```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE book PUBLIC "-//OASIS//DTD DocBook XML V4.1.2//EN"
        "http://www.oasis-open.org/docbook/xml/4.1.2/docbookx.dtd" []>

<book id="Linux-filesystems-API">
 <bookinfo>
  <title>Linux Filesystems API</title>

  <legalnotice>
   <para>
     This documentation is free software; you can redistribute
     it and/or modify it under the terms of the GNU General Public
     License as published by the Free Software Foundation; either
     version 2 of the License, or (at your option) any later
     version.
   </para>

   <para>
     This program is distributed in the hope that it will be
     useful, but WITHOUT ANY WARRANTY; without even the implied
     warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
     See the GNU General Public License for more details.
   </para>

   <para>
     You should have received a copy of the GNU General Public
     License along with this program; if not, write to the Free
     Software Foundation, Inc., 59 Temple Place, Suite 330, Boston,
     MA 02111-1307 USA
   </para>

   <para>
     For more details see the file COPYING in the source
     distribution of Linux.
   </para>
  </legalnotice>
 </bookinfo>

<toc></toc>

  <chapter id="vfs">
     <title>The Linux VFS</title>
     <sect1 id="the_filesystem_types"><title>The Filesystem types</title>
!Iinclude/linux/fs.h
     </sect1>
     <sect1 id="the_directory_cache"><title>The Directory Cache</title>
!Efs/dcache.c
!Iinclude/linux/dcache.h
     </sect1>
     <sect1 id="inode_handling"><title>Inode Handling</title>
!Efs/inode.c
!Efs/bad_inode.c
     </sect1>
     <sect1 id="registration_and_superblocks"><title>Registration and
Superblocks</title>
!Efs/super.c
```

```
     </sect1>
     <sect1 id="file_locks"><title>File Locks</title>
!Efs/locks.c
!Ifs/locks.c
     </sect1>
     <sect1 id="other_functions"><title>Other Functions</title>
!Efs/mpage.c
!Efs/namei.c
!Efs/buffer.c
!Efs/bio.c
!Efs/seq_file.c
!Efs/filesystems.c
!Efs/fs-writeback.c
!Efs/block_dev.c
     </sect1>
  </chapter>

  <chapter id="proc">
     <title>The proc filesystem</title>

     <sect1 id="sysctl_interface"><title>sysctl interface</title>
!Ekernel/sysctl.c
     </sect1>

     <sect1 id="proc_filesystem_interface"><title>proc filesystem
interface</title>
!Ifs/proc/base.c
     </sect1>
  </chapter>

  <chapter id="sysfs">
     <title>The Filesystem for Exporting Kernel Objects</title>
!Efs/sysfs/file.c
!Efs/sysfs/symlink.c
!Efs/sysfs/bin.c
  </chapter>

  <chapter id="debugfs">
     <title>The debugfs filesystem</title>

     <sect1 id="debugfs_interface"><title>debugfs interface</title>
!Efs/debugfs/inode.c
!Efs/debugfs/file.c
     </sect1>
  </chapter>

  <chapter id="LinuxJDBAPI">
  <chapterinfo>
  <title>The Linux Journalling API</title>

  <authorgroup>
  <author>
     <firstname>Roger</firstname>
     <surname>Gammans</surname>
     <affiliation>
     <address>
```

```
      <email>rgammans@computer-surgery.co.uk</email>
     </address>
    </affiliation>
     </author>
  </authorgroup>

  <authorgroup>
   <author>
    <firstname>Stephen</firstname>
    <surname>Tweedie</surname>
    <affiliation>
     <address>
      <email>sct@redhat.com</email>
     </address>
    </affiliation>
   </author>
  </authorgroup>

  <copyright>
   <year>2002</year>
   <holder>Roger Gammans</holder>
  </copyright>
  </chapterinfo>

  <title>The Linux Journalling API</title>

    <sect1 id="journaling_overview">
     <title>Overview</title>
    <sect2 id="journaling_details">
     <title>Details</title>
<para>
The journalling layer is  easy to use. You need to
first of all create a journal_t data structure. There are
two calls to do this dependent on how you decide to allocate the physical
media on which the journal resides. The journal_init_inode() call
is for journals stored in filesystem inodes, or the journal_init_dev()
call can be use for journal stored on a raw device (in a continuous range
of blocks). A journal_t is a typedef for a struct pointer, so when
you are finally finished make sure you call journal_destroy() on it
to free up any used kernel memory.
</para>

<para>
Once you have got your journal_t object you need to 'mount' or load the journal
file, unless of course you haven't initialised it yet - in which case you
need to call journal_create().
</para>

<para>
Most of the time however your journal file will already have been created, but
before you load it you must call journal_wipe() to empty the journal file.
Hang on, you say , what if the filesystem wasn't cleanly umount()'d . Well, it
is the
job of the client file system to detect this and skip the call to
journal_wipe().
</para>
```

<para>
In either case the next call should be to journal_load() which prepares the
journal file for use. Note that journal_wipe(..,0) calls journal_skip_recovery()
for you if it detects any outstanding transactions in the journal and similarly
journal_load() will call journal_recover() if necessary.
I would advise reading fs/ext3/super.c for examples on this stage.
[RGG: Why is the journal_wipe() call necessary - doesn't this needlessly
complicate the API. Or isn't a good idea for the journal layer to hide
dirty mounts from the client fs]
</para>

<para>
Now you can go ahead and start modifying the underlying
filesystem. Almost.
</para>

<para>

You still need to actually journal your filesystem changes, this
is done by wrapping them into transactions. Additionally you
also need to wrap the modification of each of the buffers
with calls to the journal layer, so it knows what the modifications
you are actually making are. To do this use journal_start() which
returns a transaction handle.
</para>

<para>
journal_start()
and its counterpart journal_stop(), which indicates the end of a transaction
are nestable calls, so you can reenter a transaction if necessary,
but remember you must call journal_stop() the same number of times as
journal_start() before the transaction is completed (or more accurately
leaves the update phase). Ext3/VFS makes use of this feature to simplify
quota support.
</para>

<para>
Inside each transaction you need to wrap the modifications to the
individual buffers (blocks). Before you start to modify a buffer you
need to call journal_get_{create,write,undo}_access() as appropriate,
this allows the journalling layer to copy the unmodified data if it
needs to. After all the buffer may be part of a previously uncommitted
transaction.
At this point you are at last ready to modify a buffer, and once
you are have done so you need to call journal_dirty_{meta,}data().
Or if you've asked for access to a buffer you now know is now longer
required to be pushed back on the device you can call journal_forget()
in much the same way as you might have used bforget() in the past.
</para>

<para>
A journal_flush() may be called at any time to commit and checkpoint
all your transactions.
</para>

<para>
Then at umount time , in your put_super() (2.4) or write_super() (2.5)
you can then call journal_destroy() to clean up your in-core journal object.
</para>

<para>
Unfortunately there a couple of ways the journal layer can cause a deadlock.
The first thing to note is that each task can only have
a single outstanding transaction at any one time, remember nothing
commits until the outermost journal_stop(). This means
you must complete the transaction at the end of each file/inode/address
etc. operation you perform, so that the journalling system isn't re-entered
on another journal. Since transactions can't be nested/batched
across differing journals, and another filesystem other than
yours (say ext3) may be modified in a later syscall.
</para>

<para>
The second case to bear in mind is that journal_start() can
block if there isn't enough space in the journal for your transaction
(based on the passed nblocks param) - when it blocks it merely(!) needs to
wait for transactions to complete and be committed from other tasks,
so essentially we are waiting for journal_stop(). So to avoid
deadlocks you must treat journal_start/stop() as if they
were semaphores and include them in your semaphore ordering rules to prevent
deadlocks. Note that journal_extend() has similar blocking behaviour to
journal_start() so you can deadlock here just as easily as on journal_start().
</para>

<para>
Try to reserve the right number of blocks the first time. ;-). This will
be the maximum number of blocks you are going to touch in this transaction.
I advise having a look at at least ext3_jbd.h to see the basis on which
ext3 uses to make these decisions.
</para>

<para>
Another wriggle to watch out for is your on-disk block allocation strategy.
why? Because, if you undo a delete, you need to ensure you haven't reused any
of the freed blocks in a later transaction. One simple way of doing this
is make sure any blocks you allocate only have checkpointed transactions
listed against them. Ext3 does this in ext3_test_allocatable().
</para>

<para>
Lock is also providing through journal_{un,}lock_updates(),
ext3 uses this when it wants a window with a clean and stable fs for a moment.
eg.
</para>

<programlisting>

```
journal_lock_updates() //stop new stuff happening..
journal_flush()        // checkpoint everything.
..do stuff on stable fs
journal_unlock_updates() // carry on with filesystem use.
```

```
</programlisting>

<para>
The opportunities for abuse and DOS attacks with this should be obvious,
if you allow unprivileged userspace to trigger codepaths containing these
calls.
</para>

<para>
A new feature of jbd since 2.5.25 is commit callbacks with the new
journal_callback_set() function you can now ask the journalling layer
to call you back when the transaction is finally committed to disk, so that
you can do some of your own management. The key to this is the journal_callback
struct, this maintains the internal callback information but you can
extend it like this:-
</para>
<programlisting>
        struct  myfs_callback_s {
                //Data structure element required by jbd..
                struct journal_callback for_jbd;
                // Stuff for myfs allocated together.
                myfs_inode*    i_commited;

        }
</programlisting>

<para>
this would be useful if you needed to know when data was committed to a
particular inode.
</para>

    </sect2>

    <sect2 id="jbd_summary">
     <title>Summary</title>
<para>
Using the journal is a matter of wrapping the different context changes,
being each mount, each modification (transaction) and each changed buffer
to tell the journalling layer about them.
</para>

<para>
Here is a some pseudo code to give you an idea of how it works, as
an example.
</para>

<programlisting>
  journal_t* my_jnrl = journal_create();
  journal_init_{dev,inode}(jnrl,...)
  if (clean) journal_wipe();
  journal_load();

   foreach(transaction) { /*transactions must be
                           completed before
                           a syscall returns to
                           userspace*/
```

```
          handle_t * xct=journal_start(my_jnrl);
          foreach(bh) {
                journal_get_{create,write,undo}_access(xact,bh);
                if ( myfs_modify(bh) ) { /* returns true
                                          if makes changes */
                          journal_dirty_{meta,}data(xact,bh);
                } else {
                          journal_forget(bh);
                }
          }
          journal_stop(xct);
   }
   journal_destroy(my_jrnl);
</programlisting>
    </sect2>

    </sect1>

    <sect1 id="data_types">
     <title>Data Types</title>
     <para>
        The journalling layer uses typedefs to 'hide' the concrete definitions
        of the structures used. As a client of the JBD layer you can
        just rely on the using the pointer as a magic cookie  of some sort.

        Obviously the hiding is not enforced as this is 'C'.
     </para>
        <sect2 id="structures"><title>Structures</title>
!Iinclude/linux/jbd.h
        </sect2>
    </sect1>

    <sect1 id="functions">
     <title>Functions</title>
     <para>
        The functions here are split into two groups those that
        affect a journal as a whole, and those which are used to
        manage transactions
     </para>
        <sect2 id="journal_level"><title>Journal Level</title>
!Efs/jbd/journal.c
!Ifs/jbd/recovery.c
        </sect2>
        <sect2 id="transaction_level"><title>Transasction Level</title>
!Efs/jbd/transaction.c
        </sect2>
    </sect1>
    <sect1 id="see_also">
     <title>See also</title>
        <para>
          <citation>
           <ulink
url="ftp://ftp.uk.linux.org/pub/linux/sct/fs/jfs/journal-design.ps.gz">
                Journaling the Linux ext2fs Filesystem, LinuxExpo 98, Stephen
Tweedie
```

```
          </ulink>
         </citation>
        </para>
        <para>
          <citation>
          <ulink
url="http://olstrans.sourceforge.net/release/OLS2000-ext3/OLS2000-ext3.html">
               Ext3 Journalling FileSystem, OLS 2000, Dr. Stephen Tweedie
          </ulink>
          </citation>
        </para>
    </sect1>

  </chapter>

  <chapter id="splice">
      <title>splice API</title>
  <para>
        splice is a method for moving blocks of data around inside the
        kernel, without continually transferring them between the kernel
        and user space.
  </para>
!Ffs/splice.c
  </chapter>

  <chapter id="pipes">
      <title>pipes API</title>
  <para>
        Pipe interfaces are all for in-kernel (builtin image) use.
        They are not exported for use by modules.
  </para>
!Iinclude/linux/pipe_fs_i.h
!Ffs/pipe.c
  </chapter>

</book>
```