USENIX Association

Proceedings of the 4th Annual Linux Showcase & Conference, Atlanta

Atlanta, Georgia, USA
October 10–14, 2000

Linux on the System/390

Adam Thornton
Sine Nomine Associates

Abstract

This paper is meant to serve as a general overview of the Linux port to the IBM System/390 mainframe architecture. The System/390 architecture is introduced, and its history and design are briefly discussed, including IBM's operating system VM, which allows virtualization of the System/390 and hence a way to split the machine into a large number of virtual machines. The short history of Linux on the platform is then covered, and ways in which running Linux on the System/390 make sense are discussed, chief among them the possibility of running many Linux instances on a single System/390.

Differences between the System/390 port and other ports of Linux are introduced, and the necessity for a general solution to the problem of timer interrupts in a virtual environment is raised. The I/O model of the System/390 is then described, with an example comparing a Linux/390 network driver with a PCI network driver that highlights some of the idiosyncracies of the System/390. Ways that a developer can begin working with Linux for System/390 are suggested, ranging from use of an employer's existing machine through acquisition of a used development machine to Hercules, a free System/390 emulator for Linux. Finally, areas in which development help is most badly needed are spotlighted.

Introduction

The System/390 is one of the more recent targets for a Linux port. IBM's port became available in mid−December of 1999, and formal support for it was announced May 17. Further commitments have been made in the meantime by IBM. IBM is clearly heavily invested in supporting Linux on all its platforms (with the exception of the AS−400), and that most definitely includes the System/390. The System/390 port looks, from user mode, like any other Linux system; inside the kernel, and especially inside the device model, it's somewhat unusual. Although mainframes are typically very expensive computers, hobbyists and people wishing to evaluate Linux/390 without investing in a System/390 can use Hercules, a free System/390 emulator, to see Linux on the System/390 with no investment but their time.

History and Architecture of the System/390

The System/390 is the direct descendent of IBM's System/360 series, announced in 1964. The binary format of the instructions has not changed, so it should, in theory, be possible to run a program written for a 360/25 in 1965 on a modern Multiprise 3000. In practice it is more difficult, since although the instructions themselves have not changed, the interface to operating system services certainly has as operating systems have come and gone.

In the mainframe world, you will hear about at least two operating systems. OS/390 is what is usually used for heavy−duty commercial work; it's the successor of MVS, which is the successor to OS/360. Most sites doing serious data processing on the System/390 use OS/390. However, more important from the Linux perspective is VM, "Virtual Machine." VM virtualizes the System/390 hardware. The usual shell run under VM is CMS, the Conversational Monitor System. In essence, each user gets his or her own copy of a System/390, with attached peripherals, to play with. If you've used VMWare on Intel, VM is the same idea, but with 30 years longer to mature, and implemented on hardware that is friendly to self−virtualization.

VM is much more conservative of system resources than VMWare. A typical CMS installation on a large system can support between ten and twenty thousand simultaneous users, each with a unique (virtual) System/390. VM, however, doesn't have to run CMS. Because the interface it presents is, simply a System/390 as defined in the *Principles of Operation,* it can be used to run TPF, VSE/ESA, OS/390[1], and, of course, Linux.

The architecture itself is a fairly standard big−endian 32−bit design. You will often hear System/390 operating systems referred to as 31−bit: this refers to the amount of memory immediately addressable. This is 31 rather than 32 bits because the 360 and 370 architectures were 24−bit designs, and to maintain compatibility when XA ("eXtended Addressing") mode came along, the top bit in a word was set high to indicate XA rather than 370 mode. Additionally, the ESA architecture allows for up to 16Tb of expanded storage, which is accessed by a fancy version of bank switching. Linux on the System/390, however, is still restricted to a 31−bit address space for the time being. The largest usable machine appears to be 1919M

[1] This is a very common use of VM: you can test a new version of OS/390 under VM and work out the migration bugs so when you upgrade the production machine, there is very little downtime of the primary system.

(which is 2G minus 128M minus 1M, but I don't know what significance that has).

If you need a larger machine, currently the only way to go beyond 1919M is to use the XPRAM driver to put swap space in expanded memory (which can lie beyond 2G). This limitation will vanish at the end of this year, when IBM introduces its successor to the S/390 line. It is code−named "Freeway", and not much is known about it (at least by me) other than that it is a 64−bit architecture, and it will presumably include compatibility modes to allow older software to continue to run. A 64−bit Linux port is expected almost immediately on its introduction. Therefore, don't expect to see much action from Boebligen on fixing the 1919M limitation; the recommended solution, if you need that much real memory, will probably be to buy a 64−bit engine.

You've probably heard of EBCDIC. It is the character encoding used by traditional mainframe operating systems. Its primary feature is that it is not ASCII; interoperability with ASCII systems therefore requires translation tables and is a bit of a headache. Linux/390, however, is not (unlike Unix System Services, a platform to make porting Unix applications to OS/390 and VM easier) EBCDIC; it's a plain old ASCII system. As far as the hardware is concerned, it's all just ones and zeroes. The only place you see the EBCDIC translation is in the console driver, as the console expects to see EBCDIC characters appear on it.

The System/390 has two classes of instruction: problem mode and supervisor mode. These correspond to user and kernel states, and allow for privileged instructions which cause a machine trap when a user−mode program tries to execute them. The instruction set is very, very rich: this is an extremely CISCy machine (the current 390 Principles of Operations, or architecture definition, is the size of a telephone book). For this reason, and because IBM provides a very good macro assembler, a great deal of programming on the System/390 is still done in assembly language rather than a high−level language.

History of Linux on the System/390

Linux on the System/390 is an idea that has been being kicked around since Linux's earliest days, but not much was done until 1998 or so. Linas Vepstas and others began a port of Linux, called "Bigfoot", which was an implementation that ran on System/370 (the 390's predecessor) and later processors. By early December 1999, Bigfoot would boot and usually load /bin/sh before panicking and crashing.

In mid−December, IBM Boebligen released its port of Linux to the System/390. The IBM port has significant differences from the Bigfoot port. Most notable is that it actually runs. However, it does require that you have a second−generation (G2) or later CMOS System/390 machine, as it uses certain halfword immediate instructions introduced with the G2 architecture. Peter Schulte−Strack has released a set of "Vintage" patches which allow Linux to run on a G1 machine, which opens the door to a wider variety of processors. However, the IBM port is certainly never going to run on a 370.

While Bigfoot was a vastly interesting project, and was developed as a proper Open Source project rather than as a skunkworks secret endeavor, it appears to be dead, or at least in stasis. All further mention of Linux on the System/390 will assume the IBM port.

The System/390 port was originally bootstrapped by writing a cross−compiling backend to GCC that produced System/390 code, cross−compiling glibc for the 390 architecture, and then uploading those and the kernel built with them to a real System/390. At this point there are two distributions of Linux for the System/390, so it is no longer necessary to build a cross−compilation environment: simply allocate enough disk space, copy Linux/390 onto it, IPL (Initial Program Load, or IBM for "boot"), and go.

Operational Characteristics of the System/390

The System/390 has more I/O capacity than any other computer on the planet. IBM has spent 35 years evolving this line of computers to support enormous databases with rock−solid reliability.

However, the System/390 does not have particularly good CPU horsepower. That's not to say you don't get reasonable performance out of one: they have quite a bit of oomph, but in terms of MIPS per dollar, Intel or Alpha beats them hands−down. They certainly are not cheap, and if what you need is computation rather than I/O, then it makes no sense to run Linux on a System/390. Further, floating−point support on pre−G5 models is not IEEE floating−point. Since Linux expects IEEE floating−point, those instructions must be emulated for earlier processors, which causes a very noticeable speed reduction on anything requiring much floating−point[2], most notably KDE.

However, much of the time, CPU performance is not what is needed, and I/O is. Web hosting is the obvious

[2] Those old enough to remember Linux on the i386 or the i486SX remember the pain of emulated floating−point.

example; in fact, pretty much any sort of e−commerce application is going to have far more significant I/O needs than CPU needs.

One of the features of the System/390 is the ability to split a single physical machine into many virtual machines. This can be done in hardware, with a facility called LPAR ("Logical PARtition"), which allows up to 15 machines to be carved out of a single computer. It can also be done in software. Traditionally, this has been done by running VM.

David Boyes ran a test under VM (VM itself was running in an LPAR of a medium−to−large System/390, from which it could use a maximum of 10% of the machine's cycles) in which he brought up 41,400 simultaneous Linux images before the virtual machine ran out of resources. Although this number is not representative of a real workload (it was Apache serving static pages only), there is a customer running over 3200 Linux machines (as of August 5), in production, on a single System/390.
On August 2, IBM announced VIF, the Virtual Image Facility, which is to all intents and purposes a stripped−down VM, with the ability to run multiple virtual machines, but with the sophisticated monitoring and resource allocation tools removed. VIF would let you run hundreds or thousands of images; however, it would not let you allocate resource caps to particular machines, so it would not be possible to keep one greedy Linux user from affecting performance for the rest of the Linux images on the machine. Under VM it is easy to cap each machine's resource usage, and to change those caps on the fly.

For OS/390−only shops, Linux has been reported to run under ISX, which is essentially a virtual machine facility for OS/390. ISX is not a complete implementation of a System/390: notably, it doesn't do SIE, and therefore cannot run VM, but it implements enough of the architecture to run Linux.

Why would you want to run Linux on the System/390? There are several good reasons. First is that VM (or VIF) gives you the ability to run thousands of virtual machines of the same piece of hardware. For an ISP or ASP, the savings in terms of facilities and management costs quickly overcome the higher initial cost of the System/390 hardware: if you are building a data center based around Suns, your crossover point is around 25 servers; with Intel, it's more like 150 servers. In either case, this is a small fraction of the several thousand machines VM can run with acceptable performance.

Networking many virtual machines is much easier than managing a physical facility. All you need to do is to pick a machine that owns the actual network interface, or define one machine per interface if you prefer. This machine has traditionally been the VM TCP/IP virtual machine (that is, the virtual machine in charge of VM's TCP stack), but there is no reason it couldn't be an OS/390 LPAR or, indeed, a Linux virtual machine. That machine is then set up as the gateway, providing route information via routed about the machines behind it. If you preferred, and if your network was simple enough, you could, of course, use static routes. The virtual machines then run with point−to−point connections over virtual channel−to−channel connections or over VM's Inter−User Communications Vehicle (IUCV). vCTC speeds are about 250MB/s, and IUCV about 500MB/s, sustained.

For certain webhosting models, the ability to bring a new server online for a customer within 90 seconds at a marginal cost of very nearly zero is a compelling argument. For commercial hosting you're probably going to want to run under VM rather than VIF, so that you can guarantee SLAs by preventing resource hogging. On the other hand, if what you want to do is server consolidation without guaranteed resource limits, VIF would be ideal. For instance, you might want to take a bunch of departmental servers and place them on a single box under unified management, while preserving departmental autonomy by giving each department its own virtual machine.

A university (universities are traditionally good places to find underutilized mainframes with pre−existent VM licenses) might want to run an advanced operating systems or networking class where you can give each student his or her own machine. The student can have absolute control over that machine, and thus you can do interesting systems programming assignments without the risk of a traditional shared machine setup and without the cost of giving each student a physical machine. The University of Nebraska is, in fact, doing exactly this. Academic clustering research can also benefit from Linux under VM, since it's a lot easier and cheaper to bring up multiple images on the same physical box (assuming the mainframe is already installed at the site) than it is to wire a bunch of machines together.

Finally, it even sometimes makes sense to run only a single Linux image on the System/390. This might be the case if your shop runs a database on OS/390 (or VM) already. By putting up Linux, either in an LPAR or under VM, you gain all the benefits of running Apache as your Web front end: it's free, it's easy to find staff with administrative skills, you can use Perl, Python, PHP, or even ASP as a scripting environment, and, best of all, the network connection between your Web server and your database runs at memory speed

over vCTC or IUCV.

Any multi−tier application that depends on a System/390 as its back end benefits from this approach. This does not have to be a web server/database combination. For example, if you had a POP or IMAP server residing on your mainframe providing mail services to users, you might want to implement spam and relaying protection in your SMTP MTA. While you could code this from scratch, sendmail already knows how to do it: it might be much more efficient to run sendmail in a virtual machine to provide this protection and then hand the remaining mail off to your native mail server.

The only reason I can think of to run Linux native on a real System/390 as the only operating system would be if you have a very low−end System/390 as a development machine (assuming the P/390 console problem has been solved), or if you have a System/390 which has been decommissioned, and thus you no longer have OS/390 or VM for it, but want it to do something useful until it develops a hardware problem. Of course, if you're using Hercules, and thus getting an emulated System/390 for free, Linux makes a lot of sense; this is really a special case of a low−end development box.

The Kernel

The kernel is pretty much under the control of IBM Boebligen. However, it also largely works, so there really isn't very much to do.

There are two outstanding issues, solutions to both of which are rumored to be under development by IBM.

The first, and most significant, concerns the timer interrupt. The default value of HZ on the System/390, as on all other architectures except the Alpha, is 100[3]. In normal operation, an interrupt handler executed 100 times a second is not a problem. However, consider the case where you have 5000 virtual machines all executing at once. Then you're needing to service half a million interrupts each second, and that eats heavily into the amount of processor time available to get useful work done. In fact, both for the 41,400 image test and for the 3200 virtual machines in production, the value of HZ has been set to 10. This means that interactive performance is atrocious, although since the production servers are running only INN and bind, their performance is still adequate for their workload.

The right answer to this problem is to have something like `#ifdef RUNNING_VIRTUAL` in the kernel, which disables timer interrupts in the idle task, and upon receiving a real interrupt, sets the jiffy count by querying its hypervisor to ask for the real time of day. This would not just be a VM fix: it is equally applicable to Linux under VMWare, under Plex86, and to User−Mode Linux.

The second issue involves PAGEX support. Currently, when Linux pages, the current Linux task is suspended while the right page is brought into memory. PAGEX comes from the VM world, and would allow the Linux machine to signal to its hypervisor that the machine is page−faulting. That would not only enable us to do away with one level of paging (VM's paging is much more sophisticated than Linux's), but would enable VM to more efficiently schedule its tasks, which might include multiple Linux machines.

System/390 support was integrated into the main kernel tree as of 2.2.14. Whether it works right out of the box is another matter. 2.4 support should be along Real Soon Now. The kernel development keeps tickling hitherto undiscovered bugs in GCC and glibc, and high optimization levels and thread support are still a little buggy. The assembly syntax used owes a lot more to GCC than to the mainframe, so old−time mainframers may feel a bit out of place, but anyone who has worked with assembly code under GCC will feel right at home, once the new instruction set is digested.

The only truly odd part of the System/390 architecture to the developer steeped in PC culture is the I/O subsystem.

System/390 I/O

System/390 devices are attached to channels, which are essentially device busses. Traditionally channels were parallel bus−and−tag architectures, supplied on huge cables[4], but this has been supplanted in the past decade by fiber optic ESCON channels. In any event, they've always been very fast compared to their contemporaries. One way to think of System/390 peripherals is as SCSI on steroids. For an I/O

[3] I've heard that HZ changes to 1000 in 2.4, at least for the Intel processor. This would be a very bad idea for the System/390, especially if it is running a virtual environment, as you will see.

[4] The term given to reconfiguring physical devices by rewiring them to different channels, typically underneath the raised floor of a machine room, was "poking the boa" (as in, "I can't go have a beer with you Friday; I have to poke the boa."). One of the leading channel cable manufacturers called itself "Anaconda." The names are pretty accurate, in terms of the cable size.

operation, the host processor issues a command to the device (possibly a single command, but more likely an entire channel program), and the device proceeds with its work asynchronously. Then once the device has completed the operation, it will send an interrupt to the processor stating that it's finished its task. Channel commands are documented in *Principles of Operation* as well. Each device has its own subchannel, and up to 65,536 devices are attachable simultaneously.

This makes writing device drivers for Linux/390 more akin to writing SCSI drivers than to writing any other sort of device driver. It's certainly a black and esoteric art, which requires intimate knowledge not only of the Linux device driver structure (Allesandro Rubini's book *Linux Device Drivers* is very helpful here), but of S/390 channel architecture and device design.

A description of the abstraction of the System/390 device model and the API that device driver authors should write to is found in *kernel−source*/Documentation/s390/cds.txt, and is required reading if you're going to be working on device drivers. To summarize: the IRQ does not exist as such on the System/390. However, to modify as little of the extant (mostly x86−based) code as possible, the developers decided to map System/390 subchannels onto IRQs. Basically, instead of being restricted to 16 IRQs, as you are with the ISA bus and its derivatives, under Linux/390, you get 65,536 IRQs to play with.

However, bolting the System/390 I/O model onto Intel assumptions is not without peril. Perhaps the most confusing piece to people who write Intel device drivers is that the dev_id parameter has been reused. While on the Intel architecture, it is used to specify multiple devices sharing an interrupt, on System/390 dev_id serves as a shared buffer to let the generic interrupt layer and the device−specific driver communicate about the status of the interrupt. There are some other differences that can easily bite the developer; because disabling an interrupt on the System/390 actually means telling the device "do not accept any more interrupts" rather than masking a bit on the PIC, disable_irq() is problematic, since we would like it to be able to return an error condition. Thus it and enable_irq() are defined to return int rather than void, which is Intel's behavior.

All actual work done in the device driver is done through the do_IO() interface; the device driver may not directly issue commands on its own. Aside from that, writing a device driver is pretty straightforward: disable the interrupt you're currently handling, schedule a bottom half if it's going to be a long−running process, do your work, reenable interrupts. Of course, you should try to avoid setting the DOIO_WAIT_FOR_INTERRUPT flag, which turns on synchronous processing, since that will cause other CPUs to spin in an SMP environment, and few production System/390s are uniprocessor[5].

Take, as an example, the difference between how bits are put on the wire with *kernel−source*/drivers/net/ne2k−pci.c and *kernel−source*/drivers/s390/net/ctc.c. The PCI driver uses a function called ne2k_pci_block_output (Listing 1), which messes with a bunch of setup (including some little−endian ugliness), sets ei_status.dmaing to indicate the DMA channel is busy, then writes a bunch of 16−or−32 bit chunks, depending on whether it's in 16 or 32−bit mode. It does this with the outsl or outsw functions. Finally, it resets ei_status.dmaing and exits.

On the other hand, the corresponding function in ctc.c, which describes I/O through a 3088 channel interface (which is what is emulated in a vCTC), is called ctc_tx (Listing 2). It too does a lot of setup, mostly related to whether or not the device is currently busy. If it isn't, the spinlock is set to assure synchronization across multiple CPUs, and device transmittal status is set to busy with ctc_test_and_setbit_busy(). The packet to be transmitted is moved into the lp field of the driver's privptr data structure (of type ctc_priv, basically a representation of the 3088's state), and a whole pile of status is set via the channel[] data structure to represent the I/O occuring on this device. The actual write is anticlimactic: do_IO() is called, with privptr supplying all the commands and data, which has the effect of calling WRITE with the contents of lp being put on the wire. Then the busy flag is turned off, the spinlock is unset, and the function returns.

In short, writing device drivers is kind of a mess. Fortunately, there are a few examples in the kernel source tree under *kernel−source*/drivers/s390, and some support functions can be found in *kernel−source*/arch/s390. Be warned that many of the Boebligen developers do not believe in comments; it's not easy going. Once you have device specifications, so you know what channel commands to issue to achieve the desired results, it's not particularly difficult, or at least, no more so than writing device drivers for any other architecture usually is.

Some devices have their channel interfaces well−documented. In that case, writing the device driver is a matter of implementation. It may be a little

[5] Indeed, the uniprocessor support in Linux/390 is still somewhat buggy; it is recommended to build an SMP kernel even for uniprocessor systems because of some unwanted interaction with the network drivers on non−SMP systems.

technically difficult, but there's nothing especially groundbreaking about it. For devices with proprietary interfaces, you're potentially in trouble. If you really feel a need to write a device driver for an undocumented interface, then you'd better hope you're running under VM. If you are, then VM's debugging facilities will let you trace and record input and output to the device, and you can begin to reverse−engineer the communication protocol that way. One good example might be the Shared File System (SFS) for VM. IBM published a set of Rexx scripts that allow you to do SFS over TCP/IP. Armed with that knowledge and a packet sniffer, it would not be very hard to determine what the proprietary SFS communication protocol really consists of. Of course, you can always mount SFS filesystems via the VM NFS server, so the utility of the time investment would be questionable at best.

User Mode

There's really not much at all to say here. It's Linux. There's no EBCDIC in sight. It looks, tastes, feels, and smells like Linux. Porting user−mode applications is generally trivial; if the application does not depend on hardware that isn't available on the System/390, and if it doesn't make any assumptions about byte order, it should, and almost always does, just compile and work. For now it's necessary to patch config.sub and config.guess to recognize Linux/390; autoconf will eventually include system definitions for it, and it's a two−line patch to each file in any case.

How Can I Play?

There are several approaches to developing for Linux/390. Some require access to a real System/390, and some don't.

The easiest, of course, is if your organization already has a System/390. If you're running VM, then it's probably not going to be a problem to allocate a 64M class G (general user) machine with a few hundred megabytes of disk. If not, then it might be a little tougher to get an LPAR, but if you request it and have it happen at your next scheduled maintenance window, it shouldn't be impossible.

If you work at a well−funded shop serious about getting into the Linux/390 market, then you really should invest in a VM license. The ability to create multiple images as well as the debugging and monitoring facilities make development a great deal easier.

If you don't have a System/390, then you might want to acquire one. The smallest machine, currently, is the Multiprise 3000. Even through the IBM Partners in Development program, it's going to cost you $65,000 or more. You may have more luck on the used market. A PCI−based P/390, which is a System/390 on a full−sized PCI card, cost me $5000. MCA versions are cheaper (as little as $800−$1000), but require an MCA PS/2 or RS/6000 to run, which I didn't have. Not all hosts are suitable for a P/390, but the IBM PC Server 325 was popular on Onsale.com last year, and works just fine (the 330 and the 500 were the officially supported platforms). Expect to pay $700 or so for one. Then you need an OS/2 license for the host machine (the P/390 is hosted by OS/2 on the PC, AIX for the RS/6000); however, Warp Server in the shrinkwrap is available for next to nothing from the usual auction sites. In essence, you're looking at about $6000 for the hardware and necessary glue code. Make sure that your P/390 comes with the Licensed Internal Code that supplies the System/390 microcode, or what you have is a very expensive, albeit pretty, paperweight, not a computer.

There's also a commercial software solution: Fundamental Software publishes Flex−ES, which is a System/390 emulator for Intel. Their prices start at about $15,000, but on a high−end Intel system, you can get quite a bit more CPU power than you can from a P/390.

By the time you read this, Linux/390 will almost certainly be booting natively on the P/390. As of the time of writing (early August) there were still some minor problems with the console driver that required setting a hardware breakpoint and manually clearing the registers to get Linux to boot without VM on the P/390.

A VM license will set you back many thousands of dollars. If you're doing this on the cheap, don't go there. However, there's one more solution, which is very slow, but has the great advantage of being free.

Roger Bowler wrote (and Jay Maynard now maintains) Hercules, a System/370 and System/390 emulator for Linux. It emulates the hardware well enough to boot Linux/390 (and, unlike ISX, well enough to run VM), and there exists a device driver (essentially a dummy network driver) for the host system that can make the emulated System/390 think it has a 3088 CTC connection to the host, so you can run TCP/IP applications to talk to the host (e.g. to install SuSE via NFS). The Hercules license allows non−commercial use for no charge.

Hercules is quite slow compared to the alternatives, but on a fast PC it has begun to approach the speed of a first−generation P/390. And, of course, it doesn't

cost anything. It's certainly the best evaluation platform available, although it may not be sufficient for serious development work. At least you can get a feeling for what Linux on the System/390 looks like and whether it's something you're interested in pursuing.

You could also build a cross−compilation environment and port software that way. But that's no fun, it's impossible to test in the absence of a System/390, and with Hercules available, there's really no reason to do so.

One other thing to try, if you're serious about Linux/390 work: call IBM. They made me a very attractive deal on a development system, and threw in a VM license. They badly want people developing for the platform, and are certainly serious about making life easier for their developers.

Installing Linux/390

Once you have a G2−or−later System/390, either real or emulated, with a few hundred megabytes of disk space, then you need to choose a distribution. At the moment your choices are Marist College's sort−of−Red−Hat based Linux or SuSE. TurboLinux is on the way but is not yet available.

Basically, this is like installing any other Linux. You prepare the disk space you're going to use (if you're using CMS minidisks, you need to format and reserve them within CMS before booting Linux), boot either from tape or, under VM, the virtual card reader, set up your network devices so you can get to installation media on other machines, use mkfs to build new filesystems (currently ext2 only, although SuSE may contain support for reiserfs by the time you read this), and put the files onto those filesystems. In the case of the Marist distribution, this means unpacking a giant tarball and then editing the configuration files in place; for SuSE, it's just like every other SuSE installation: mount the CD−image somewhere it's accessible via ftp or NFS, select the packages you want, and install.

What Can I Do?

Most of the effort right now needs to focus on device drivers. There are two glaring needs at the moment.

First, Linux desperately needs an open−source network device driver. Because IBM still realizes substantial profits from licensing its OSA−2 network interface design to other companies, the ethernet driver is object code only[6]. Work is being done on an Open−Source device driver to speak the CLAW protocol, which would enable Linux to talk to a channel−attached router (e.g., any Cisco 7xxx router). Hercules can use the ctc driver to talk to a virtual network interface on the host. However, if you want to make a difference fast, write a very dumb, but open−sourced, 3172 network driver for Linux/390. This would require reverse−engineering the 3172 communication protocols and implementing a large enough subset to let you put bytes on and take bytes off the network.

The second need is presumably being addressed by IBM. Although Linux/390 can boot from a tape, it cannot use tape devices. Until there is native tape support, it is difficult to implement a backup solution for Linux/390, although another avenue of approach might be to write or port an Amanda client for VM or OS/390. However, as with any device driver development, familiarity with both Linux and with the hardware platform is necessary.

There are a number of nonessential "wouldn't it be nice ifs." Channel−attached printer support is one such. However, it's not nearly as necessary as a tape driver: lpr works fine, and lpd implementations exist for VM and OS/390.

Although recent versions of mainframe operating systems have finally incorporated TCP/IP, much of the mainframe world still relies on SNA (Systems Network Architecture) to do its networking. There is a Linux−SNA project, and although it is not yet incorporated into Linux/390, it is sponsored by TurboLinux. Since TurboLinux has announced its intention to produce a Linux/390 distribution, it would be surprising if Linux−SNA weren't included. If you know both SNA and Linux, and would like to see Linux (whether on the System/390 or on Intel) penetrate further into traditional mainframe shops, this would be an excellent area to explore.

If you intend to run Linux under VM, improved access to VM facilities (particularly debugging and performance monitoring), would be wonderful. Although Neale Ferguson has written hcp, which allows you to issue CP commands from Linux/390, there's a lot of work still to be done. This too, obviously requires deep acquaintance with the System/390 architecture and with VM's services. If you're trying this route, a CMS minidisk filesystem driver would be a good place to start (even though you can access the data over NFS already).

The System/390 is one of the most reliable pieces of hardware on the planet, with MTBFs of 60 years for recent 9672s. However, Linux high−availability

[6] This is legal because the driver is loaded as a module and not actually linked into the kernel.

software support lags pretty far behind that of Solaris. Red Hat has announced Piranha, an Open−Source software HA project. I'm working on bringing it to Linux/390, and any work you can do on it will help out not just Linux/390 but Linux as a whole.

If you want to work on really bleeding−edge High Availability stuff, consider the "VM Stun" project of David Boyes and Perry Ruiter. VM already supports clustering for multiple boxes in the same location cabled together. Global clustering would stretch that cable. The essence of the idea is that a virtual machine state, and its meta−information (size of the machine, privilege class, attached devices, and so on) would be frozen, the entire package would be shipped over the wire to some other location, and then the package would be thawed and paged into the address space of the new host machine. Saving the virtual machine state is easy, but saving, shipping, and restoring the meta−information is much more difficult.

We're discovering new GCC and glibc bugs almost daily. If you're more adventurous than I am, you could work on these. And, of course, if there's any particular piece of software you want, that doesn't yet exist for Linux/390, then you're free to port it yourself. Most applications require nothing more than a patch to config.sub and config.guess and then a recompilation. Doom took me about 20 minutes to port. Quake's a little harder, and because it relies on OpenGL (System/390s do *not* have accelerated video), is probably going to be unacceptably slow, even on a fast processor. Before you expend the energy, check out SuSE and the Iron Penguin project to see whether someone else has already ported your application. The odds are good that it's already been done.

If it hasn't, and if you aren't using autoconf to generate an appropriate system definition, it's been my experience that the Linux PowerPC port of anything is usually the right place to start. It's 32 bits, it's big−endian, and it usually doesn't use any inline assembly. Because it's Linux, we can assume GCC and GNU make and the usual GNU tool layout. Once you can specify these parameters in the Makefile, the battle is just about won. High levels of optimization are still pretty buggy, so turning optimization down to −O1 is usually the right thing to do. If even that generates code that doesn't quite work, −O0 may be required. These problems should go away as the bugs get shaken out of the compiler and libraries.

Conclusion

Linux/390 is an exciting and fun platform to develop for. From a user−mode perspective it's not very exciting, since everything behaves as it should and there's little surprise. However, writing device drivers is challenging, and writing tools to interact with a traditional System/390 environment, be that VM, OS/390, or VSE, is difficult and engrossing. The System/390 for the first time puts Linux on machines with the potential for extremely high reliability and enormous I/O throughput. VM (or VIF) gives the ability to run thousands of Linux images on a single piece of hardware, which not only is a cost−effective solution for the ASP/ISP market, but is also a great opportunity to do clustering research much more simply than with a traditional, discrete−machine setup.

Resources

http://linux390.marist.edu
http://www.s390.ibm.com/linux
http://www.vm.ibm.com/linux
http://www.linux390.com
http://linux.s390.org
http://penguinvm.princeton.edu

Listings

Listing 1: ne2k_pci.c, output of bytes to network in ne2k_pci_block_output()

```
   /* Now the normal output. */
        outb(count & 0xff, nic_base + EN0_RCNTLO);
        outb(count >> 8,   nic_base + EN0_RCNTHI);
        outb(0x00, nic_base + EN0_RSARLO);
        outb(start_page, nic_base + EN0_RSARHI);
        outb(E8390_RWRITE+E8390_START, nic_base + NE_CMD);
        if (ei_status.ne2k_flags & ONLY_16BIT_IO) {
                outsw(NE_BASE + NE_DATAPORT, buf, count>>1);
        } else {
                outsl(NE_BASE + NE_DATAPORT, buf, count>>2);
                if (count & 3) {
                        buf += count & ~3;
                        if (count & 2)
                                outw(cpu_to_le16(*((u16*)buf)++), NE_BASE + \
                                      NE_DATAPORT);
                }
        }

        dma_start = jiffies;
```

Listing 2: ctc.c, moving of packet into privptr and output of bytes to network in ctc_tx()

```
        (__u8 *)lp = (__u8 *) &privptr->channel[WRITE].free_anchor->block->length \
                              + privptr->channel[WRITE].free_anchor->block->length;
        privptr->channel[WRITE].free_anchor->block->length += \
                  skb->len + PACKET_HEADER_LENGTH;
        lp->length = skb->len + PACKET_HEADER_LENGTH;
        lp->type = 0x0800;
        lp->unused = 0;
        memcpy(&lp->data, skb->data, skb->len);
        (__u8 *) lp += lp->length;
        lp->length = 0;
        dev_kfree_skb(skb);
        privptr->channel[WRITE].free_anchor->packets++;
        if (test_and_set_bit(0, (void *)&privptr->channel[WRITE].IO_active) == 0) {
              ctc_buffer_swap(&privptr->channel[WRITE].free_anchor, \
                  &privptr->channel[WRITE].proc_anchor);
                privptr->channel[WRITE].ccw[1].count = \
                  privptr->channel[WRITE].proc_anchor->block->length;
                privptr->channel[WRITE].ccw[1].cda   = \
                  (char *)virt_to_phys(privptr->channel[WRITE].proc_anchor->block);
                parm = (__u32) &privptr->channel[WRITE];
                rc2 = do_IO (privptr->channel[WRITE].irq, \
                  &privptr->channel[WRITE].ccw[0], parm, 0xff, flags );
                if (rc2 != 0)
                        ccw_check_return_code(dev, rc2);
                dev->trans_start = jiffies;
        }
```