Linux Writeback Queues

Wu Fengguang
<wfg@linux.intel.com>

November 4, 2008

Outline

1. **writeback queues**
   - bugs => solutions => principles
2. **writeback policies**
   - location ordering
   - lazy writeback

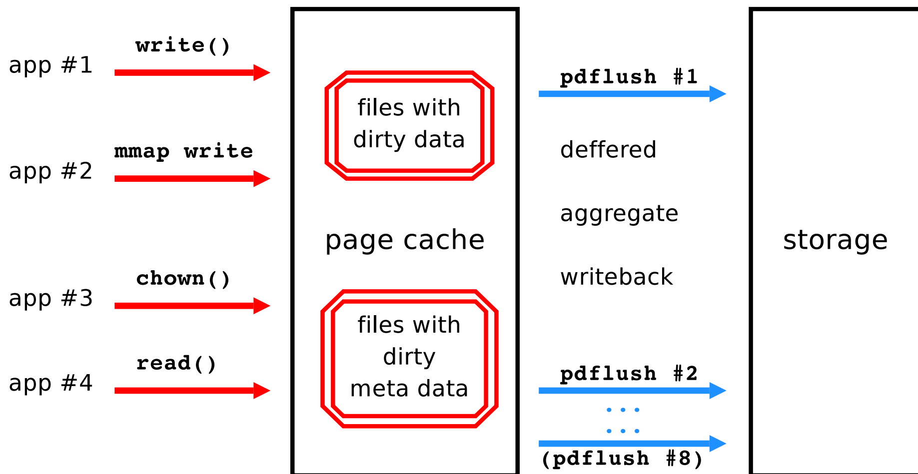
the writeback mess

```
linux-2.6.22 $ wc -l fs/fs-writeback.c mm/page-writeback.c
  704 fs/fs-writeback.c
 1027 mm/page-writeback.c
 1731 total
```

That code does so many different things it ain't funny. This is why when one thing gets changed, something else gets broken.

- Andrew Morton

page cache writeback



pdflush threads

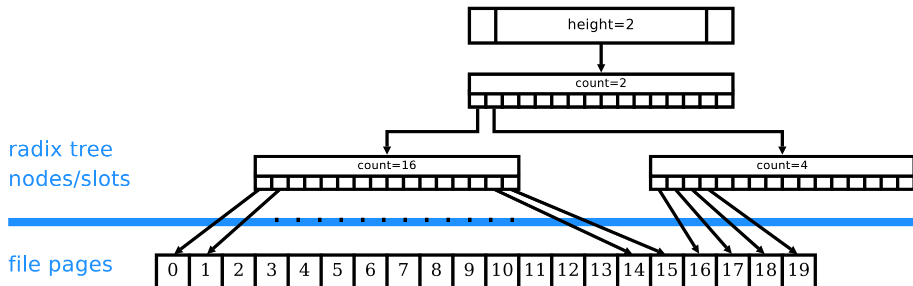
2∼8 kernel threads, writing back dirty data in background

USER	PID	%CPU	%MEM	VSZ	RSS	TTY	STAT	START	TIME	COMMAND
root	198	0.0	0.0	0	0	?	S	08:21	0:00	[pdflush]
root	199	0.0	0.0	0	0	?	S	08:21	0:00	[pdflush]

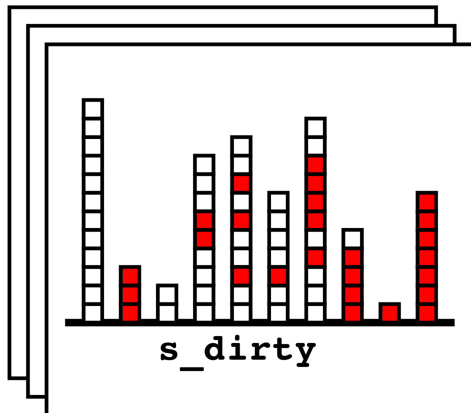
expected function

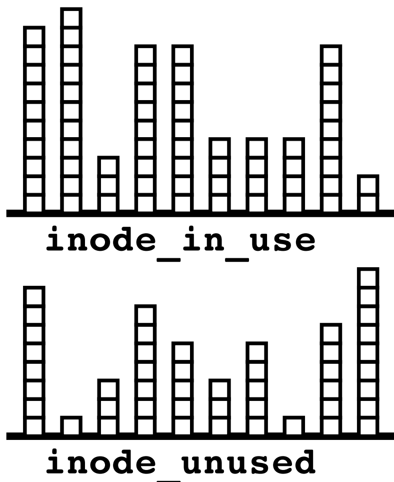
```
on every 5s:
    for each dirty old inode (i.e. now - dirtied_when > 30s):
        sync inode meta-data & pages to disk

 5s: tunable via /proc/sys/vm/dirty_writeback_centisecs
30s: tunable via /proc/sys/vm/dirty_expire_centisecs
```

page cache organization



inode list organization



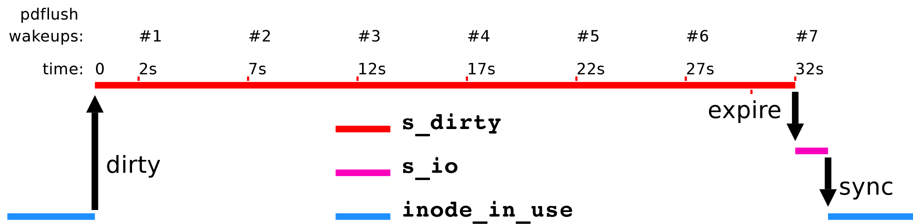


writeback iterations

Writeback task as 3-level loops:

```
for all super_blocks
  for all dirty inodes (expired)
    for all dirty pages
      sync page
```

dirty-expire-sync time line



2-queue park-splice-work model

data structure

```
two /ordered/ queues per superblock:
- s_dirty: park imporing dirty inodes
- s_io:    hold dirty inodes to be worked in this wakeup
```

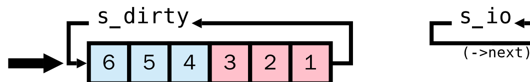
work flow

```
0) s_dirty accepting newly dirtied inodes (as always!)

1) splice s_dirty to s_io
2) iterate through *old* dirty inodes in s_io for writing back
3) splice remained young dirty inodes in s_io back to s_dirty
```

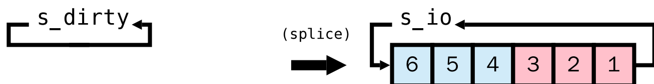
2-queue park-splice-work model: example

pdflush status

0) sleeping:

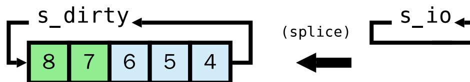


1) wake up:



2) syncing:

3) done:



fairness issue: large file delays small files

- file 1 is 1GB
- file 2,3 are 1KB

- file 1 will delay file 2,3 for too long time.

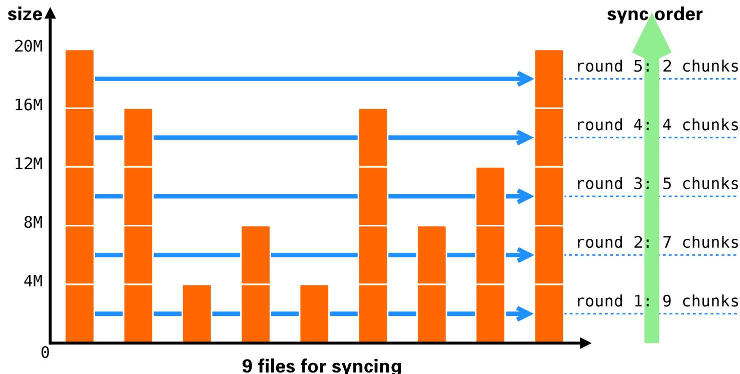
live-lock issue: busy file blocks other files

- file 1 is being written to fast enough

- the following files could be blocked for a long time, if not for ever.

solution: striped syncing

- **sync at most 4MB/file at a time**
- **move partially-synced files to the tail of `s_dirty`, so that they can be served in the next round**



lack-of-writeback problem

LKML message from David Chinner <dgc@sgi.com> on Feb 2006:

```
The workload involves ~16 postmark threads running in the background
each creating ~15m subdirectories of ~1m files each. The idea is
that this generates a nice, steady background file creation load.
Each file is between 1-10k in size, and it runs at 3-5k creates/s.

The disk subsystem is nowhere near I/O bound - the luns are less than 10%
busy when running this workload, and only writing about 30-40MB/s aggregate.

The problem comes when I run another thread that writes a large single
file to disk. e.g.:

# dd if=/dev/zero of=/mnt/dgc/stripe/testfile bs=1024k count=4096

to write out a 4GB file. Now this goes straight into memory (takes
about 7-8s) with some writeback occurring. The result is that approximately
2.5GB of the file is still dirty in memory.

It then takes over an hour to write the remaining data to disk. The
pattern of writeback appears to be that roughly every
dirty_expire_centisecs a chunk of 1024 pages (16MB on altix) are
written to for that large file, and it is done in a single flush.
```

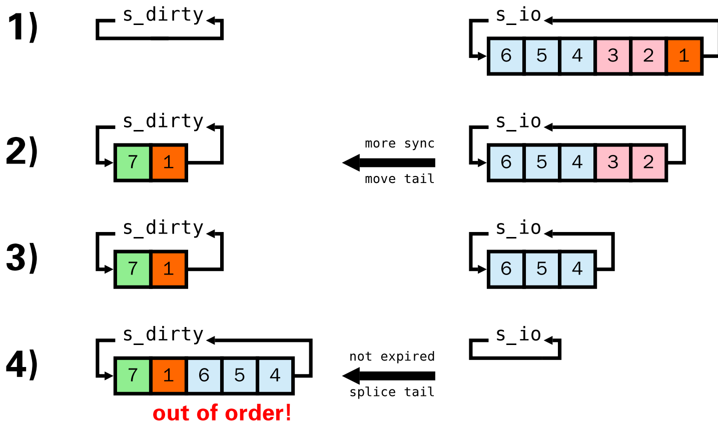
lack-of-writeback problem: short version

- large dirtied file
- continuously emerging small dirty files

- large file writeback slowed down to 1024 pages / 30 sec

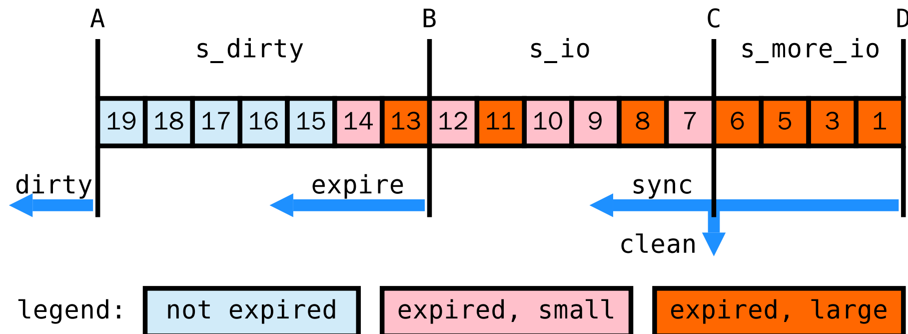
lack-of-writeback problem: reasoning

- s_dirty goes out-of-order when (4) follows (2),
- file 1 blocked until expiration of file 6.



lack-of-writeback problem: proposal

inodes need more work $\Longrightarrow$ `s_more_io`



writeback code

```
splice s_more_io to s_io.tail
splice s_dirty   to s_io

for each inode on s_io
        if young, break
        writeback inode
                if inode needs more work, put on s_more_io
```

writeback code - more complete

```
1 + loop
2 +     nr_to_write = MAX_WRITEBACK_PAGES(=1024)
3 +     for each super block
4 +         if s_io is empty
5               splice s_more_io to s_io.tail
6               move expired s_dirty inodes to s_io
7           for each inode on s_io
8               writeback up to nr_to_write pages
9               if inode needs more work, put on s_more_io
10 +            if pages skipped, put back to s_dirty
11 +            nr_to_write -= pages written
12 +            break if nr_to_write <= 0
13 +        break if nr_to_write <= 0
14 +    break if nr_to_write > 0
```

line 4: starvation/livelock prevention
line 10/14: source of the next two bugs

starvation/live-lock prevention

Let A = large file; B,C,D,E,F,G,. . . = small files.

no line 4

```
sync 4MB of A; draw more files;
sync 4MB of A; draw more files;
sync 4MB of A; draw more files;
sync 4MB of A; draw more files;
sync 4MB of A; draw more files;
...  // A starves the following files
sync B,C,D,E,F,G,...; draw more files
...  // livelock on imporing expired dirty files
```

with line 4

```
sync 4MB of A; sync B,C;
sync 4MB of A; sync D,E;
sync 4MB of A; sync F,G;
...
done with this superblock's dataset
```

lack-of-writeback-2 problem

```
% ls -his /var/x
847824 200M /var/x

% dmesg|grep 847824  # generated by a debug printk
[  529.263184] redirtied inode 847824 line 548
[  564.250872] redirtied inode 847824 line 548
[  759.198568] redirtied inode 847824 line 548

# line 548 in fs/fs-writeback.c:
543                 if (wbc->pages_skipped != pages_skipped)
544                         /*
545                          * writeback is not making progress due to locked
546                          * buffers.  Skip this inode for now.
547                          */
548                         redirty_tail(inode);
549

More debug efforts show that __block_write_full_page()
never has the chance to call submit_bh() for that big dirty file:
the buffer head is *clean*. So basicly no page io is issued by
__block_write_full_page(), hence pages_skipped goes up.
```

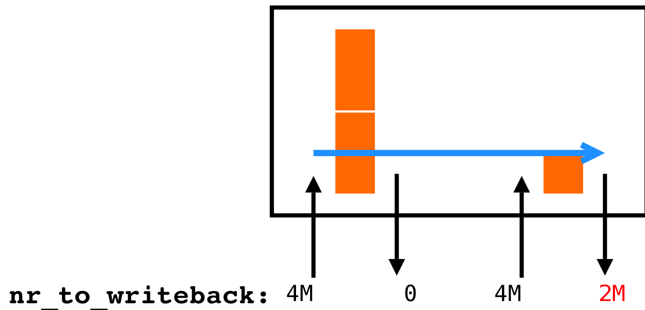
lack-of-writeback-2 solution

```
the comment in __block_write_full_page():
1713        /*
1714         * The page was marked dirty, but the buffers were
1715         * clean.  Someone wrote them back by hand with
1716         * ll_rw_block/submit_bh.  A rare case.
1717         */
```

`pages_skipped` **accounts 'locked buffer', but here is 'clean buffer'!**
So fix it with:

```
--- linux-2.6.23-rc2-mm2.orig/fs/buffer.c
+++ linux-2.6.23-rc2-mm2/fs/buffer.c
@@ -1713,7 +1713,6 @@ done:
             * The page and buffer_heads can be released at any time from
             * here on.
             */
-           wbc->pages_skipped++;   /* We didn't write this page */
    }
    return err;
```

lack-of-writeback-3



lack-of-writeback-3 problem

```
          s_io            s_more_io
          --------------------------
1)        100M,1K         0
2)        1K              96M
3)        0               96M

1) initial state with a 100M file and a 1K file
2) 4M written, nr_to_write <= 0, so write more
3) 1K written, nr_to_write > 0, no more writes (BUG!)
```

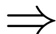
nr_to_write > 0 in (3) fools upper layer to think that data have all been written out. The big dirty file is actually still sitting in s_more_io.

We have to return when a full scan of s_io completes.
So **nr_to_write** > 0 does not necessarily mean that "all data are written".

lack-of-writeback-3 solution

Introduce `more_io` flag to indicate that some superblock(s)

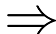
- **have more work in `s_more_io`;**
- **temporarily yielded to give other superblocks a chance.**



- **files get synced fast, finally! (?)**
- **ext2/reiserfs/jfs goes 100% iowait ...**

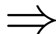
dirty flag & tag - rationale

- LRU reclaim triggers page-by-page writeback



- address space sequential writeback is desired

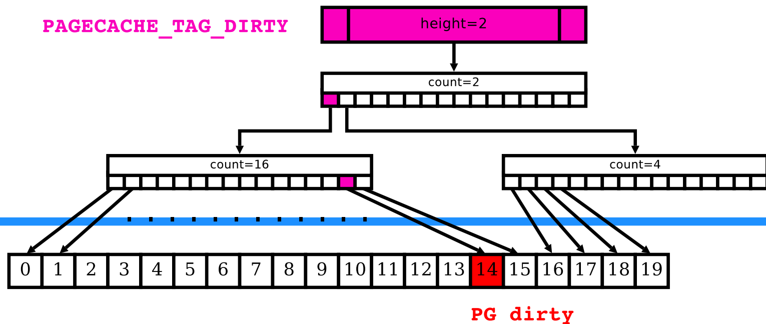
- tag dirty pages in radix tree to speedup lookup



dirty pages are

- defined by `PG_dirty`
- searchable by `PAGECACHE_TAG_DIRTY`

dirty flag & tag - demo



dirty flag & tag - protocol

```
redirty_page_for_writepage
  __set_page_dirty_nobuffers
                              if !TestSetPageDirty
                                      set PAGECACHE_TAG_DIRTY

set_page_writeback
  test_set_page_writeback
                              if !TestSetPageWriteback
                                      set PAGECACHE_TAG_WRITEBACK
                              if !PageDirty
                                      clear PAGECACHE_TAG_DIRTY

end_page_writeback
  test_clear_page_writeback
                              if TestClearPageWriteback
                                     clear PAGECACHE_TAG_WRITEBACK
```

dirty flag & tag - example

```
set_page_dirty (page is locked except called from zap_pte_range())
                                     PG_dirty
               (infinite small time)
                                     PG_dirty PAGECACHE_TAG_DIRTY
```

⟹ (triggers writeback some time later)

```
lock_page
                           PG_locked PG_dirty PAGECACHE_TAG_DIRTY
clear_page_dirty_for_io
                           PG_locked          PAGECACHE_TAG_DIRTY
set_page_writeback
                           PG_locked PG_writeback PAGECACHE_TAG_WRITEBACK
unlock_page (wait for io)
                                     PG_writeback PAGECACHE_TAG_WRITEBACK
end_page_writeback
```

dirty flag & tag - redirty examples

```
(a) set_page_dirty
    lock_page
    clear_page_dirty_for_io
    redirty_page_for_writepage
    set_page_writeback
    unlock_page
    end_page_writeback
```

```
(b) set_page_dirty
    lock_page
    clear_page_dirty_for_io
    set_page_writeback
    redirty_page_for_writepage
    unlock_page
    end_page_writeback
```

```
---------------------------------------------------------------------------------

(a)           PG_dirty PAGECACHE_TAG_DIRTY
    PG_locked PG_dirty PAGECACHE_TAG_DIRTY
    PG_locked          PAGECACHE_TAG_DIRTY
    PG_locked PG_dirty PAGECACHE_TAG_DIRTY
    PG_locked PG_dirty PAGECACHE_TAG_DIRTY PG_writeback PAGECACHE_TAG_WRITEBACK
              PG_dirty PAGECACHE_TAG_DIRTY

(b)           PG_dirty PAGECACHE_TAG_DIRTY
    PG_locked PG_dirty PAGECACHE_TAG_DIRTY
    PG_locked          PAGECACHE_TAG_DIRTY
    PG_locked                              PG_writeback PAGECACHE_TAG_WRITEBACK
    PG_locked PG_dirty PAGECACHE_TAG_DIRTY PG_writeback PAGECACHE_TAG_WRITEBACK
              PG_dirty PAGECACHE_TAG_DIRTY
```

reiserfs bug tracing

The page has both PG_dirty(D)/PAGECACHE_TAG_DIRTY(d) after being written to; and then only PAGECACHE_TAG_DIRTY(d) remains after the file is closed.

```
[T0] /home/wfg# cat > /test/tiny
[T1] hi
[T2] /home/wfg#
```

```
[T1] /home/wfg# echo /test/tiny > /proc/filecache
[T1] /home/wfg# cat /proc/filecache
     # file /test/tiny
     # flags U:PG_uptodate D:PG_dirty B:PG_buffer d:PAGECACHE_TAG_DIRTY
     # idx    len     state      refcnt
     0        1       ___UD__Bd_      2
[T2] /home/wfg# tail -2 /proc/filecache
     # idx    len     state      refcnt
     0        1       ___U___Bd_      2
```

reiserfs bug tracing (cont.)

Notice the non-zero 'cancelled_write_bytes' after /tmp/hi is copied.

```
[T0] /home/wfg# echo hi > /tmp/hi
[T1] /home/wfg# cp /tmp/hi /dev/stdin /test
[T2] hi
[T3] /home/wfg#
```

```
[T2] /proc/4397# cd /proc/`pidof cp`
[T2] /proc/4713# cat io
     rchar: 8396
     wchar: 3
     syscr: 20
     syscw: 1
     read_bytes: 0
     write_bytes: 20480
     cancelled_write_bytes: 4096
```

```
[T3] /proc/4713# cat io
     rchar: 8399
     wchar: 6
     syscr: 21
     syscw: 2
     read_bytes: 0
     write_bytes: 24576
     cancelled_write_bytes: 4096
```

reiserfs bug summary

Reiserfs could accumulate dirty sub-page-size files until umount time. They cannot be synced to disk by pdflush routines or explicit ‘sync’ commands. Only ‘umount’ can do the trick.

The direct cause is: the dirty page’s PG_dirty is wrongly _cleared_.

Call trace:

```
    [<ffffffff8027e920>] cancel_dirty_page+0xd0/0xf0
    [<ffffffff8816d470>] :reiserfs:reiserfs_cut_from_item+0x660/0x710
    [<ffffffff8816d791>] :reiserfs:reiserfs_do_truncate+0x271/0x530
    [<ffffffff8815872d>] :reiserfs:reiserfs_truncate_file+0xfd/0x3b0
    [<ffffffff8815d3d0>] :reiserfs:reiserfs_file_release+0x1e0/0x340
    [<ffffffff802a187c>] __fput+0xcc/0x1b0
    [<ffffffff802a1ba6>] fput+0x16/0x20
    [<ffffffff8029e676>] filp_close+0x56/0x90
    [<ffffffff8029fe0d>] sys_close+0xad/0x110
    [<ffffffff8020c41e>] system_call+0x7e/0x83
```

Fixed the bug by removing the cancel_dirty_page() call.

ext2 bug tracing

```
Tough task: cannot reproduce...
Wrote a kernel module for end user, do jprobes on requeue_io() to print inode info.

Get this:
          inode 114019(sda7/.kde) count 2,2 size 0 pages 1
          0     2       0       U____
          inode 114025(sda7/cache-ibm) count 2,1 size 0 pages 1
          0     2       0       U____
          inode 114029(sda7/socket-ibm) count 2,3 size 0 pages 1
          0     2       0       U____
          inode 114017(sda7/0266584877) count 3,6 size 0 pages 1
          0     2       0       U____

The .kde/cache-ibm/socket-ibm/0266584877 above are confirmed to be directories.

To reproduce:
                console 1              console 2
                $ mkdir a; cd a
                $ touch b; rm b
                                       $ rmdir a
                _________ 100% iowait _________
                $ cd ..
```

ext2 bug fix

```
writeback: clear PAGECACHE_TAG_DIRTY for truncated page in block_write_full_page()

The 'truncated' page in block_write_full_page() may stick for a long time.
E.g. ext2_rmdir() will set i_size to 0, and then the dir inode may hang around
because of being referenced by someone.

So clear PAGECACHE_TAG_DIRTY to prevent pdflush from retrying and iowaiting on it.

Signed-off-by: Fengguang Wu <wfg@mail.ustc.edu.cn>
---

--- linux.orig/fs/buffer.c
+++ linux/fs/buffer.c
@@ -2820,7 +2820,9 @@ int block_write_full_page(struct page *p
                  * freeable here, so the page does not leak.
                  */
                 do_invalidatepage(page, 0);
+                set_page_writeback(page);
                 unlock_page(page);
+                end_page_writeback(page);
                 return 0; /* don't care */
         }
```

jfs bug fix

```
commit 29a424f28390752a4ca2349633aaacc6be494db5
Author: Dave Kleikamp <shaggy@linux.vnet.ibm.com>
Date:   Thu Jan 3 13:09:33 2008 -0600

JFS: clear PAGECACHE_TAG_DIRTY for no-write pages

When JFS decides to drop a dirty metapage, it simply clears the META_dirty
bit and leave alone the PG_dirty and PAGECACHE_TAG_DIRTY bits.

When such no-write page goes to metapage_writepage(), the ‘relic’
PAGECACHE_TAG_DIRTY tag should be cleared, to prevent pdflush from
repeatedly trying to sync them.  This is done through
set_page_writeback(), so call it should be called in all cases.  If
no I/O is initiated, end_page_writeback() should be called immediately.

This is how __block_write_full_page() does things.

Signed-off-by: Dave Kleikamp <shaggy@linux.vnet.ibm.com>
CC: Fengguang Wu <wfg@mail.ustc.edu.cn>
```

game over

More buggy fs?! Let's stop the game...

general fix

When we didn't write back all the pages...

`linux/fs/fs-writeback.c  __sync_single_inode()`

```
         inode->i_state |= I_DIRTY_PAGES;
-        requeue_io(inode);
+        if (wbc->nr_to_write <= 0) {
+                /*
+                 * slice used up: queue for next turn
+                 */
+                requeue_io(inode);
+        } else {
+                /*
+                 * somehow blocked: retry later
+                 */
+                redirty_tail(inode);
+        }
```

writeback iterations

Writeback task as 3-level loops:

```
for all super_blocks
  for all dirty inodes (expired)
    for all dirty pages
      sync page
```

root of problem: cannot de-dirty inodes in one shot

- **the dataset is highly dynamic**
  - newly/repeatedly dirtied/expired inodes
  - newly/repeatedly dirtied pages
  - gone inodes/pages
- **inodes cannot be synced for now**
  - locked inodes/pages/etc
  - io queue is congested
- **inodes should not be synced in one shot**
  - large files: starvation for small files
  - busy files: could lead to livelock
- **superblocks should not be synced in one shot**
  - fairness/livelock issues
- **cannot hold `inode_lock` for a long time**
  1. exit after writing 4MB data (break if >= `MAX_WRITEBACK_PAGES`)
  2. take a breath
  3. re-enter loop and continue from where we left

writeback iterations in practice

The real-world writeback logic is kind of

```
while(more io)
  for a in A        // iterate super_blocks
    for b in B      // iterate dirty inodes (expired)
      for c in C    // iterate dirty pages
        sync, break, re-enter and continue on every 4MB
```

where

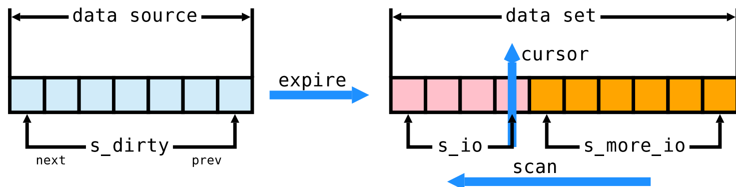
- cursors a,b,c should be properly saved
- sets A,B,C should remain stable in traversing

so that the iterations can

- continue from last position
- finish in bounded time

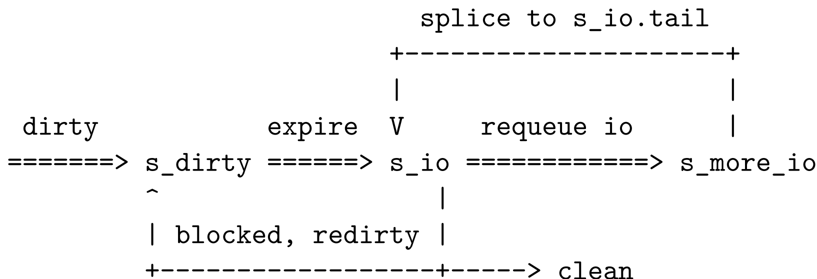
inode list iteration - model

The set, cursor and data source are:




```
- s_io.prev is cursor ‘b’, which remembers our position in the list;

- s_io+s_more_io forms set ‘B’, which is only updated to pull in new
  expired inodes from s_dirty, after one full iteration of it is completed
  *and* cursor ‘a’ has completed one full iteration on the superblocks.
```

inode list iteration - protocol



s_dirty contains expired and non-expired dirty inodes. The non-expired ones are in time-of-dirtying(`dirtied_when`) order.

s_io contains expired and non-expired dirty inodes, with expired ones at the head. Unexpired ones (at least) are in time order.

s_more_io contains dirty expired inodes which haven't been fully written. Ordering doesn't matter.

- Andrew Morton

inode list iteration - example

Suppose some 8MB dirty files Fn are to be synced. The iterations may go like this:

```
      s_more_io               s_io                                s_dirty
                              [ queue_io() ]                    [ F1-F5 expire ]
T0                            F1,F2,F3,F4,F5                      F6,F7,F8
T1    F1                         F2,F3,F4,F5                      F6,F7,F8
T2    F1,F2                         F3,F4,F5                      F6,F7,F8
T3    F1,F2,F3                         F4,F5                      F6,F7,F8
T4    F1,F2,F3,F4                         F5                      F6,F7,F8
T5    F1,F2,F3,F4,F5                                              F6,F7,F8
                              [ queue_io() ]                    [ F6-F8 expire ]
T6                            F1,F2,F3,F4,F5,F6,F7,F8
T7                               F2,F3,F4,F5,F6,F7,F8
T8                                  F3,F4,F5,F6,F7,F8
T9                                     F4,F5,F6,F7,F8
T10                                       F5,F6,F7,F8
T11                                          F6,F7,F8
T12   F6                                        F7,F8
T13   F6,F7                                        F8
T14   F6,F7,F8
                              [ queue_io() ]
T15                           F6,F7,F8
T16                              F7,F8
T17                                 F8
```

writeback objectives

- be fair
- be efficient
- KISS, as always

- collect jobs into big batches (sync every 5s)
- once triggered, send all data into io queue AFAP
- stripe files and sync in turn (will it hurt performance?)

the atime deficiency

```
sys_read() => file_accessed()
           => touch_atime()
```

```
For every file that is read from the disk, lets do a ...
write to the disk!

And, for every file that is already cached and which we
read from the cache ... do a write to the disk!

                                          - Ingo Molnar


note: write = update atime and mark the inode dirty
```

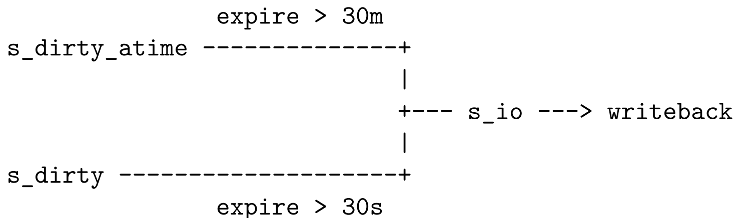
atime solutions

- **noatime / nodiratime**
  - best!
  - no luck as default
- **relatime**
  - good enough, shall not break mutt
  - ‘could’ be default

```
if (atime < ctime || atime < mtime)
        update atime
```

atime writeback improvement possibilities

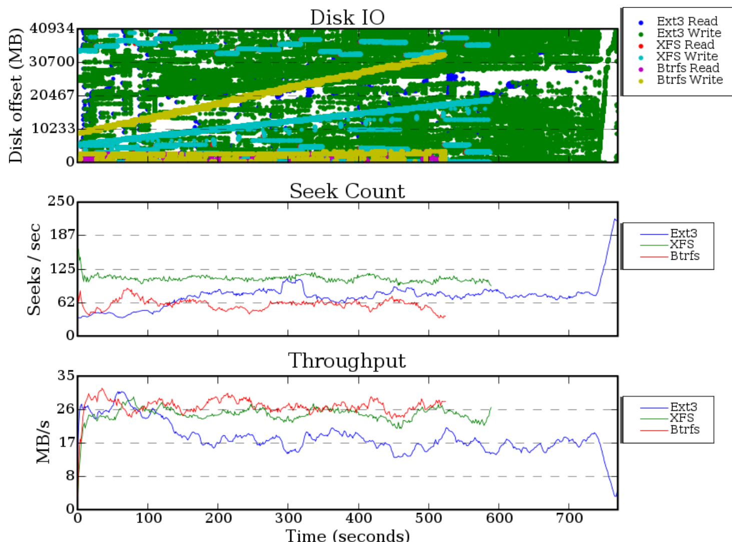
- **more atime writeback delays**



`cons: will hold more inodes in memory and increase umount time`

- **piggy back atime updates - clustered writeback**

ordering by location - why



http://oss.oracle.com/~mason/compilebench/makej/

ordering by location - key

address hint: inode number

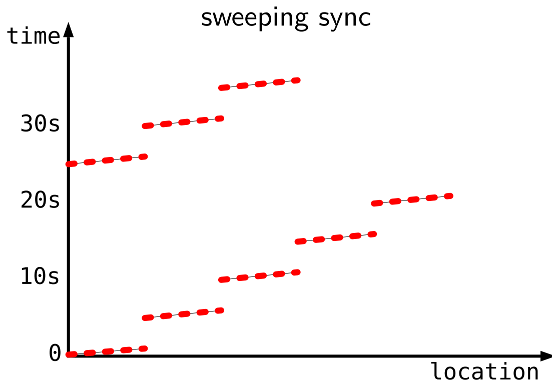
normally,

- **inode number is proportional to address of inode;**
- **data blocks for inode N will be close to inode N.**

or more loosely

- **inode N will be close to inode N+1;**
- **data blocks for inode N will be close to inode N+1.**

ordering by location - how



- **store** `s_dirty` **inodes in a radix tree, indexed by inode number**
- **sweep 1/5 of the inode number space in every 5s**
- **put inodes with** `dirty_age > 5s` **to io queue**

ordering by location - try it out

- **patchset available at**
  http://lkml.org/lkml/2007/8/27/45
- **try outs and feedbacks are warmly welcome**
- **favorable performance numbers are the key to mainline inclusion**

```
[PATCH 0/3] [RFC][PATCH] clustered writeback
[PATCH 1/3] writeback: introduce queue_dirty()
[PATCH 2/3] writeback: introduce dirty_volatile_interval
[PATCH 3/3] writeback: writeback clustering by inode number
```

Acknowledgements

- Andrew Morton
- Chris Mason
- David Chinner
- Ingo Molnar
- Ken Chen
- Michael Rubin
- Peter Zijlstra

Thank you!