC can currently run most off the shelf SunOS 4 programs without modification, thus we could think about Linux as an "upgrade", since it is being supported and it is being actively developed. It has a small and clean kernel thus making it suitable to run on low end SPARC machines.

12 Acknowledgments

The Linux/SPARC port would not have been possible without the help from all of the volunteers of the project, the SparcLinux team consists of:

- David S. Miller (davem@caip.rutgers.edu)
- Adrian Rodriguez (adrian@franklins-tower.rutgers.edu)
- Eddie Dost (ecd@skynet.be)
- Miguel de Icaza (miguel@nuclecu.unam.mx)
- David Redman (David.Redman@eng.sun.com)
- Andrew Tridgell (tridge@cs.anu.edu.au)
- Jakub Jelinek (jj@sunsite.mff.cuni.cz)
- Dave Sitsky (sits@cafe.anu.edu.au)

- Donnie Barnes (djb@redhat.com)
- Peter Zaitcev (zaitcev@ithil.mcst.ru)
- Elliot Lee (sopwith@redhat.com)
- Mauricio Plaza (mok@nuclecu.unam.mx)
- Ralph Bugg (ralp@db.erau.edu)
- Tom Dyas (tdyas@hardees.rutgers.edu)

And the support of the GNU and the Linux communities

13 Bibliography

[1] The SPARC Architecture Manual Version 8. SPARC International. Prentice Hall, 1992.
[2] lmbench: Portable tools for performance analysis. Larry McVoy and Carl Staelin, in the Proceedings of the Usenix 1996 Annual Technical Conference.