Linux Debugging Techniques

CS378 - Linux Kernel Programming, UT
4/14/2003

Steve Best
IBM Linux Technology Center
JFS for Linux
http://oss.software.ibm.com/jfs
IBM Austin, Texas
sbest@us.ibm.com
Overview of Talk

- Types of Problems
- Tools
- Error and Debug Messages
- Handling Failures
- Kernel Investigation
- Handling a System Crash
- Oops Analysis Example
- LKCD/Lcrash
Types of Problems

- Application/User space vs. Kernel
  - Difference in difficulty
  - Problem's source
- Development vs. Production
  - Difference in tool availability
  - More difficult to reproduce
Tools

- Library and system call trace
  - strace, ltrace

- Debuggers
  - gdb, ddd
  - kgdb
  - kdb

- Built-In
  - Oops data upon a panic/crash

- Dump Facility
  - Linux Kernel Crash Dump - lkcd
Tools (continued)

- Linux Trace Toolkit
  - ltt
- Custom Kernel Instrumentation
  - dprobes
- Special console functions
  - Magic SysReq key
Error/ Debug Messages

- System error logs
  - var/log/*
  - dmesg
- SysLog
- Console
- Application or Module debug level
Handling Failures

- System Crash
  - Collect and analyze oops/panic data
  - Collect and analyze dump with lkcd

- System Hang
  - Use Magic SysReq Keys
  - NMI invoking a dump using lkcd
  - Look at the hang using debugger
    - kdb or kgdb
  - ps command
Kernel Investigation

- User mode Linux - run Linux under Linux
- Debuggers
  - Gdb and /proc/kcore
  - Remote kernel debugging
    - kgdb & serial connection
    - Kdb with or without serial connection
- Lcrash on running system
- Adding printk's in the kernel
Handling a System Crash

- Occurs when a critical system failure is detected
- Kernel routine call oops
  - Attempts to report/record system state
  - Information is limited (after the fact)
- Better to have an entire system memory dump
  - LKCD project on Sourceforge
  - Thorough analysis and investigation can be done
Panic/Oops Analysis

error generates Oops msg
Run through kysmoops Formats Oops msg source code that has problem
Panic/Oops Analysis

- **Steps**
  - Collect oops output, System.map, /proc/ksyms, vmlinux, /proc/modules
  - Use ksymoops to interpret oops
    - Instructions is /usr/src/linux/Documentation/oops-tracing.txt
    - Ksymoops(8) man page

- **Brief analysis**
  - Ksymoops disassembles the section of code
  - EIP points to the failing instruction
  - The call trace section shows how the code got there

- How to find failing line of code?
Example that causes Oops

Change code inside the mount code for JFS

- add null pointer exception to the mount code
- mount -t jfs /dev/hdb1 /jfs
- Oops is create since the mount code isn't functioning correctly
Oops Example

Unable to handle kernel NULL pointer dereference at virtual address 00000000

\texttt{c01588fc} <---- EIP (Instruction Pointer)

*\texttt{pde} = 00000000

Oops: 0000

CPU: 0

EIP: 0010:[jfs\_mount+60/704]

EFLAGS: 00010246

eax: cd83a000   ebx: 00000000   ecx: 00000001   edx: 00000003

esi: cd83a000   edi: cdf22d20   ebp: 00000000   esp: cdb71e74

ds: 0018   es: 0018   ss: 0018

Process mount (pid: 113, stackpage=cdb71000)

Stack: 00000000 000000f0 cd83a000 cdf22d20 cd859c00 00000000 c0155d4f cd83a000
cd83a044 cd83a000 cd83a07c cd839000 c0130b03 cd83a000 cd839000 00000000
00030246 fffffffea cdf371e0 cd9d2c20 cdffa320 0000000a cd860000 cd86000a

**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]
Oops Example (Cont.)

>>EIP; c01588fc <jfs_mount+3c/2c0>  =======
Trace; c0155d4f <jfs_read_super+11f/2b0>
Trace; c0130b03 <get_sb_bdev+233/2e0>
Trace; c013105d <do_kern_mount+bd/150>
Trace; c013ff73 <do_add_mount+23/d0>
Trace; c010f050 <do_page_fault+0/4f0>
Trace; c0106e04 <error_code+34/40>
Trace; c01401fc <do_mount+13c/160>
Trace; c014006b <copy_mount_options+4b/a0>
Trace; c014029c <sys_mount+7c/c0>
Trace; c0106cf3 <system_call+33/40>
Code;  c01588fc <jfs_mount+3c/2c0>
Oops Example (Cont.)

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

Code; c0158903 <jfs_mount+43/2c0>
7: 68 4c 9f 20 c0           push  $0xc0209f4c

Code; c0158908 <jfs_mount+48/2c0>

c: e8 f3 9b fb ff          call  fff9c04 <_EIP+0xffff9c04> c0112500 <printk+0/110>

Code; c015890d <jfs_mount+4d/2c0>
11: 6a 01                    push  $0x1

Code; c015890f <jfs_mount+4f/2c0>
13: 56                       push  %esi
Find failing line of Code

- EIP of `c01588fc` is within the routine `jfs_mount`
- Next, disassemble the routine `jfs_mount` using `objdump`
- `objdump -d jfs_mount.o`

```
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
  42: 55  push  %ebp
```
C Source Code

```c
int jfs_mount(struct super_block *sb)
{
    ...
    int *ptr; /*Added line 1 */
    jFYI(1, ("\nMount JFS\n");

    if ((rc = chkSuper(sb))) {
        goto errout20;
    }
    108  ptr=0; /* Added Line 2*/
    109  printk("%d\n",*ptr); /* Added Line 3*/
}```
- **kgdb**
  - Remote host Linux kernel debugger through gdb provides a mechanism to debug the Linux kernel using gdb
  - Gives you source level type of debugging

- **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
  - Doesn't give you source level type of debugging
Debuggers

- GNU Debugger (gdb)
  - Free Software Foundation's debugger
  - Command line
  - Several graphical tools
    - Data Display Debugger (DDD)

- Ways to view process with this debugger
  - Attach to view already running process
  - Run command to start program
  - Look at an existing core file

- Debugging with GDB Tutorial
Some useful gdb commands

- `attach, at` Attach to an already running process.
- `backtrace, bt` Print a stack trace.
- `break, b` Set a breakpoint.
- `clear` Clear a breakpoint.
- `delete` Clear a breakpoint by number.
- `detach` Detach from the currently attached process.
- `display` Display the value of an expression after execution stops.
- `help` Display help for gdb commands.
- `jump` Jump to an address and continue the execution there.
- `list, l` Lists the 10 lines.
- `next, n` Step to the next machine language instruction.
- `print, p` Print the value of an expression.
- `run, r` Run the current program from the start.
- `set` Change the value of a variable.
kgdb Setup for build machine (laptop)

  - 2.4.18 linux-2.4.18-kgdb-1.5.patch

- Apply the kernel patch and rebuild the kernel

- If possible build the component into the kernel which you need to debug on
  - There is a way to debug modules (info on web page)

- Create a file called .gdbinit and place it in your kernel source sub directory (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
- kgdb setup requires two machines
Setup for Test machine

- Add the append=gdb line to lilo
  image=/boot/bzImage-2.4.17
  label=gdb2417
  read-only
  root=/dev/sda8
  append="gdb gdbttyS=1 gdbbaud=115200 nmi_watchdog=0"

- Pull the kernel and modules that you built on your build machine over to the test machine.
Setup for Test machine (Cont.)

- script to bring kernel and modules over 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
```

best@sfb. Userid and machine name.
bzImage-2.4.17. Name of the kernel that will be booted on the test machine.
- Connect the two machines using null-modem cable
- Build machine
  - start gdb in kernel source tree (i.e. /usr/src/linux-2.4.17)
  - breakpoint () at gdbstub.c:1159
  - (gdb) cont
- Test machine
  - Waiting for connection from remote gdb...
- Common problem (null-modem) cable
  - not connected to the correct serial port
- Build machine
- CTRL+C will get you back into the debugger
- Useful gdb commands
  - where
  - info threads
  - thread xx
null pointer exception using mount problem

mount -t jfs /dev/sdb /jfs  (test machine)

modified source (jfs_mount.c)

```c
int *ptr;            /* line 1 added */

jFYI(1, ("\nMount JFS\n"));

if (!(rc = chkSuper(sb))) {
    goto errout20;
}

108  ptr=0;       /* line 2 added */
109  printk("%d\n",*ptr);  /* line 3 added */
```
gdb info displayed

Program received signal SIGSEGV, Segmentation fault.
jfs_mount (sb=0xf78a3800) at jfs_mount.c:109

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

gdb points you to exact line of failure
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.

- Doesn't provide source level debugging
- Get the patch from SGI web page
  - Look from the kernel that you are interested in (2.4.19)
  - File xfs-2.4.19-rc3-split-kdb-i386.bz2
  - Patch kernel
  - Configure kdb (under kernel hacking)

Note:
- United Linux 1.0 has this debugger built-in already
- kdb setup requires only one machine
- Use kdb need to be text mode
  - CLTR+ALT+F1 (into)
  - CLTR+ALT+F7 (back)
- Everything setup correctly
  - press pause key
  - Entering kdb (current=0xc034a000, pid 0) due to keyboard Entry
  - Serial console with CTRL-A
- Ready to look at processes
  - ps will show you running processes
  - btp <pid> will show you the back strace for <pid>
  - bta will show you you all back straces
KDB can be invoked

- Early init by adding "kdb=early" lilo flag
- Serial console with CTRL-A
- Console with Pause key
- When a pre-set breakpoint is triggered
- On panic
  - `ps` will show you running processes
  - `btp <pid>` will show you the back strace for `<pid>`
  - `bta` will show you all back straces
<table>
<thead>
<tr>
<th>Task Addr</th>
<th>Pid</th>
<th>Parent [*]</th>
<th>cpu</th>
<th>State</th>
<th>Thread</th>
<th>Command</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xc127c000</td>
<td>00000001</td>
<td>00000000</td>
<td>1</td>
<td>000</td>
<td>stop</td>
<td>0xc127c270 init</td>
</tr>
<tr>
<td>0xc1270000</td>
<td>00000002</td>
<td>00000001</td>
<td>1</td>
<td>000</td>
<td>stop</td>
<td>0xc1270270 keventd</td>
</tr>
<tr>
<td>0xcdf3c000</td>
<td>00000003</td>
<td>00000001</td>
<td>1</td>
<td>000</td>
<td>stop</td>
<td>0xcdf3c270 ksoftirqd</td>
</tr>
<tr>
<td>0xcdf3a000</td>
<td>00000004</td>
<td>00000001</td>
<td>1</td>
<td>000</td>
<td>stop</td>
<td>0xcdf3a270 kswapd</td>
</tr>
<tr>
<td>0xcdf38000</td>
<td>00000005</td>
<td>00000001</td>
<td>1</td>
<td>000</td>
<td>stop</td>
<td>0xcdf38270 bdflush</td>
</tr>
<tr>
<td>0xcdf36000</td>
<td>00000006</td>
<td>00000001</td>
<td>1</td>
<td>000</td>
<td>stop</td>
<td>0xcdf36270 kupdated</td>
</tr>
<tr>
<td>0xc12c0000</td>
<td>00000007</td>
<td>00000001</td>
<td>1</td>
<td>000</td>
<td>stop</td>
<td>0xc12c0270 jfsIO</td>
</tr>
<tr>
<td>0xc12be000</td>
<td>00000008</td>
<td>00000001</td>
<td>1</td>
<td>000</td>
<td>stop</td>
<td>0xc12be270 jfsCommit</td>
</tr>
<tr>
<td>0xc12bc000</td>
<td>00000009</td>
<td>00000001</td>
<td>1</td>
<td>000</td>
<td>stop</td>
<td>0xc12bc270 jfsSync</td>
</tr>
<tr>
<td>0xcdeeaa000</td>
<td>00000011</td>
<td>00000001</td>
<td>1</td>
<td>000</td>
<td>stop</td>
<td>0xcdeeaa270 evms_async</td>
</tr>
</tbody>
</table>
Add three lines of code into jfs_mount

mount -t jfs /dev/evms/hdb1 /jfs

end up in the kdb with oops like screen
c0180c8c
*pde = 00000000
Oops: 0000
CPU:   0
EIP: 0010:[<c0180c8c>]    Not tainted
EFLAGS: 00001024
eax: c7865a00 ebx: 00000000 ecx: 00000020 edx: cdf33be0
esi: c7865a00 edi: c8fb8f20 ebp:00000000 esp: ca511e30
ds: 0018 es: 0018 ss: 0018
Process mount (pid: 917, stackpage= ca511000)
Stack: 00000000 000000f0 c7865a00 c8fb8f20 c81d49a0
call Trace:

code:
entering kdb (current=0xca510000, pid 917) Oops: Oops
due to oops @ 0xc0180c8c
kdb>bt
EBP   EIP   Function(args)
0xc0180c8c **jfs_mount+0x3c** (0xc785a00, 0x0, 0x0, 0x0, 0x0)
0xc017dd59 **jfs_super_super+0x149** (0x7865a00, 0x0, 0x0, 0x0, 0x336e5,...)
0xc0135ee9 **get_sb_bdev=0x219** (0xc0316204, 0x0, 0xca42b000,...)
0xc01360dc **do_kern_mount+x5c** (0xca67000, 0x0, 0xca42b000, 0x0)
0xc0145f89 **do_add_mount+0x79** (0x0, 0xca67000, 0xca511f64,...)

Interrupt registers:
eax = 0x00000000 ebx = 0x00000000 ecx= 0xca67000 edx= 0xca511f64
esi = 0xc014624b edi= 0xca511f64 esp= 0x00000000 eip=0x00000000

Interrupt from user space, end of kernel space
kdb> go
Segmentation fault
Use kdb to solve a hang

Example taking snapshot using EVMS
- calls file system to lock and then unlock
  - jfs_write_super_lockfs
  - jfs_unlock_fs

Dead Lock is in the jfs_unlock_fs routine by not calling txResume(sb);
static void jfs_write_super_lockfs(struct super_block *sb) {
    struct jfs_sb_info *sbi = JFS_SBI(sb);
    struct jfs_log *log = sbi->log;

    if (!(sb->s_flags & MS_RDONLY)) {
        txQuiesce(sb);
        lmLogShutdown(log);
    }
}

static void jfs_unlockfs(struct super_block *sb) {
    struct jfs_sb_info *sbi = JFS_SBI(sb);
    struct jfs_log *log = sbi->log;
    int rc = 0;

    if (!(sb->s_flags & MS_RDONLY)) {
        if ((rc = lmLogInit(log))) /* the bug was this if was wrong */
            jERROR(1,                   
                   ("jfs_unlock failed with return code %d\n", rc));
        else
            txResume(sb);
    }
}
Use kdb to solve a hang

Example taking snapshot using EVMS

- Problem after snapshot the volume can't be used
- CTRL+ALT+F1 (get you text mode)
- pause key to get you into (kdb)
- ps to find the copy (cp)
  
  0xc88e8000 00001276 00001256 1 000 stop 0xc88e8270 cp

- back trace on pid 1276
- btp 1276
- **back trace**

  - 0xc019a665 txBegin+0xa5 (0xc2f9fe00, 0x0)
  - 0xc017e126 jfs_truncate_nolock+0xc6 (0xc4e54a00,0x0, 0x0)
  - 0xc017e1d1 jfs_truncate+0x41 (0xc4e54a00)
  - 0xc012260b vmtruncate+0xfb (0xc44e54a00, 0x0, 0x0,0xc88e9ee8, 0x0)
  - 0xc0144346 inode_setattr+0x26( 0xc4e54a00, 0xc88e9ee8, 0x3a95a0f0,0x7, 0xc88e9f10)

- The back trace shows that txBegin is where start looking

- Use `objdump -d jfs_txngmgr.o` to see where in txBegin
display instructions

- id txBegin
  - push %ebp
  - push %esi
  - push %ebx

- id txBegin+a5
  - mov %esi,%eax
  - mov $0x0,(%ebx)

Set breakpoint

- bp txBegin
Do operation that will cause breakpoint to hit

- Entering kdb (current=0xc9714000,pid 902) due to Breakpoint @ 0xc019a5c0
- btp 902 (back trace)
- bc 0 (clear breakpoint)
- go

Options supported by kdb

- kdb help
- man pages /usr/src/linux/Documentation/kdb
  - man .kdb.mm
  - presentation called "slides"
Use `objdump -d jfs_txnmgr.o` to see where in `txBegin`

- `0xc019a665 txBegin+0xa5 (0xc2f9fe00, 0x0)`
- `000003e0 <txBegin>`
  - `push %ebp`
  - `push %edi`
  - `push %esi`
  - `push %ebx`
  - `sub %0x20,%esp`
- `+a5` to start of `txBegin` and that is where the back trace tells us the code is
Kernel must be built with CONFIG_MAGIC_SYSREQ

- Kernel hacking section
- turn on Magic SysRq key
- rebuild the kernel

Need to be in text mode (CTRL+ALT+F1)

- Once in text mode issue the following keys
  - <ALT+ScrollLock>
  - <CTRL+ScrollLock>

- Magic keystrokes will give stack trace of
  - running processes
  - all processes
Kernel must be built with CONFIG_MAGIC_SYSREQ

- Look in /var/log/messages
- If everything setup correctly
  - System will converted symbolic kernel addresses
- Back trace will be written to /var/log/messages
Back trace without symbols

Aug 2 16:40:07 snow kernel: gpm S 00000000 0 1032 1
1055 1013 (NOTLB)

Aug 2 16:40:07 snow kernel: Call Trace: [<c01233cc>]
Aug 2 16:40:07 snow kernel: [<c0123340>]
Aug 2 16:40:07 snow kernel: [<c014c90f>]
Aug 2 16:40:07 snow kernel: [<c014ccb9>]
Aug 2 16:40:07 snow kernel: [<c013d017>]
Aug 2 16:40:07 snow kernel: [<c0108c3b>]
Aug 2 16:40:07 snow kernel:
Back trace with symbols

Sep 4 08:57:27 sfb1 kernel: kswapd        S CA4FC820  5720  4  1 5 3 (L-TLB)

Sep 4 08:57:27 sfb1 kernel: Call Trace:    [kswapd+136/192]
[kswapd+0/192] [stext+0/48] [kernel_thread+38/48] [kswapd+0/192]

Sep 4 08:57:27 sfb1 kernel: Call Trace:    [<c012b568>] [<c012b4e0>]
[c0105000>] [<c0107086>] [<c012b4e0>]

Sep 4 08:57:27 sfb1 kernel: bdflush       S C030FC20  6640  5  1 6 4 (L-TLB)

Sep 4 08:57:27 sfb1 kernel: Call Trace:    [interruptible_sleep_on+60/96]
[bdflush+168/176] [stext+0/48] [kernel_thread+38/48] [bdflush+0/176]

Sep 4 08:57:27 sfb1 kernel: Call Trace:    [<c011262c>] [<c01350e8>]
[c0105000>] [<c0107086>] [<c0135040>]

Sep 4 08:57:27 sfb1 kernel: kupdated      S 00200286  5892  6  1 7 5 (L-TLB)
Magic SysRq keys

- Back trace without symbols
- How to fix
  - klogd with -x option and this Omits EIP translation and therefore doesn't read the System.map file.
  - System.map file in /boot
back trace

bash S CA4FC780 0 1258 1256 1276 1257 (NOTLB)
Call Trace: [sys_wait4+900/960] [system_call+51/64]
Call Trace: [<c0117bd4>] [<c0108873>]

cp D 00000080 0 1276 1258 (NOTLB)
Call Trace: [txBegin+165/752] [jfs_truncate_nolock+198/304]
[jfs_truncate+65/91] [jfs_truncate+0/91] [vmtruncate+251/288]
Call Trace: [<c019a665>] [<c017e126>] [<c017e1d1>] [<c017e190>]
[c012260b]
    [inode_setattr+38/224] [notify_change+156/256] [cached_lookup+16/80]
[do_truncate+70/96] [open_namei+1013/1328] [dentry_open+227/400]
    [<c0144346>] [<c01444fc>] [<c0139d90>] [<c012fad6>] [<c013b0a5>]
    [<c0130a63>]

The back trace shows that txBegin is place to start looking
Questions
Set of utilities and kernel patch that allow crash dump to be captured

LKCD must be installed before a failure occurs!

When is a crash dump taken?
- A kernel Oops occurs
- A kernel panic occurs
- Administrator initiates a crash dump (Alt+SysRq+c)

Version 4 of LKCD supports following architectures
- i386
- ia64
- alpha

4.1-1 is latest release
Home page

Add kernel patch
- `cd /usr/src/linux-2.4.18`
- `patch -p1 --dry-run < /usr/src/lkcd/lkcd-4.1-1-2.4.18.patch`
  - check if the patch goes on cleanly
- `patch -p1 < /usr/src/lkcd/lkcd-4.1-1-2.4.18.patch`

Kernel hacking section
- Linux Kernel Crash Dump (LKCD) support (turn on)
- rebuild the kernel
- `cp Kerntypes /boot`

Check to see if `/proc/sys/dump` exists
- `ls -d /proc/sys/dump`
Get the Linux Kernel Crash Dump utilities

- lkcdutils-4.1-1.src.rpm
- tar zxvf lkcdutils-4.1-1.tar.gz
- cd lkcdutils-4.1

Build the lkcdutils

- ./configure
- make
- make install

Edit system startup scripts to configure LKCD and save crash dumps

- Add startup script
  - /sbin/lkc config
  - /sbin/lkc save
Linux Kernel Crash Dump (LKCD)

- Last step is to configure your dump device
  - example use the /dev/hdb1 disk partition as dump device
  - symbolic link to this partition
    - `ln -s /dev/hdb1 /dev/vmdump`

- Update the kernel with this new device
  - `/sbin/lkcd config`
Linux Kernel Crash Dump (LKCD)

<4>Pid: 4315, comm: pdosd
<4>EIP: 0010:[<c01145f3>] CPU: 0
<4>EIP is at __wake_up [kernel] 0x4f (2.4.18-3lcrash)
<4> EFLAGS: 00000286 Tainted: PF
<4>EAX: c79c122c EBX: c79c122c ECX: c204c02c EDX: c204c000
<4>ESI: 00000001 EDI: c79c1228 EBP: c204de4c DS: 0018 ES: 0018
<4>CR0: 8005003b CR2: 40013000 CR3: 04c0d000 CR4: 00000010
<4>Call Trace: [<c880f90a>] do_get_write_access [jbd] 0x10a
<4>[<c880fd01>] journal_get_write_access_R5a493269 [jbd] 0x35
<4>[<c8821a9c>] ext3_reserve_inode_write [ext3] 0x30
<4>[<c8821b28>] ext3_mark_inode_dirty [ext3] 0x18
<4>[<c8823953>] ext3_unlink [ext3] 0x14b
<4>[<c8822569>] ext3_lookup [ext3] 0x71
<4>[<c013e848>] vfs_permission [kernel] 0x78
<4>[<c014085f>] vfs_unlink [kernel] 0x147
<4>[<c013f7ca>] lookup_hash [kernel] 0x6a
<4>[<c0140936>] sys_unlink [kernel] 0x96
<4>[<c887dd26>] nct_unlink [kaznmod_BASE] 0x4a
<4>[<c887dd57>] nct_unlink [kaznmod_BASE] 0x7b
<4>[<c01085f7>] system_call [kernel] 0x33
PID: 1254, comm: pdosaudtitd
EIP: 0010:[<c0114832>] CPU: 0
EIP is at sleep_on [kernel] 0x4a (2.4.18-3lcrash)
EFLAGS: 00000286 Tainted: PF
EAX: c79c122c EBX: c79c122c ECX: 00000000 EDX: c204de3c
ESI: 00000286 EDI: c7f77000 EBP: c4541df0 DS: 0018 ES: 0018
Call Trace: [<c880faa2>] do_get_write_access [jbd] 0x2a2
[<c880fd01>] journal_get_write_access_R5a493269 [jbd] 0x35
[<c8821a9c>] ext3_reserve_inode_write [ext3] 0x30
[<c88177f0>] .rodata.str1.1 [jbd] 0x30
[<c8815317>] __jbd_kmalloc [jbd] 0x1b
[<c8821b28>] ext3_mark_inode_dirty [ext3] 0x18
[<c8821bc7>] ext3_dirty_inode [ext3] 0x83
[<c0147b76>] __mark_inode_dirty [kernel] 0x2e
[<c0149065>] update_atime [kernel] 0x51
[<c0128520>] do_generic_file_read [kernel] 0x4d0
[<c0128520>] generic_file_read [kernel] 0x9d
[<c01286f4>] file_read_actor [kernel] 0x0
[<c01360c9>] sys_read [kernel] 0x95
This trace shows that you have dead lock in `do_get_write_access [jbd]` which is journaling part for the ext3 file system. The source for `do_get_write_access` is in `/usr/src/linux/fs/jbd/transaction.c`. 