Redbooks Paper

Gregory Geiselhart
Vic Cross

Linux on IBM eserver zSeries and S/390: Porting LEAF to Linux on zSeries

Preface

This Redpaper describes a port of the Linux Embedded Appliance Firewall (LEAF) to Linux on zSeries. LEAF can be used in a specialized implementation for routing in a large-scale Linux on zSeries installation (sometimes colloquially known as a “penguin colony”).

Objectives

The objectives of this paper are:

- To discuss what LEAF is and how it works
- To show an example deployment method of LEAF on zSeries which takes advantage of centralized management using z/VM

This paper does not discuss how to configure and operate LEAF, but instead examines porting issues that may applied to other Linux projects requiring a small footprint.

Important: This paper is not a recommendation on how to build Linux router images on zSeries. It is a description of one way to do so, using a purpose-built distribution of Linux.

In the IBM Redbook *Linux on IBM @server zSeries and S/390: Large Scale Linux Deployment*, SG24-6824, the authors describe a method of cloning Linux systems which will work for any type of Linux system, not just routers. However, LEAF offers some advantages (as used on zSeries under z/VM) that make it attractive for the particular task of centrally managing Linux router guests.

If anything, the paper reinforces the fact that almost anything you can do on Intel Linux, you can do on zSeries Linux!

What is LEAF

LEAF is a project derived from the original Linux Router Project (LRP). LRP described a way to implement a customized Linux distribution which was small enough to fit on a floppy disk, and yet provide advanced routing functionality such as traffic shaping, firewalling, and even DHCP and DNS service.

Both LEAF and LRP are designed to be as small as possible, primarily for reliability and security reasons. The small footprint of LEAF/LRP makes it ideal for recycling old PC hardware for use as routers.

There are a number of different LEAF and LRP sub-distributions, each with a different focus or objective. For instance, the LEAF-Bering distribution (on which this paper is based) was the first LEAF distribution to use the Linux 2.4 kernel, in order to take advantage of the Shorewall firewall and other enhancements.

For more information about the LEAF project, and links to documents about how LEAF works on the Intel platform, visit the project home page. There is a lot of information on the site, including user and developer guides, installation instructions, FAQs and HOWTOs, and testimonials. The LEAF Project home page can be found at:

`http://leaf.sourceforge.net`

How LEAF works

As mentioned, LEAF is designed to be as small as possible, in terms of both code size and system requirements. LEAF builds a filesystem entirely in memory, and runs the Linux system from that - this eliminates the need even for a hard drive. The boot process reads package files from some persistent medium (usually a floppy disk, but can be a CD-ROM or network server) and builds the filesystem in memory from the contents of those packages.

Once the filesystem is built, the package store is no longer required and can be removed. This is a valuable security feature, as a potential cracker can only corrupt the filesystem in memory; if a compromise is discovered or suspected, a reboot of the router will return the system to a known state.

LEAF components

The main components of LEAF are *packages*, similar to packages such as RPM or DEB used by other distributions. In keeping with the low-overhead nature of LEAF, a LEAF package is basically just a tar archive with a particular format.

The LEAF developers build special kernels, with a small configuration to reduce the footprint (the 2.4.18 kernel used in the LEAF Bering distribution on Intel is just under 900kB in size, compared to nearly 2MB for the zSeries 2.4.7 installation kernel). LEAF also uses an initial root device (initrd) which provides the bootstrap facility that gets the router configuration started.

A key component of the LEAF distribution is a software utility called BusyBox. BusyBox is a program described as a “multi-call binary”, which includes the function of dozens of separate programs as shipped with a normal Linux distribution. Using BusyBox for LEAF considerably reduces the amount of space required for the distribution.

Boot process of LEAF on i386

When a LEAF router on the i386 platform boots, a series of actions take place that lead to the eventual running of the router system. We will describe the process in some detail here, so that we can later compare the differences between the i386 process and the zSeries process.

Syslinux boot loader

The computer’s BIOS locates the boot devices, and loads the master boot record (MBR) from that device. In this case, the code is part of the syslinux utility. Syslinux reads its configuration file to determine options such as the command line to be passed to the kernel. In the case of LEAF, there are additional

parameters passed in the kernel command line that are actually ignored by the kernel, but picked up by the LEAF initialization script later in the boot process. Syslinux loads the initial root device and the kernel image into memory, then starts executing the kernel.

Linux kernel

The kernel is decompressed (if needed) and executed. The kernel then decompresses and mounts the initial root device, and executes the system configuration script.

Linuxrc

This is the system configuration script. The actions taken by this script can vary between LEAF distributions, but usually at least the following actions will be performed:

- Loading packages, from disk or other method such as TFTP
- Loading kernel modules (if required)
- Activating network interfaces
- Creating RAM disks for temporary or log files
- Other actions, such as creating symbolic links for BusyBox

The script parses the kernel command line to obtain some configuration information, such as which packages to load and where to load them from.

/sbin/init

The last stage in the bootup process is to start the init process, which reads the /etc/inittab file and starts other processes (such as console gettys) as required.

/etc/init.d/rc

This script reads links from the /etc/rc?.d directories (for the desired runlevel) and runs the linked programs in the configured order. This process is how system daemons such as httpd and cron are started.

/sbin/getty

The getty processes are started by init, and manage the connected terminals. They usually run the login program, which handles the process of user login.

The last three stages in this process are similar to the startup process of a “normal” Linux distribution.

LEAF configuration system

The method used to configure a LEAF router is the **lrcfg** utility. This is a program that allows an operator to make configuration changes to the package files on a LEAF router. Each package contains a definition file which **lrcfg** uses to determine which files in the package are configuration files that can be edited. **lrcfg** launches an editor that allows the operator to make the required changes to the configuration file.

When editing configuration files using **lrcfg**, only the version of the file that resides in LEAF's in-storage filesystem is changed. To make the change part of the configuration of the router, the LEAF boot medium must be available for **lrcfg** to write a new version of the modified package with the changes in place.

Tip: This is a valuable security feature. Without access to the boot medium, it is impossible to make inadvertent or malicious permanent changes to the LEAF configuration.

The main **lrcfg** panel is shown in Figure 1.

```
                    LEAF configuration menu

       1 ) Network configuration

       2 ) System configuration

       3 ) Packages configuration
                                     b) Back-up a package

                                    c) Back-up your LEAF disk

                                     h) Help
  q) quit
------------------------------------------------------------------------------
       Selection:
```

Figure 1 lrcfg main panel

LEAF in the penguin colony

We now introduce the LEAF router concept to zSeries Linux. We discuss how "z/LEAF" differs from Intel LEAF, describe the process we used to build a z/LEAF system from scratch, and how our z/LEAF system works in practice.

Why use LEAF in a penguin colony

When running a large-scale Linux on zSeries installation, it is important that the infrastructure overhead is kept as low as possible. This means that not only CPU utilization, but also disk and storage consumption, must be minimized.

LEAF's design fits this model very well. Designed originally to work on low-end Intel hardware, it can run in as little as 8MB of RAM and be entirely contained on a single 3 1/2" floppy disk. On zSeries, this low-overhead design makes it an interesting candidate for use as a router in a large-scale Linux deployment. Combined with features of z/VM, a collection of LEAF routers can be managed from a central location.

Differences between Intel and zSeries LEAF

Overall, the structure of the LEAF installation has not changed. There are a few areas that were slightly altered for LEAF's use on zSeries.

Boot loader

On Intel LEAF, the Syslinux bootloader is used to load the kernel (with the desired parameter line) and provide the initial root device. However, Syslinux is not currently available on zSeries Linux, so we need an alternative.

The processing of Syslinux is quite similar to the boot process used by the installation system kernels used by the major zSeries Linux distributors. These kernels have an integrated loader which reads the kernel, parmfile and initial root device into storage from either a tape device or a z/VM reader.

Since this is an established process in zSeries Linux, we have used the "z/VM reader and installation kernel" boot method to replace Syslinux.

Kernel

As mentioned previously, the kernel being used is the kernel as supplied with the distributors' installation system. This kernel has the following advantages:

- Support for initial root device (initrd) built in
- DASD driver loaded as module

Accessing DASD using the DASD module is a desirable feature that provides additional security. By attaching and detaching the configuration disk as required, we can be reasonably sure that undesirable changes to the configuration cannot be made.

Location of LEAF packages

We were unable to wait for the completion of the zSeries floppy disk drive support[1], so our LEAF implementation utilizes a Linux ext2 filesystem on a shared z/VM minidisk for storing the LEAF package files. This disk is attached during the running of the /leafrc script, and detached at the end of the script.

Bringing LEAF to zSeries

In this section, we describe the process we used to port LEAF-Bering to Linux on zSeries.

Kernel

We used the SuSE 2.4.17 professional kernel for initial development. After initially working with tuning the build, we decided that using a 64-bit distribution gave no appreciable benefit, and in fact added to the size of the build. Therefore, we changed to a 31-bit SuSE distribution. Table 1 lists the improvements we were able realize in the size of the image:

Table 1 Space saved by switching to a 31-bit distribution

	64-bit	**31-bit**
Kernel file	3.22MB	2.57MB
Initial root device	6MB	4MB
Storage required for IPL	24MB	16MB

However, due to other considerations, we had to increase the storage allocation to perform functional work with the system. This is described in “QETH driver memory allocation” on page 17.

[1] Of course, this is a statement made in jest. However, on a system such as the Multiprise 3000, which provides emulated device support, it would be possible to define the machine’s floppy disk drive as an accessible device, perhaps by creating a 2880 block emulated FBA disk on it. Since the kernel file would occupy more space then this, however, it would not be practical for this application.

Leafrc

The leafrc script, which functions as a system pre-build script to install the LEAF packages and configure basic devices, required some work to allow for the way that z/LEAF is set up.

As discussed, we used a shared minidisk (formatted ext2) to store the LEAF package files. Rather than give the LEAF guests permanent access to this disk, we chose to control access by having the guest link the disk when it is required. In order to do this, we used the cpint utility.

The leafrc script has modifications to install the cpint utility, as well as to install the DASD driver and mount the package disk. Once packages are installed, the disk is unmounted and unlinked.

LEAF packages

Not all of the LEAF packages available with the Bering distribution have been ported to zSeries at this time. At present, the following packages have been built and tested:

- root
- local
- etc
- tc

To assemble these packages for zSeries, we simply obtained the package list from the existing i386 package file and reproduced the list using files from a zSeries system. For many packages (local and etc, for example) no changes were required since they contained no i386 binaries.

Running LEAF on zSeries

In order to get the most benefit from the LEAF packaging method, we can take advantage of features of z/VM to manage the LEAF routers centrally.

Centralized boot files

Normally when you use the installation system to build a Linux system on z/VM, you will transfer the kernel, parmfile and initial root device to the A disk of your Linux userid and execute an exec that punches the files to the reader and IPLs. If we used the same approach with our LEAF routers, we would have these files scattered all over our z/VM system, making it difficult to keep releases synchronized.

Instead, we will keep the files on a central userid A disk, and punch the files from the central userid to the reader of the LEAF router we wish to IPL. Then, we use the XAUTOLOG command to log on the LEAF userid. Adding an IPL command to the LEAF userids' directory entries will automatically IPL the card reader when the guest logs on, commencing the LEAF boot process without any other intervention.

The simple CMS EXEC shown in Example 1 gives an example of how the LEAF router users can be automatically punched and logged on.

Example 1 LEAFIPL EXEC - sample LEAF IPL script

```
/* */
PARSE ARG LEAFID;
IF LEAFID = '' THEN DO
  SAY 'Usage: LEAFIPL rtrid';
  SAY '  where rtrid=the userid of the LEAF router to be IPLed.';
  EXIT 1;
END;
'PURGE 'LEAFID' RDR ALL'
'SPOOL PUNCH 'LEAFID' RDR'
'PUNCH SLES7PROF KERNEL A (NOH'
'PUNCH LEAF PARMFILE A (NOH'
'PUNCH INITRD LEAF A (NOH'
'XAUTOLOG 'LEAFID
```

We used this process quite successfully during early work on z/LEAF. When we started looking at centralized configuration management (see "Centralized configuration process" on page 10), however, we realized that we would need two support guests: one running Linux to support the configuration process, and another running CMS to provide the IPL processing. This seemed wasteful, considering that the CMS guest would get very little use.

While we were considering this, we learned about the Linux "unit record" device driver. This driver allows a Linux guest to interact with the VM spool; in particular, it would allow us to keep the kernel, parm and initrd files in a Linux filesystem, and punch to the LEAF guests' readers from Linux. Now, we could use a single management guest running Linux for configuration and starting the LEAF images.

We copied our CMS IPL exec to Linux, and made some alterations to it to allow running it on Linux[2]. Example 2 on page 10 shows the resultant REXX script.

[2] Yes, perhaps we should have re-written the exec as a shell or Perl script, but with the shell scripting skills of our team we decided it was easier to install Regina REXX than to do the re-write (even though there are only six lines that require re-writing). Perhaps in a future revision of this paper it may appear as shell script...

Example 2 leafipl.rexx - sample LEAF IPL script

```
#!/usr/bin/rexx
PARSE ARG LEAFID;
IF LEAFID = '' THEN DO
  SAY 'Usage: LEAFIPL rtrid';
  SAY '  where rtrid=the userid of the LEAF router to be IPLed.';
  EXIT 1;
END;
'/sbin/hcp PURGE 'LEAFID' RDR ALL'
'/sbin/hcp SPOOL PUNCH 'LEAFID' RDR'
'ur copy -b sles7.kernel /dev/vmpunch'
'/sbin/hcp CLOSE PUNCH'
'ur copy -t 'LEAFID'.parmfile /dev/vmpunch'
'/sbin/hcp CLOSE PUNCH'
'ur copy -b initrd.gz /dev/vmpunch'
'/sbin/hcp CLOSE PUNCH'
'/sbin/hcp CH 'LEAFID' RDR ALL KEEP NOHOLD'
'/sbin/hcp XAUTOLOG 'LEAFID
```

Tip: For more information on the unit-record driver and corresponding utilities, refer to the discussion on `ur` in *Linux on IBM @server zSeries and S/390: Large Scale Linux Deployment*, SG24-6824.

Centralized configuration process

On i386 LEAF, configuration changes are made by logging on to the LEAF router and running the **`lrcfg`** program. This makes changes to the in-memory filesystem of the LEAF router. If the changes are to be persistent, the relevant package(s) are backed up to the boot medium.

On zSeries LEAF, we want the configuration process to be done entirely from a central server. In order to do this, we change the processing of configuration updates slightly.

On our central configuration machine, when a configuration change is required for a LEAF router, we expand the package files into a temporary space and run the configuration changes against those expanded files. When the change is complete, we back up the altered packages, which writes the updated package files ready for the next restart of the LEAF router.

If a dynamic configuration change is required, we run an additional command that signals the LEAF router to re-link to the configuration disk and re-expand the altered package(s).

Note: You could use the `lrcfg` program on each LEAF router to make temporary changes. This is feasible, but be aware that by doing so you are increasing the chances that a configuration change is made without being saved.

Router-unique packages

Since the LEAF IPL process includes the packages to be installed in the kernel command line, one of the things that must be unique for each LEAF router is the kernel parmline. This allows a different set of packages to be specified for each router, as required.

Some packages, however, contain router-unique files. The best example of this is the etc package, which populates the /etc directory of the LEAF router. Each router will have different versions of common configuration files (such as /etc/hosts) which must be managed on a per-router basis.

The easiest solution is to duplicate these particular packages for each router, and specify the name of the package file for that particular router on the kernel command line. Using the LEAF router image name as part of the package name makes it easier to identify which packages have been customized for a particular router.

Starting a z/LEAF router

We now describe the process of starting a z/LEAF router image.

z/LEAF boot sequence

The LEAF boot sequence on zSeries is roughly similar to the boot process on i386, except for the initial startup. Instead of using the syslinux utility as a bootloader, the installation kernel's internal loader is used to read the kernel, parmfile and initial root device from the z/VM reader into storage, and start executing the kernel.

Once the kernel is executing, the boot process is almost identical to i386.

Getting the LEAF system into the reader

Before we can IPL a LEAF guest from its virtual reader, the correct files must be punched there. We do this from a central management guest, as described in "Centralized boot files" on page 8.

Starting the z/LEAF guest

Once the files are in the guest's reader, we can log the guest on and have it IPL from the reader. Since we want this process to be fully automatic, we want the IPL to take place automatically. There are two ways of getting the guest to IPL at logon:

- IPL command in the guest's directory entry
- PROFILE EXEC on the guest's A disk containing an IPL command

The second option would be used if you wanted the ability to IPL the LEAF router guests into CMS and perform operations there. This is not necessary, since we do not expect to use CMS and have no other reason to give the guest an A disk. Therefore, we will ensure that the directory entry for the LEAF guests has the correct IPL command to start the guest from the virtual reader.

Having set up the guest so that it will automatically IPL when it is logged on, we now need to provide a way for the guest to get logged on without intervention. This is done using the XAUTOLOG command at the end of the IPL exec.

Note: Remember that the userid you intend to use to log on other guests must be authorized to do so! The AUTOLOG statement in a user directory entry lists those users who are allowed to use XAUTOLOG to log that guest on. We made sure that each of our z/LEAF guests had our configuration guest ID listed in the AUTOLOG statement.

Running the z/LEAF guest

Once the z/LEAF guest is logged on and IPLed, no further action is necessary. It will perform its boot processing and start functioning automatically.

Tip: As with any unattended Linux guest, it is a good idea to have centralized logging. Under VM, the easiest way to do this is to spool the guest's console to another userid. There are network-based methods (remote syslog, etc), but spooling the console allows you to get message output from the very start of the guest's IPL.

Building z/LEAF

The process of building the z/LEAF system originally involved a lot of manual command work. We started with a LEAF Bering system on Intel, to gain a working understanding of the boot and configuration processes. Then, we moved the initial root device to our Linux on zSeries system, and gradually replaced files from the LEAF system with the equivalent packages from the zSeries Linux installation.

BusyBox

The majority of the function of LEAF (outside the kernel and modules) is provided by the BusyBox utility. BusyBox implements a large number of system utility functions in a single file called a “multi-call binary”. Packaging many different programs in a single binary is used for embedded projects such as LEAF for three main reasons:

- Saving code space through non-duplication of code

  Every executable file contains some common code that is used to load and execute the program. Combining many programs into one file (compared to the same programs in their own command file) means you have only one copy of this common code, used by all the programs.

- Saving file system space through reduced filesystem overhead

  Most computer filesystems have some inefficiency due to the way that file space is allocated (in multiples of the filesystem block size, 4KB on zSeries). This means that, unless the size of a file is an exact multiple of the block size, some space will be wasted at the end of every file. If many programs share a single file, part of only one block will be lost instead of one for every file.

- Reduced code size due to elimination of unnecessary functions

  Most software utilities have a large number of features and functions compiled in, which leads to a large executable file. Many of these functions might not be used in the embedded application environment. When making a utility part of a multi-call binary, the packager will usually take the opportunity to remove some unnecessary options, thus reducing the file size.

The makers of BusyBox take many common Linux/UNIX utilities and package them in a multi-call format. BusyBox can be customized so that the various utilities are either included or excluded from the resulting binary.

Building BusyBox on zSeries was not difficult. For the purpose of this project, we left out some components (such as modprobe) which were i386-specific code. To build BusyBox, we issued the the command **`./configure && make`**.

Note: In the case of modprobe, it was much easier to simply bring a copy of the binary over from the host distribution than to try and port the platform-specific code in BusyBox to zSeries. Of course, that increased the size of our resulting system.

The biggest difficulty in building BusyBox was in making sure that the components included in the build of the BusyBox binary matched up with what was required by LEAF. This was made easier with the realization that the components required would appear as symbolic links to the BusyBox binary on a LEAF system. All we had to do was list the directories /usr/bin, /usr/sbin and so on, looking for all the symbolic links and making the appropriate configuration in the BusyBox header file.

Other system applications: SSHD

Once we had networking set up, we of course wanted to log on to our LEAF routers over the network! To do this, we needed to provide a Telnet or SSH daemon to support a logon interface. Following the process used earlier in our system build, we simply copied the files from our source SuSE distribution into the root package of our LEAF system.

When we rebooted our router and attempted to connect using SSH, it was unsuccessful. Starting the syslog server did not provide any information for us (other than the log of the connection request by inetd). Running sshd from the command line yielded the following error message:

```
sshd: error while loading shared libraries: libpam.so.0: cannot load shared
object file: No such file or directory
```

The SSH daemon as supplied with our source distribution was linked against the PAM libraries, which we had not copied to our LEAF build. Our first thought was to copy the libraries in and try again, but we consider PAM to be too much overhead for a simple router image (which will not have users logging on in the general course of events). So for now, console access via the VM TCP/IP stack will do.

Using z/LEAF

Our z/LEAF environment consisted of the following guests:

LEAFMSTR was a SuSE Linux Enterprise Server 7.2 system

0191 disk: CMS A disk (original work prior to Linux ur device driver)
0200 disk: package files for all LEAF routers

0201 disk: swap (200 cyl)
0202 disk: root filesystem (3138 cyl)

LEAF routers

0200 disk: temporary read-only access to packages

In practice, we came across a few catches when using LEAF, most concerning storage (which is the primary issue when working with in-storage Linux distributions like LEAF).

Storage allocated in 8MB units

We wanted to be able to finely control the amount of storage given to our LEAF routers. We were making changes to the size of the initial root device to make it fit in a smaller size, so as to reduce the overall footprint of our router images. Using DirMaint, we altered the initial storage size of our LEAF guests in 4MB intervals. However, when we IPLed our LEAF images, Linux did not recognize the storage change.

Figure 2 on page 16 shows an example; the command output comes from one of our LEAF guests, LEAF2.

```
# hcp q stor
STORAGE = 28M
Ready;

# dmesg | grep Memory
Memory: 19620k/24576k available (1924k kernel code, 0k reserved, 617k data,
48k init)

# cat /proc/meminfo
cat /proc/meminfo
        total:    used:    free:  shared: buffers:  cached:
Mem:  21348352 15400960  5947392        0  4206592  7376896
Swap:        0        0        0
MemTotal:        20848 kB
MemFree:          5808 kB
MemShared:           0 kB
Buffers:          4108 kB
Cached:           7204 kB
SwapCached:          0 kB
Active:            704 kB
Inact_dirty:     10608 kB
Inact_clean:         0 kB
Inact_target:       36 kB
HighTotal:           0 kB
HighFree:            0 kB
LowTotal:        20848 kB
LowFree:          5808 kB
SwapTotal:           0 kB
SwapFree:            0 kB
```

Figure 2 Storage allocation on LEAF2

We seem to have lost 4MB! Once we changed to 32MB, things were different, as seen in Figure 3 on page 17.

```
# hcp q stor
STORAGE = 32M
Ready;

# dmesg | grep Memory
Memory: 27664k/32768k available (1924k kernel code, 0k reserved, 617k data,
48k init)

# cat /proc/meminfo
total:    used:    free:  shared: buffers:  cached:
Mem:  29585408 25669632  3915776        0  4214784  7430144
Swap:        0        0        0
MemTotal:        28892 kB
MemFree:          3824 kB
MemShared:           0 kB
Buffers:          4116 kB
Cached:           7256 kB
SwapCached:          0 kB
Active:             60 kB
Inact_dirty:     11312 kB
Inact_clean:         0 kB
Inact_target:        4 kB
HighTotal:           0 kB
HighFree:            0 kB
LowTotal:        28892 kB
LowFree:          3824 kB
SwapTotal:           0 kB
SwapFree:            0 kB
```

Figure 3 Storage allocation on LEAF1

This apparent 8MB boundary behavior reduced our ability to finely tune our storage allocation.

QETH driver memory allocation

Before we started using networking on our LEAF images, we found that we could successfully get them IPLed with a storage size of 24MB (anything smaller was insufficient space to complete the init processing). This gave us sufficient storage to start some daemons and verify from the console that things were working. Once IPLing was sorted, the next stage was to get the routers on the network. We included the QDIO and QETH modules from our source SuSE distribution, IPLed our system, configured chandev, and loaded the modules. Then, when we tried to start daemons, our system froze.

Suspecting memory contention, we re-IPLed and checked memory usage before and after module loading. We found that the storage consumed by the QDIO driver was not significant (in contrast to the QETH driver, which has significant storage consumption).

```
# cat /proc/meminfo
        total:    used:    free:  shared: buffers:  cached:
Mem:  21348352 15400960  5947392        0  4206592  7376896
Swap:        0        0        0
...
# insmod qdio
Using /lib/modules/2.4.7-SuSE-SMP/qdio.o
qdio: loading QDIO base support version 2 ($Revision: 1.79 $/$Revision: 1.44
$)
debug: qdio_setup: new level 2
debug: qdio_labs: new level 2
debug: qdio_sense: new level 2
debug: qdio_trace: new level 2
# cat /proc/meminfo
        total:    used:    free:  shared: buffers:  cached:
Mem:  21348352 15536128  5812224        0  4206592  7376896
Swap:        0        0        0
...
# insmod qeth
Using /lib/modules/2.4.7-SuSE-SMP/qeth.o
qeth: loading qeth S/390 OSA-Express driver ($Revision: 1.135 $/$Revision:
1.53 $/$Revision: 1.18 $)
 qeth: allocated 0 spare buffers
...
qeth: Trying to use card with devnos 0x700/0x701/0x702
qeth: Device 0x700/0x701/0x702 is an OSD Express card (level: 2938)
with link type Gigabit Eth (portname: NIC0700)
 qeth: IPv6 not supported on eth0
# cat /proc/meminfo
        total:    used:    free:  shared: buffers:  cached:
Mem:  21348352 19714048  1634304        0        0  7376896
Swap:        0        0        0
...
```

Figure 4 QETH storage utilization

The memory utilization of our LEAF guest has jumped by over 4MB. Once we then tried to start various daemons, the system ran out of memory.

One way around this would be to set up swap devices to give Linux some swap space. While there is nothing wrong with this, it does increase the complexity of the guest's configuration, and the idea is to keep things as simple as possible.

We could also allocate some expanded storage to our LEAF guests. The usual way for Linux to handle expanded storage is with the xpram device driver, which makes the expanded storage available as a block device. This is then used to define swap space. We now end up with a variation of the previous solution, with no appreciable advantage to the guest over just allocating the extra storage as more central storage.

The 4MB[3] requirement of the QETH driver is documented elsewhere. We mention it here because in this situation, since it is a low-memory implementation, it was an issue that caused a significant effect to our solution.

DirMaint files in our readers

Every now and then, we would find that an IPL of one of our LEAF routers would fail. On accessing the console output, we would see the following message:

```
HCPGIR450W CP entered; disabled wait PSW 000E0000 00000232
```

This is an IPL failure wait-state that indicates that no IPL-able image was found. This was very confusing, since the first command we issue in our IPL exec is:

```
'/sbin/hcp PURGE 'LEAFID' RDR ALL'
```

We assumed this command would clear the reader of any files. We checked that all of our punch commands process successfully.

After one such failure, we IPLed CMS in our LEAF guest and issued the RDRList command. The command output is displayed in Figure 5 on page 20.

[3] Actually 8MB in total, since 4MB comes from the guest and another 4MB comes from CP.

```
 LEAF2  RDRLIST  A0  V 164 Trunc=164 Size=4 Line=1 Col=1 Alt=0
Cmd Filename Filetype Class User  at Node     Hold Records  Date      Time
    LEAF2    VMLINUX  PUN A DIRMAINT VMLINUX  NONE      10 OPEN-    000C
    (none)   (none)   PUN A LEAFMSTR VMLINUX  NONE   33709 10/02     0:26:35
    (none)   (none)   PUN A LEAFMSTR VMLINUX  NONE       3 10/02     0:26:42
    (none)   (none)   PUN A LEAFMSTR VMLINUX  NONE   15096 10/02     0:26:42
```

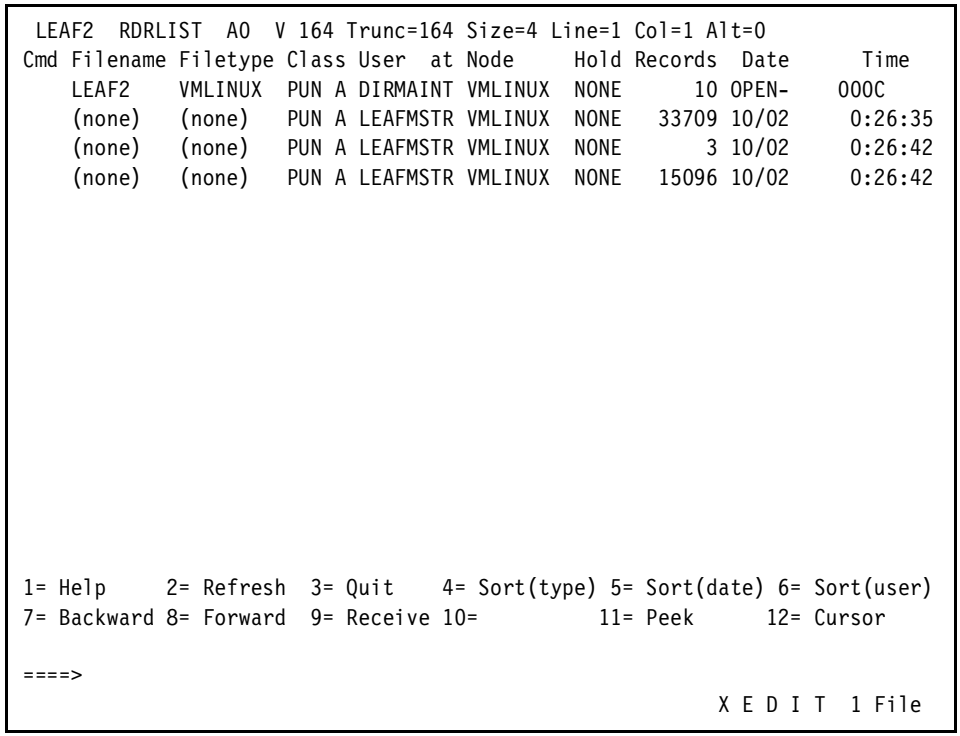

Figure 5 Reader List command on LEAF2

At this point we remembered that we had done some operations on the guest userid using DirMaint prior to the last punch of LEAF2's reader. The DirMaint file was not deleted by the PURGE RDR ALL that we issued from our LEAF IPL exec.

We initially thought that the DirMaint file was not being deleted because it was OPEN, and PURGE does not touch open files. This was not the case, however, as while trying to diagnose the problem we discovered that the file was indeed closed before the IPL and it was the IPL attempt that opened the file.

Further investigation revealed that we did not have sufficient privilege on our LEAFMSTR guest. Since it was defined as a class G user, we could delete our own spooled files but not those of any other user. Adding class D to LEAFMSTR fixed our problem by giving LEAFMSTR sufficient privilege to operate on other users' output on the spool of our LEAF routers.

Curiosity: According to the VM online help, without class D privilege we should not have been able to purge files on another user's spool at all.

Enhancing LEAF on zSeries

There are a few ways in which z/LEAF can be enhanced and developed that we did not have an opportunity to investigate fully.

The “Gold” z/LEAF build

This paper discusses experiences with z/LEAF that have been gained through hacking and experimentation. To produce a real z/LEAF distribution, it would be necessary to follow the build process in the same way as the LEAF developers do when producing a build for i386.

LEAF is based on the Debian GNU/Linux distribution. To build a true LEAF distribution, it would be necessary to start with a Debian installation on zSeries.

Note: You can find out all about Debian on S/390 (including how to install it) by going to the Debian GNU/Linux Web site at:

http://www.debian.org

Debian provides an installation method that can be run from any existing Linux system. This makes it very easy to install a new Debian system. Look for “debootstrap” in the installation documentation.

One thing to watch out for is the process of installing network connectivity. We had to download the IBM OCO drivers from the IBM developerWorks site and install them manually. We used the kernel 2.4.17 “31-bit no-timer” modules from the “August 2001 stream” page.

The process to be followed would be:

1. Obtain the correct kernel source tree, and patch it with the recommended S/390 patches from IBM developerWorks.
2. Patch the kernel with the LEAF-recommended kernel patches, and build a LEAF kernel.
3. Build the LEAF support packages (BusyBox, TinyLogin, etc) from source using the LEAF configuration.
4. Copy the LEAF scripts (**`leafrc`**, **`lrcfg`**, **`lrpkg`**) to the system.
5. Build the LEAF packages, and copy them to package media.
6. Test your LEAF build.

Useful documentation on building packages for LEAF and compiling a LEAF kernel are available from the LEAF project Web site. Note that you are not

restricted to LEAF Bering, as you can port any of the LEAF builds (recall that we chose Bering simply because it was based on the Linux 2.4 kernel).

Tip: The author will be continuing work on the z/LEAF port, so watch for updates to this paper.

BusyBox insmod for s390

The BusyBox package can provide the functionality of insmod, but the code is specific to only two or three platforms (none of which is zSeries). This means that we cannot use this code on zSeries.

To avoid this issue, we had to copy the insmod program from our host distribution to the LEAF package. This increases the total size of the package compared to using the BusyBox code.

In order to truly align z/LEAF with the LEAF builds on other systems, the insmod code in BusyBox would have to be ported to s390 (and s390x for 64-bit).

Sharing system resources

One way to further minimize the overall resource consumption of a number of z/LEAF routers would be to have them share as much system resource as possible. The current IPL process, while it is easy to manage, does create a fair amount of duplication. This duplication occurs for two reasons:

- Each z/LEAF router loads kernel and modules into its own storage
- The kernel, initrd and parameter files sit it the router guests' readers

These issues could be addressed with some development. There is some ongoing work in using facilities of z/VM to share storage between guests, for the purposes of sharing kernel code. This is ideal for the z/LEAF scenario, since all of the routers would run identical kernel code.

Note: The topic of using VM facilities to share kernel code among guests was introduced in *Linux on IBM @server zSeries and S/390: ISP/ASP Solutions*, SG24-6299. It is a field of ongoing research.

To reduce spool usage of the z/LEAF guests, our IPL script could be changed to purge the reader after the IPL. However, this introduces a timing dependency since we do not want to purge the reader until the guest has finished its IPL processing. It would be better to change the leafrc script to purge the reader from within the guest itself at an appropriate time.

Another method to reduce spool usage would be to use a kernel that IPLs over the network (using kernel auto-configuration and a network-accessible root device) instead of from an initrd in the VM reader. LEAF supports this configuration, but some work would be required to verify its applicability to z/LEAF.

Restriction: The catch here is with the current broadcast support available in zSeries Guest LANs and HiperSockets. Since the protocols used by kernel auto-configuration (DHCP, BOOTP, RARP) use MAC-level broadcasts and rely on consistent MAC addresses, they do not currently function on Guest LANs. If your z/LEAF router was to configure itself over a “real” network via an OSA things should work, but in our environment it would not because our z/LEAF guests were all attached to QDIO Guest LANs.

We know of a project to provide true broadcast support to Guest LANs, but we do not know if this work will provide the consistent MAC address that a z/LEAF router would require (for BOOTP and RARP, in particular).

How a multi-call binary works

The BusyBox package is one of the best examples of a multi-call binary. This concept allows a single executable file to perform the function of dozens of different utilities usually packaged as separate files. Multi-call binaries exploit a number of operating system features that make it possible for a user of a system to not even know that the programs they are running are all, in fact, the same file.

Invoking the BusyBox multi-call binary

There are two ways to invoke BusyBox functions. In the first method, you issue the command **`busybox`** followed by the name of the function you want to issue. For example, **`busybox ls`** would perform the directory list function (equivalent to the usual **`ls`** command). This method requires no administration, but users of the program would have to remember that they could not simply perform a function by issuing the name of a command.

The second method is to set up a set of symbolic links to the BusyBox executable, each with the name of a function implemented by BusyBox. When BusyBox is run, it checks the name by which it was invoked, and uses that name as the function to be executed. This method requires some administration, as the symbolic links must be maintained, but system users can follow the normal practice of performing a function by issuing the name of the command.

Tip: Current versions of BusyBox include a parameter that will make BusyBox create appropriate symbolic links for all of the functions that the binary implements. This means that you can start with an empty directory and have BusyBox fill it with the right links for that build of the program.

Figure 6 shows the symbolic links created for **ln** and **ls** commands.

```
# pwd
/bin
# ls -l l*
lrwxrwxrwx 1 root root           12 Oct  2 00:11 ln -> /bin/busybox
lrwxrwxrwx 1 root root           14 Oct  2 00:11 login -> /bin/tinylogin
lrwxrwxrwx 1 root root           12 Oct  2 00:11 ls -> /bin/busybox

# ls -lG
ls: invalid option -- G
BusyBox v0.60.3 (2002.09.26-00:58+0000) multi-call binary

Usage: ls [-1AacCdeFilnpLRrSsTtuvwxXhk] [filenames...]

List directory contents

...
```

Figure 6 BusyBox session

The first **ls** output shows that `/bin/ls` is a symbolic link to `/bin/busybox`. The next part of the output (generated by providing an un-implemented option to the **ls** command) shows that the BusyBox binary is being used to process the **ls** command.

Note: In the first output you can also see `login`, which is a symbolic link to `/bin/tinylogin`. TinyLogin is a partner program to BusyBox and performs the functions of programs like `login` and `sulogin`. These functions could have been implemented in BusyBox, but for security reasons it is preferred to have a separate executable for login processing.

Figure 6 also shows us another feature of the BusyBox utility. In the full GNU implementation of **ls**, the -G option is valid (it suppresses the display of the group name from the directory list). In the interests of saving space, however, not all of the function of the various utilities is provided. This is quite appropriate for

BusyBox, however, since the idea is to eliminate unused (or little used) functions in the interests of reducing the executable size.

Writing a multi-call binary

So, how does a multi-call binary like BusyBox, when invoked using a symbolic link, know what function to perform? The answer is that the way a multi-call binary program is written differs from a normal program.

The C language is used for most systems programming on UNIX/POSIX systems. Programs written in C always have a main() function, which is the first part of the program to be executed. The main function is written in a particular way, to allow the operating system to pass parameters to it. A typical main() function declaration appears here:

```
int main(int argc, *char argv[])
```

The parameters passed to the main() function are argc, an integer containing the number of parameters passed by the system to the program, and argv, the list of the parameters passed. By convention[4], there will always be at least one parameter passed to the program: the name used to invoke the program. This is usually the command typed by the user at the shell prompt to invoke the command, and will just about always be the name of the file that contains the program. In C notation, this value (the first item in the array called argv) is argv[0].

Most single call binaries ignore the contents of argv[0], as the program is designed to perform a single task and it is irrelevant what name the system used to invoke the program.

Note: Some programs, for security reasons, do make sure that the command issued is correct. This can prevent a malicious user from executing a program they should not have access to.

A multi-call binary pays attention to this parameter, however, and uses it to determine which function to execute. In the case of BusyBox, if argv[0] is the same as the executable file name it will use the second item in the parameter list (argv[1]) as the name of the function to be executed. If argv[0] is *not* the same as the name of the BusyBox executable file, it will attempt to use the contents of argv[0] as the name of the requested function.

[4] On UNIX/POSIX systems, at least.

References

IBM Redbooks

- *Linux on IBM @server zSeries and S/390: Large Scale Linux Deployment*, SG24-6824

  http://www.ibm.com/redbooks/abstracts/sg246824.html

Referenced Web sites

- Linux Embedded Appliance Firewall home page

  http://leaf.sourceforge.net/

- Debian GNU/Linux home page

  http://www.debian.org

The team that wrote this paper

This Redpaper was produced by specialists from around the world working at the International Technical Support Organization, Poughkeepsie Center.

Gregory Geiselhart is a project leader for Linux on zSeries at the International Technical Support Organization, Poughkeepsie Center.

Vic Cross is the Linux for zSeries and S/390 Team Leader at Independent Systems Integrators, IBM's Large Systems Business Partner in Australia. He has more than 15 years of experience in general computing, seven of which has been spent working on S/390 and zSeries. He holds a Bachelor of Computing Science degree from Queensland University of Technology. His areas of expertise include networking and Linux. He is a co-author of IBM Redbooks *Linux on IBM @server zSeries and S/390: ISP/ASP Solutions*, SG24-6299, and *Linux on IBM @server zSeries and S/390: Large Scale Linux Deployment*, SG24-6824.

Thanks to the following people for their contributions to this project:

Terry Barthel, Dave Bennin, Alison Chandler, Roy Costa, Al Schwab
International Technical Support Organization, Poughkeepsie Center

Grant Bigham
IBM Global Services Australia

Notices

This information was developed for products and services offered in the U.S.A.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:
IBM Director of Licensing, IBM Corporation, North Castle Drive Armonk, NY 10504-1785 U.S.A.

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law: INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

COPYRIGHT LICENSE:
This information contains sample application programs in source language, which illustrates programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. You may copy, modify, and distribute these sample programs in any form without payment to IBM for the purposes of developing, using, marketing, or distributing application programs conforming to IBM's application programming interfaces.

This document created or updated on December 4, 2002.

Send us your comments in one of the following ways:

- Use the online **Contact us** review redbook form found at:
  ibm.com/redbooks
- Send your comments in an Internet note to:
  redbook@us.ibm.com
- Mail your comments to:
  IBM Corporation, International Technical Support Organization
  Dept. HYJ Mail Station P099
  2455 South Road
  Poughkeepsie, NY 12601-5400 U.S.A.

Trademarks

The following terms are trademarks of the International Business Machines Corporation in the United States, other countries, or both:

developerWorks™	Multiprise®	S/390®
DirMaint™	Redbooks(logo)™ 	SP™
IBM eServer™	Redbooks™	z/VM™
IBM®	RMF™	zSeries™

The following terms are trademarks of International Business Machines Corporation and Lotus Development Corporation in the United States, other countries, or both:

Lotus®	Word Pro®

The following terms are trademarks of other companies:

ActionMedia, LANDesk, MMX, Pentium and ProShare are trademarks of Intel Corporation in the United States, other countries, or both.

Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

Java and all Java-based trademarks and logos are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

C-bus is a trademark of Corollary, Inc. in the United States, other countries, or both.

UNIX is a registered trademark of The Open Group in the United States and other countries.

SET, SET Secure Electronic Transaction, and the SET Logo are trademarks owned by SET Secure Electronic Transaction LLC.

Other company, product, and service names may be trademarks or service marks of others.