Mastering Linux debugging techniques

Key strategies to locate and stomp bugs on Linux

Level: Intermediate

Steve Best (sbest@us.ibm.com), JFS core team member, IBM

01 Aug 2002

There are various ways to watch a running user-space program: you can run a debugger on it and step through the program, add print statements, or add a tool to analyze the program. This article describes methods you can use to debug programs that run on Linux. We review four scenarios for debugging problems, including segmentation faults, memory overruns and leaks, and hangs. (This article appears in the August 2002 issue of the *IBM developerWorks journal*.)

This article presents four scenarios for debugging Linux programs. For Scenario 1, we use two sample programs with memory allocation problems that we debug using the MEMWATCH and Yet Another Malloc Debugger (YAMD) tools. Scenario 2 uses the strace utility in Linux that enables the tracing of system calls and signals to identify where a program is failing. Scenario 3 uses the Oops functionality of the Linux kernel to solve a segmentation fault problem and shows you how to set up the kernel source level debugger (kgdb) to solve the same problem using the GNU debugger (gdb); the kgdb program is the Linux kernel remote gdb via serial connection. Scenario 4 displays information for the component that is causing a hang by using a magic key sequence available on Linux.

General debugging strategies

When your program contains a bug, it is likely that somewhere in the code, a condition that you believe to be true is actually false. Finding your bug is a process of confirming what you believe is true until you find something that is false.

The following are examples of the types of things you may believe to be true:

- At a certain point in the source code, a variable has a certain value.
- At a given point, a structure has been set up correctly.
- At a given `if-then-else` statement, the `if` part is the path that was executed.
- When the subroutine is called, the routine receives its parameters correctly.

Finding the bug involves confirming all of these things. If you believe that a certain variable should have a specific value when a subroutine is called, check it. If you believe that an `if` construct is executed, check it. Usually you will confirm your assumptions, but eventually you will find a case where your belief is wrong. As a result, you will know the location of the bug.

Debugging is something that you cannot avoid. There are many ways to go about debugging, such as printing out messages to the screen, using a debugger, or just thinking about the program execution and making an educated guess about the problem.

Before you can fix a bug, you must locate its source. For example, with segmentation faults, you need to know on which line of code the seg fault occurred. Once you find the line of code in question, determine the value of the variables in that method, how the method was called, and specifically why the error occurred. Using a debugger makes finding all of this information simple. If a debugger is not available, there are other tools to use. (Note that a debugger may not be available in a production environment, and the Linux kernel does not have a debugger built in.)

This article looks at a class of problems that can be difficult to find by visually inspecting code, and these problems may occur only under rare circumstances. Often, a memory error occurs only in a combination of circumstances, and sometimes you can discover memory bugs only after you deploy your program.

Useful memory and kernel debugging tools

There are various ways to track down user-space and kernel problems using debug tools on Linux. Build and debug your source code with these tools and techniques:

User-space tools:

- Memory tools: MEMWATCH and YAMD
- strace
- GNU debugger (gdb)
- Magic key sequence

Kernel tools:

Scenario 1: Memory debugging tools

As the standard programming language on Linux systems, the C language gives you a great deal of control over dynamic memory allocation. This freedom, however, can lead to significant memory management problems, and these problems can cause programs to crash or degrade over time.

Memory leaks (in which `malloc()` memory is never released with corresponding `free()` calls) and buffer overruns (writing past memory that has been allocated for an array, for example) are some of the common problems and can be difficult to detect. This section looks at a few debugging tools that greatly simplify detecting and isolating memory problems.

- Kernel source level debugger (kgdb)
- Built-in kernel debugger (kdb)
- Oops

MEMWATCH

MEMWATCH, written by Johan Lindh, is an open source memory error detection tool for C that you can download (see the Resources later in this article). By simply adding a header file to your code and defining MEMWATCH in your gcc statement, you can track memory leaks and corruptions in your program. MEMWATCH supports ANSI C, provides a log of the results, and detects double-frees, erroneous frees, unfreed memory, overflow and underflow, and so on.

Listing 1. Memory sample (test1.c)

```
#include <stdlib.h>
#include <stdio.h>
#include "memwatch.h"

int main(void)
{
  char *ptr1;
  char *ptr2;

  ptr1 = malloc(512);
  ptr2 = malloc(512);

  ptr2 = ptr1;
  free(ptr2);
  free(ptr1);
}
```

The code in Listing 1 allocates two 512-byte blocks of memory, and then the pointer to the first block is set to the second block. As a result, the address of the second block is lost, and there is a memory leak.

Now compile memwatch.c with Listing 1. The following is an example makefile:

test1

```
gcc -DMEMWATCH -DMW_STDIO test1.c memwatch
c -o test1
```

When you run the test1 program, it produces a report of leaked memory. Listing 2 shows the example memwatch.log output file.

Listing 2. test1 memwatch.log file

```
MEMWATCH 2.67 Copyright (C) 1992-1999 Johan Lindh

...
double-free: <4> test1.c(15), 0x80517b4 was freed from test1.c(14)
...
unfreed: <2> test1.c(11), 512 bytes at 0x80519e4
{FE FE FE FE FE FE FE FE FE FE FE FE ..............}

Memory usage statistics (global):
  N)umber of allocations made:  2
  L)argest memory usage :       1024
  T)otal of all alloc() calls:  1024
  U)nfreed bytes totals :       512
```

MEMWATCH gives you the actual line that has the problem. If you free an already freed pointer, it tells you. The same goes for unfreed memory. The section at the end of the log displays statistics, including how much memory was leaked, how much was used, and the total amount allocated.

YAMD

Written by Nate Eldredge, the YAMD package finds dynamic, memory allocation related problems in C and C++. The latest version of YAMD at the time of writing this article was 0.32. Download yamd-0.32.tar.gz (see Resources). Execute a `make` command to build the program; then execute a `make install` command to install the program and set up the tool.

Once you have downloaded YAMD, use it on test1.c. Remove the `#include memwatch.h` and make a small change to the makefile, as shown below:

test1 with YAMD

```
gcc -g test1.c -o test1
```

Listing 3 shows the output from YAMD on test1.

Listing 3. test1 output with YAMD

```
YAMD version 0.32
Executable: /usr/src/test/yamd-0.32/test1
...
INFO: Normal allocation of this block
Address 0x40025e00, size 512
...
INFO: Normal allocation of this block
Address 0x40028e00, size 512
...
INFO: Normal deallocation of this block
Address 0x40025e00, size 512
...
ERROR: Multiple freeing At
free of pointer already freed
Address 0x40025e00, size 512
...
WARNING: Memory leak
Address 0x40028e00, size 512
WARNING: Total memory leaks:
1 unfreed allocations totaling 512 bytes

*** Finished at Tue ... 10:07:15 2002
Allocated a grand total of 1024 bytes 2 allocations
Average of 512 bytes per allocation
Max bytes allocated at one time: 1024
24 K alloced internally / 12 K mapped now / 8 K max
Virtual program size is 1416 K
End.
```

YAMD shows that we have already freed the memory, and there is a memory leak. Let's try YAMD on another sample program in Listing 4.

Listing 4. Memory code (test2.c)

```
#include <stdlib.h>
#include <stdio.h>

int main(void)
{
  char *ptr1;
  char *ptr2;
  char *chptr;
  int i = 1;
  ptr1 = malloc(512);
  ptr2 = malloc(512);
  chptr = (char *)malloc(512);
  for (i; i <= 512; i++) {
    chptr[i] = 'S';
  }
  ptr2 = ptr1;
  free(ptr2);
  free(ptr1);
  free(chptr);
}
```

You can use the following command to start YAMD:

```
./run-yamd /usr/src/test/test2/test2
```

Listing 5 shows the output from using YAMD on the sample program test2. YAMD tells us that we have an out-of-bounds condition in the `for` loop.

Listing 5. test2 output with YAMD

```
Running /usr/src/test/test2/test2
Temp output to /tmp/yamd-out.1243
*********
./run-yamd: line 101: 1248 Segmentation fault (core dumped)
YAMD version 0.32
Starting run: /usr/src/test/test2/test2
Executable: /usr/src/test/test2/test2
Virtual program size is 1380 K
...
INFO: Normal allocation of this block
Address 0x40025e00, size 512
...
INFO: Normal allocation of this block
Address 0x40028e00, size 512
...
INFO: Normal allocation of this block
Address 0x4002be00, size 512
ERROR: Crash
...
Tried to write address 0x4002c000
Seems to be part of this block:
Address 0x4002be00, size 512
...
Address in question is at offset 512 (out of bounds)
Will dump core after checking heap.
Done.
```

MEMWATCH and YAMD are both useful debugging tools and they require different approaches. With MEMWATCH, you need to add the include file memwatch.h and turn on two compile time flags. YAMD only requires the `-g` option for the link statement.

Electric Fence

Most Linux distributions include a package for Electric Fence, but you can download it as well. Electric Fence is a `malloc()` debugging library written by Bruce Perens. It allocates protected memory just after the memory you allocate. If there is a fencepost error (running off the end of an array), your program will immediately exit with a protection error. By combining Electric Fence with gdb, you can track down exactly what line tried to access the protected memory. Electric Fence can detect memory leaks as another feature.

Scenario 2: Using strace

The `strace` command is a powerful tool that shows all of the system calls issued by a user-space program. Strace displays the arguments to the calls and returns values in symbolic form. Strace receives information from the kernel and does not require the kernel to be built in any special way. The trace information can be useful to send to both application and kernel developers. In Listing 6, a format of a partition is failing, and the listing shows the start of the strace on call out make file system (`mkfs`). Strace determines which call is causing the problem.

Listing 6. Start of the strace on mkfs

```
execve("/sbin/mkfs.jfs", ["mkfs.jfs", "-f", "/dev/test1"], &
 ...
 open("/dev/test1", O_RDWR|O_LARGEFILE) = 4
 stat64("/dev/test1", {st_mode=&, st_rdev=makedev(63, 255), ...}) = 0
 ioctl(4, 0x40041271, 0xbfffe128) = -1 EINVAL (Invalid argument)
```

```
write(2, "mkfs.jfs: warning - cannot setb" ..., 98mkfs.jfs: warning -
cannot set blocksize on block device /dev/test1: Invalid argument )
 = 98
stat64("/dev/test1", {st_mode=&, st_rdev=makedev(63, 255), ...}) = 0
open("/dev/test1", O_RDONLY|O_LARGEFILE) = 5
ioctl(5, 0x80041272, 0xbfffe124) = -1 EINVAL (Invalid argument)
write(2, "mkfs.jfs: can\'t determine device"..., ..._exit(1)
 = ?
```

Listing 6 shows that the `ioctl` call caused the `mkfs` program that was used to format a partition to fail. The `ioctl BLKGETSIZE64` is failing. (`BLKGET-SIZE64` is defined in the source code that calls `ioctl`.) The `BLKGETSIZE64 ioctl` is being added to all the devices in Linux, and in this case, the logical volume manager does not support it yet. Therefore, the mkfs code will change to call the older `ioctl` call if the `BLKGETSIZE64 ioctl` call fails; this allows `mkfs` to work with the logical volume manager.

Scenario 3: Using gdb and Oops

You can use the gdb program, the debugger from the Free Software Foundation, from the command line or from one of several graphical tools, such as Data Display Debugger (DDD) to ferret out your errors. You can use gdb to debug user-space programs or the Linux kernel. This section covers only the command line for gdb.

Start gdb by using the `gdb program name` command. The gdb will load the executable's symbols and then display an input prompt to allow you to start using the debugger. There are three ways to view a process with gdb:

- Use the attach command to start viewing an already running process; attach will stop the process.
- Use the run command to execute the program and to start debugging the program at the beginning.
- Look at an existing core file to determine the state the process was in when it terminated. To view a core file, start gdb with the following command:

  ```
  gdb programname corefilename
  ```

  To debug with a core file, you need the program executable and source files, as well as the core file. To start gdb with a core file, use the -c option:

  ```
  gdb -c core programname
  ```

  The gdb shows what line of code caused the program to core dump.

Before you run a program or attach to an already running program, list the source code where you believe the bug is, set breakpoints, and then start debugging the program. You can view extensive gdb online help and a detailed tutorial by using the `help` command.

---

kgdb

The kgdb program (remote host Linux kernel debugger through gdb) provides a mechanism to debug the Linux kernel using gdb. The kgdb program is an extension of the kernel that allows you to connect to a machine running the kgdb-extended kernel when you are running gdb on a remote host machine. You can then break into the kernel, set break points, examine data, and so on (similar to how you would use gdb on an application program). One of the primary features of this patch is that the remote host running gdb connects to the target machine (running the kernel to be debugged) during the boot process. This allows you to begin debugging as early as possible. Note that the patch adds functionality to the Linux kernel so gdb can be used to debug the Linux kernel.

Two machines are required to use kgdb: one of these is a development machine, and the other is a test machine. A serial line (null-modem cable) connects the machines through their serial ports. The kernel you want to debug runs on the test machine; gdb runs on the development machine. The gdb uses the serial line to communicate to the kernel you are debugging.

Use the following steps to set up a kgdb debug environment:

1. Download the appropriate patch for your version of the Linux kernel.

2. Build your component into the kernel, as this is the easiest way to use kgdb. (Note that there are two ways to build most components of the kernel, such as a module or directly into the kernel. For example, Journaled File System (JFS) can be built as a module or directly into the kernel. Using the gdb patch, we can build JFS directly into the kernel.)

3. Apply the kernel patch and rebuild the kernel.

4. Create a file called .gdbinit and place it in your kernel source subdirectory (in other words, /usr/src/linux). The file .gdbinit has the following four lines in it:
    - `set remotebaud 115200`
    - `symbol-file vmlinux`
    - `target remote /dev/ttyS0`
    - `set output-radix 16`

5. Add the append=gdb line to lilo, which is the boot load used to select which kernel is used to boot the kernel.
    - `image=/boot/bzImage-2.4.17`
    - `label=gdb2417`
    - `read-only`
    - `root=/dev/sda8`
    - `append="gdb gdbttyS=1 gdb-baud=115200 nmi_watchdog=0"`

Listing 7 is an example of a script that pulls the kernel and modules that you built on your development machine over to the test machine. You need to change the following items:

- `best@sfb`: Userid and machine name.
- `/usr/src/linux-2.4.17`: Directory of your kernel source tree.
- `bzImage-2.4.17`: Name of the kernel that will be booted on the test machine.
- `rcp` and `rsync`: Must be allowed to run on the machine the kernel was built on.

Listing 7. Script to pull kernel and modules to test machine

```
set -x
rcp best@sfb: /usr/src/linux-2.4.17/arch/i386/boot/bzImage /boot/bzImage-2.4.17
rcp best@sfb:/usr/src/linux-2.4.17/System.map /boot/System.map-2.4.17
rm -rf /lib/modules/2.4.17
rsync -a best@sfb:/lib/modules/2.4.17 /lib/modules
chown -R root /lib/modules/2.4.17
lilo
```

Now we are ready to start the gdb program on your development machine by changing to the directory where your kernel source tree starts. In this example, our kernel source tree is at /usr/src/linux-2.4.17. Type `gdb` to start the program.

If everything is working, the test machine will stop during boot process. Enter the `gdb` command `cont` to continue the boot process. One common problem is that the null-modem cable could be connected to the wrong serial port. If gdb does not start, switch the port to the second serial, and this should enable gdb to start.

Using kgdb to debug kernel problems

Listing 8 lists the code that was modified in the source of the jfs_mount.c file to create a segmentation fault at line 109 by creating a null pointer exception.

Listing 8. Modified jfs_mount.c code

```
int jfs_mount(struct super_block *sb)
{
...
int ptr;                        /* line 1 added */
jFYI(1, ("\nMount JFS\n"));
/ *
* read/validate superblock
* (initialize mount inode from the superblock)
* /
if ((rc = chkSuper(sb))) {
```

```
                goto errout20;
        }
108     ptr=0;                  /* line 2 added */
109     printk("%d\n",*ptr);    /* line 3 added */
```

Listing 9 displays a gdb exception after the mount command to the file system is issued. There are several commands that are available from kgdb, such as displaying data structures and values of variables and seeing what state all tasks in the system are in, where they are sleeping, where they are spending CPU, and so on. Listing 9 shows the information that the back trace provides for this problem; the `where` command is used to do a back trace, which tells the calls that were executed to get to the stopping place in your code.

Listing 9. A gdb exception and back trace

```
mount -t jfs /dev/sdb /jfs

Program received signal SIGSEGV, Segmentation fault.
jfs_mount (sb=0xf78a3800) at jfs_mount.c:109
109             printk("%d\n",*ptr);
(gdb)where
#0 jfs_mount (sb=0xf78a3800) at jfs_mount.c:109
#1 0xc01a0dbb in jfs_read_super ... at super.c:280
#2 0xc0149ff5 in get_sb_bdev ... at super.c:620
#3 0xc014a89f in do_kern_mount ... at super.c:849
#4 0xc0160e66 in do_add_mount ... at namespace.c:569
#5 0xc01610f4 in do_mount ... at namespace.c:683
#6 0xc01611ea in sys_mount ... at namespace.c:716
#7 0xc01074a7 in system_call () at af_packet.c:1891
#8 0x0 in ?? ()
(gdb)
```

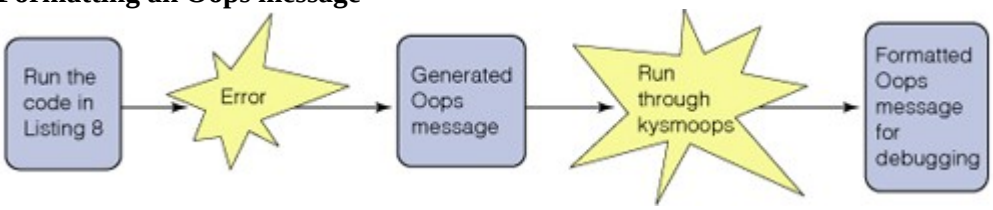
The next section looks at this same JFS segmentation fault problem without a debugger setup and uses the Oops that the kernel would have produced if you executed the code in Listing 8 in a non-kgdb kernel environment.

Oops analysis

The Oops (or panic) message contains details of a system failure, such as the contents of CPU registers. With Linux, the traditional method for debugging a system crash has been to analyze the details of the Oops message sent to the system console at the time of the crash. Once you capture the details, the message then can be passed to the ksymoops utility, which attempts to convert the code to instructions and map stack values to kernel symbols. In many cases, this is enough information for you to determine a possible cause of the failure. Note that the Oops message does not include a core file.

Let's say your system has just created an Oops message. As the author of the code, you want to solve the problem and determine what has caused the Oops message, or you want to give the developer of the code that has displayed the Oops message the most information about your problem, so it can be solved in a timely manner. The Oops message is one part of the equation, but it is not helpful without running it through the ksymoops program. The figure below shows the process of formatting an Oops message.

Formatting an Oops message



There are several items that ksymoops needs: Oops message output, the System.map file from the kernel that is running, and /proc/ksyms, vmlinux, and /proc/modules. There are complete instructions on how to use ksymoops in the kernel source /usr/src/linux/Documentation/oops-tracing. txt or on the ksymoops man page. Ksymoops disassembles the code section, points to the failing instruction, and displays a trace section that shows how the code was called.

First, get the Oops message into a file so you can run it through the ksymoops utility. Listing 10 shows the Oops created by the mount to the JFS file system with the problem that was created by the three lines in Listing 8 that were added to the mount code of JFS.

Listing 10. Oops after being processed by ksymoops

```
ksymoops 2.4.0 on i686 2.4.17. Options used
... 15:59:37 sfb1 kernel: Unable to handle kernel NULL pointer dereference at
virtual address 0000000
... 15:59:37 sfb1 kernel: c01588fc
... 15:59:37 sfb1 kernel: *pde = 0000000
... 15:59:37 sfb1 kernel: Oops: 0000
... 15:59:37 sfb1 kernel: CPU:    0
... 15:59:37 sfb1 kernel: EIP:    0010:[jfs_mount+60/704]

... 15:59:37 sfb1 kernel: Call Trace: [jfs_read_super+287/688]
[get_sb_bdev+563/736] [do_kern_mount+189/336] [do_add_mount+35/208]
[do_page_fault+0/1264]
... 15:59:37 sfb1 kernel: Call Trace: [<c0155d4f>]...
... 15:59:37 sfb1 kernel: [<c0106e04 ...
... 15:59:37 sfb1 kernel: Code: 8b 2d 00 00 00 00 55 ...

>>EIP; c01588fc <jfs_mount+3c/2c0> <=====
...
Trace; c0106cf3 <system_call+33/40>
Code; c01588fc <jfs_mount+3c/2c0>
00000000 <_EIP>:
Code; c01588fc <jfs_mount+3c/2c0>  <=====
   0: 8b 2d 00 00 00 00         mov     0x0,%ebp    <=====
Code; c0158902 <jfs_mount+42/2c0>
   6:  55                       push    %ebp
```

Next, you need to determine which line is causing the problem in jfs_mount. The Oops message tells us that the problem is caused by the instruction at offset 3c. One way you can do this is to use the objdump utility on the jfs_mount.o file and look at offset 3c. Objdump is used to disassemble a module function and see what assembler instructions are created by your C source code. Listing 11 shows what you would see from objdump, and next, we can look at C code for jfs_mount and see that the null was caused by line 109. Offset 3c is important because that is the location that the Oops message identified as the cause of the problem.

Listing 11. Assembler listing of jfs_mount

```
109     printk("%d\n",*ptr);

objdump jfs_mount.o

jfs_mount.o:    file format elf32-i386

Disassembly of section .text:

00000000 <jfs_mount>:
   0:55                          push %ebp
  ...
  2c:   e8 cf 03 00 00     call    400 <chkSuper>
  31:   89 c3                   mov     %eax,%ebx
  33:   58                      pop     %eax
  34:   85 db                   test    %ebx,%ebx
  36:   0f 85 55 02 00 00 jne   291 <jfs_mount+0x291>
  3c:   8b 2d 00 00 00 00 mov   0x0,%ebp << problem line above
  42:   55                      push    %ebp
```

---

kdb

The Linux kernel debugger (kdb) is a patch for the Linux kernel and provides a means of examining kernel memory and data structures while the system is operational. Note that kdb does not require two machines, but it does not allow you to do source level debugging like kgdb. You can add additional commands to format and display essential system data structures given an identifier or address of the data structure. The current command set allows you to control kernel operations, including the following:

- Single-stepping a processor
- Stopping upon execution of a specific instruction
- Stopping upon access (or modification) of a specific virtual memory location
- Stopping upon access to a register in the input-output address space
- Stack back trace for the current active task as well as for all other tasks (by process ID)
- Instruction disassembly

Scenario 4: Getting back trace using magic key sequence

If your keyboard is still functioning and you have a hang on Linux, use the following method to help resolve the source of the hang problem. With these steps, you can display a back trace of the current running process and all processes using the magic key sequence.

1. The kernel that you are running must be built with `CONFIG_MAGIC_SYS-REQ` enabled. You must also be in text-mode. CLTR+ALT+F1 will get you into text mode, and CLTR+ALT+F7 will get you back to X Windows.
2. While in text-mode, press <ALT+ScrollLock> followed by <Ctrl+ScrollLock>. The magic keystrokes will give a stack trace of the currently running processes and all processes, respectively.
3. Look in your /var/log/messages. If you have everything set up correctly, the system should have converted the symbolic kernel addresses for you. The back trace will be written to the /var/log/messages file.

Chasing memory overruns

You do not want to be in a situation like an allocation overrun that happens after thousands of calls.

Our team spent many long hours tracking down an odd memory corruption problem. The application worked on our development workstation, but it would fail after two million calls to `malloc()` on the new product workstation. The real problem was an overrun back around call one million. The new system had the problem because the reserved `malloc()` area was laid out differently, so the offending memory was located at a different place and destroyed something different when it did the overrun.

We solved this problem using many different techniques, one using a debugger, another adding tracing to the source code. I started to look at memory debugging tools about this same time in my career, looking to solve these types of problems faster and more efficiently. One of the first things I do when starting a new project is to run both MEMWATCH and YAMD to see if they can point out memory management problems.

Memory leaks are a common problem in applications, but you can use the tools described in this article to solve them.

Conclusion

There are many different tools available to help debug programs on Linux. The tools in this article can help you solve many coding problems. Tools that show the location of memory leaks, overruns, and the like can solve memory management problems, and I find MEMWATCH and YAMD helpful.

Using a Linux kernel patch to allow gdb to work on the Linux kernel helped in solving problems on the filesystem that I work on in Linux. In addition, the strace utility helped determine where a filesystem utility had a failure during a system call. Next time you are faced with squashing a bug in Linux, try one of these tools.

Resources

- Download MEMWATCH.
- Read the article "Linux software debugging with GDB". (*developerWorks*, February 2001)
- Visit the IBM Linux Technology Center.
- Find more Linux articles in the *developerWorks* Linux zone.

About the author

Steve Best works in the Linux Technology Center of IBM in Austin, Texas. He currently is working on the Journaled File System (JFS) for Linux project. Steve has done extensive work in operating system development with a focus in the areas of file systems, internationalization, and security.