GCC Code Coverage Report

			Exec	Total	Coverage
Directory:	./				
File:	storage/filesystem/littlefs/littlefs/lfs.c	**Lines:**	777	1392	55.8 %
Date:	2021-05-06 12:39:05	**Branches:**	322	852	37.8 %

Line	Branch	Exec	Source
1			/*
2			* The little filesystem
3			*
4			* Copyright (c) 2017, Arm Limited. All rights reserved.
5			* SPDX-License-Identifier: BSD-3-Clause
6			*/
7			#include "lfs.h"
8			#include "lfs_util.h"
9			
10			#include <inttypes.h>
11			
12			
13			/// Caching block device operations ///
14		115967	static int lfs_cache_read(lfs_t *lfs, lfs_cache_t *rcache,
15			const lfs_cache_t *pcache, lfs_block_t block,
16			lfs_off_t off, void *buffer, lfs_size_t size) {
17		115967	uint8_t *data = buffer;
18	✗✓	115967	LFS_ASSERT(block < lfs->cfg->block_count);
19			
20	✓✓	352390	while (size > 0) {
21	✗✓✗✗ ✗✗✗✗	120456	if (pcache && block == pcache->block && off >= pcache->off &&
22			off < pcache->off + lfs->cfg->prog_size) {
23			// is already in pcache?
24			lfs_size_t diff = lfs_min(size,
25			lfs->cfg->prog_size - (off-pcache->off));
26			memcpy(data, &pcache->buffer[off-pcache->off], diff);
27			
28			data += diff;
29			off += diff;

```
30                  size -= diff;
31                  continue;
32              }
33
34  ✓✓✓✗ 236423     if (block == rcache->block && off >= rcache->off &&
        ✓✗
35          115967              off < rcache->off + lfs->cfg->read_size) {
36                  // is already in rcache?
37          115967  lfs_size_t diff = lfs_min(size,
38          115967                  lfs->cfg->read_size - (off-rcache->off));
39          115967  memcpy(data, &rcache->buffer[off-rcache->off], diff);
40
41          115967  data += diff;
42          115967  off += diff;
43          115967  size -= diff;
44          115967  continue;
45              }
46
47  ✓✓✗✓   4489     if (off % lfs->cfg->read_size == 0 && size >= lfs->cfg->read_size) {
48                  // bypass cache?
49                  lfs_size_t diff = size - (size % lfs->cfg->read_size);
50                  int err = lfs->cfg->read(lfs->cfg, block, off, data, diff);
51                  if (err) {
52                      return err;
53                  }
54
55                  data += diff;
56                  off += diff;
57                  size -= diff;
58                  continue;
59              }
60
61              // load to cache, first condition can no longer fail
62          4489    rcache->block = block;
63          4489    rcache->off = off - (off % lfs->cfg->read_size);
64         13467    int err = lfs->cfg->read(lfs->cfg, rcache->block,
65          8978            rcache->off, rcache->buffer, lfs->cfg->read_size);
66     ✗✓   4489    if (err) {
67                      return err;
68                  }
69          }
```

```
70
71          115967      return 0;
72                      }
73
74            1014      static int lfs_cache_cmp(lfs_t *lfs, lfs_cache_t *rcache,
75                              const lfs_cache_t *pcache, lfs_block_t block,
76                              lfs_off_t off, const void *buffer, lfs_size_t size) {
77            1014          const uint8_t *data = buffer;
78
79          ✓✓ 117846       for (lfs_off_t i = 0; i < size; i++) {
80                              uint8_t c;
81             57909            int err = lfs_cache_read(lfs, rcache, pcache,
82                                      block, off+i, &c, 1);
83          ✗✓ 57909            if (err) {
84                                  return err;
85                              }
86
87          ✗✓ 57909            if (c != data[i]) {
88                                  return false;
89                              }
90                          }
91
92            1014          return true;
93                      }
94
95            3140      static int lfs_cache_crc(lfs_t *lfs, lfs_cache_t *rcache,
96                              const lfs_cache_t *pcache, lfs_block_t block,
97                              lfs_off_t off, lfs_size_t size, uint32_t *crc) {
98          ✓✓ 55895        for (lfs_off_t i = 0; i < size; i++) {
99                              uint8_t c;
100            52755            int err = lfs_cache_read(lfs, rcache, pcache,
101                                    block, off+i, &c, 1);
102         ✗✓ 52755            if (err) {
103                                 return err;
104                             }
105
106            52755            lfs_crc(crc, &c, 1);
107                         }
108
109           3140          return 0;
```

```
110                 }
111
112      1051       static inline void lfs_cache_drop(lfs_t *lfs, lfs_cache_t *rcache) {
113                     // do not zero, cheaper if cache is readonly or only going to be
114                     // written with identical data (during relocates)
115                     (void)lfs;
116      1051           rcache->block = 0xffffffff;
117      1051       }
118
119       548       static inline void lfs_cache_zero(lfs_t *lfs, lfs_cache_t *pcache) {
120                     // zero to avoid information leak
121       548           memset(pcache->buffer, 0xff, lfs->cfg->prog_size);
122       548           pcache->block = 0xffffffff;
123       548       }
124
125       429       static int lfs_cache_flush(lfs_t *lfs,
126                         lfs_cache_t *pcache, lfs_cache_t *rcache) {
127   ✓✗  429           if (pcache->block != 0xffffffff) {
128      1287               int err = lfs->cfg->prog(lfs->cfg, pcache->block,
129       858                       pcache->off, pcache->buffer, lfs->cfg->prog_size);
130   ✗✓  429               if (err) {
131                             return err;
132                         }
133
134   ✓✓  429               if (rcache) {
135       206                   int res = lfs_cache_cmp(lfs, rcache, NULL, pcache->block,
136       206                           pcache->off, pcache->buffer, lfs->cfg->prog_size);
137   ✗✓  103                   if (res < 0) {
138                                 return res;
139                             }
140
141   ✗✓  103                   if (!res) {
142                                 return LFS_ERR_CORRUPT;
143                             }
144                         }
145
146       429               lfs_cache_zero(lfs, pcache);
147                     }
148
149       429           return 0;
```

```
150                 }
151
152        1561     static int lfs_cache_prog(lfs_t *lfs, lfs_cache_t *pcache,
153                         lfs_cache_t *rcache, lfs_block_t block,
154                         lfs_off_t off, const void *buffer, lfs_size_t size) {
155        1561         const uint8_t *data = buffer;
156    ✗✓  1561         LFS_ASSERT(block < lfs->cfg->block_count);
157
158    ✓✓  5112         while (size > 0) {
159  ✓✓✓✗   3551             if (block == pcache->block && off >= pcache->off &&
       ✓✗
160        1561                     off < pcache->off + lfs->cfg->prog_size) {
161                             // is already in pcache?
162        1561                 lfs_size_t diff = lfs_min(size,
163        1561                         lfs->cfg->prog_size - (off-pcache->off));
164        1561                 memcpy(&pcache->buffer[off-pcache->off], data, diff);
165
166        1561                 data += diff;
167        1561                 off += diff;
168        1561                 size -= diff;
169
170    ✗✓  1561                 if (off % lfs->cfg->prog_size == 0) {
171                                 // eagerly flush out pcache if we fill up
172                                 int err = lfs_cache_flush(lfs, pcache, rcache);
173                                 if (err) {
174                                     return err;
175                                 }
176                             }
177
178        1561                 continue;
179                         }
180
181                         // pcache must have been flushed, either by programming and
182                         // entire block or manually flushing the pcache
183    ✗✓    429         LFS_ASSERT(pcache->block == 0xffffffff);
184
185  ✓✗✓✓    858         if (off % lfs->cfg->prog_size == 0 &&
186        429                     size >= lfs->cfg->prog_size) {
187                             // bypass pcache?
188                             lfs_size_t diff = size - (size % lfs->cfg->prog_size);
```

```
189                 int err = lfs->cfg->prog(lfs->cfg, block, off, data, diff);
190                 if (err) {
191                     return err;
192                 }
193
194                 if (rcache) {
195                     int res = lfs_cache_cmp(lfs, rcache, NULL,
196                             block, off, data, diff);
197                     if (res < 0) {
198                         return res;
199                     }
200
201                     if (!res) {
202                         return LFS_ERR_CORRUPT;
203                     }
204                 }
205
206                 data += diff;
207                 off += diff;
208                 size -= diff;
209                 continue;
210             }
211
212             // prepare pcache, first condition can no longer fail
213 429         pcache->block = block;
214 429         pcache->off = off - (off % lfs->cfg->prog_size);
215         }
216
217 1561    return 0;
218     }
219
220
221     /// General lfs block device operations ///
222 5001    static int lfs_bd_read(lfs_t *lfs, lfs_block_t block,
223             lfs_off_t off, void *buffer, lfs_size_t size) {
224         // if we ever do more than writes to alternating pairs,
225         // this may need to consider pcache
226 5001    return lfs_cache_read(lfs, &lfs->rcache, NULL,
227                 block, off, buffer, size);
228     }
```

```
229
230  1355  static int lfs_bd_prog(lfs_t *lfs, lfs_block_t block,
231              lfs_off_t off, const void *buffer, lfs_size_t size) {
232  1355      return lfs_cache_prog(lfs, &lfs->pcache, NULL,
233                  block, off, buffer, size);
234        }
235
236   911  static int lfs_bd_cmp(lfs_t *lfs, lfs_block_t block,
237              lfs_off_t off, const void *buffer, lfs_size_t size) {
238   911      return lfs_cache_cmp(lfs, &lfs->rcache, NULL, block, off, buffer, size);
239        }
240
241  3140  static int lfs_bd_crc(lfs_t *lfs, lfs_block_t block,
242              lfs_off_t off, lfs_size_t size, uint32_t *crc) {
243  3140      return lfs_cache_crc(lfs, &lfs->rcache, NULL, block, off, size, crc);
244        }
245
246   429  static int lfs_bd_erase(lfs_t *lfs, lfs_block_t block) {
247   429      return lfs->cfg->erase(lfs->cfg, block);
248        }
249
250   326  static int lfs_bd_sync(lfs_t *lfs) {
251   326      lfs_cache_drop(lfs, &lfs->rcache);
252
253   326      int err = lfs_cache_flush(lfs, &lfs->pcache, NULL);
254 ✗✓ 326      if (err) {
255                  return err;
256            }
257
258   326      return lfs->cfg->sync(lfs->cfg);
259        }
260
261
262        /// Internal operations predeclared here ///
263        int lfs_traverse(lfs_t *lfs, int (*cb)(void*, lfs_block_t), void *data);
264        static int lfs_pred(lfs_t *lfs, const lfs_block_t dir[2], lfs_dir_t *pdir);
265        static int lfs_parent(lfs_t *lfs, const lfs_block_t dir[2],
266                lfs_dir_t *parent, lfs_entry_t *entry);
267        static int lfs_moved(lfs_t *lfs, const void *e);
268        static int lfs_relocate(lfs_t *lfs,
```

```
269          const lfs_block_t oldpair[2], const lfs_block_t newpair[2]);
270  int lfs_deorphan(lfs_t *lfs);
271
272
273  /// Block allocator ///
274  310 static int lfs_alloc_lookahead(void *p, lfs_block_t block) {
275  310     lfs_t *lfs = p;
276
277  620     lfs_block_t off = ((block - lfs->free.off)
278  310             + lfs->cfg->block_count) % lfs->cfg->block_count;
279
280  ✓✗ 310     if (off < lfs->free.size) {
281  310         lfs->free.buffer[off / 32] |= 1U << (off % 32);
282              }
283
284  310     return 0;
285  }
286
287  180 static int lfs_alloc(lfs_t *lfs, lfs_block_t *block) {
288   53     while (true) {
289  ✓✓ 670         while (lfs->free.i != lfs->free.size) {
290  437             lfs_block_t off = lfs->free.i;
291  437             lfs->free.i += 1;
292  437             lfs->free.ack -= 1;
293
294  ✓✓ 437             if (!(lfs->free.buffer[off / 32] & (1U << (off % 32)))) {
295                         // found a free block
296  127                     *block = (lfs->free.off + off) % lfs->cfg->block_count;
297
298                         // eagerly find next off so an alloc ack can
299                         // discredit old lookahead blocks
300  ✓✓✗✓ 330                 while (lfs->free.i != lfs->free.size &&
301   76                            (lfs->free.buffer[lfs->free.i / 32]
302   76                                & (1U << (lfs->free.i % 32)))) {
303                             lfs->free.i += 1;
304                             lfs->free.ack -= 1;
305                         }
306
307  127                     return 0;
308                  }
```

```
309                 }
310
311                 // check if we have looked at all blocks since last ack
312 ✗✓      53      if (lfs->free.ack == 0) {
313                     LFS_WARN("No more free space %" PRIu32,
314                             lfs->free.i + lfs->free.off);
315                     return LFS_ERR_NOSPC;
316                 }
317
318     106         lfs->free.off = (lfs->free.off + lfs->free.size)
319      53                 % lfs->cfg->block_count;
320      53         lfs->free.size = lfs_min(lfs->cfg->lookahead, lfs->free.ack);
321      53         lfs->free.i = 0;
322
323                 // find mask of free blocks from tree
324      53         memset(lfs->free.buffer, 0, lfs->cfg->lookahead/8);
325      53         int err = lfs_traverse(lfs, lfs_alloc_lookahead, lfs);
326 ✗✓      53      if (err) {
327                     return err;
328                 }
329             }
330         }
331
332     337     static void lfs_alloc_ack(lfs_t *lfs) {
333     337         lfs->free.ack = lfs->cfg->block_count;
334     337     }
335
336
337             /// Endian swapping functions ///
338    6568     static void lfs_dir_fromle32(struct lfs_disk_dir *d) {
339    6568         d->rev     = lfs_fromle32(d->rev);
340    6568         d->size    = lfs_fromle32(d->size);
341    6568         d->tail[0] = lfs_fromle32(d->tail[0]);
342    6568         d->tail[1] = lfs_fromle32(d->tail[1]);
343    6568     }
344
345    3140     static void lfs_dir_tole32(struct lfs_disk_dir *d) {
346    3140         d->rev     = lfs_tole32(d->rev);
347    3140         d->size    = lfs_tole32(d->size);
348    3140         d->tail[0] = lfs_tole32(d->tail[0]);
```

```
349        3140      d->tail[1] = lfs_tole32(d->tail[1]);
350        3140  }
351
352        1387  static void lfs_entry_fromle32(struct lfs_disk_entry *d) {
353        1387      d->u.dir[0] = lfs_fromle32(d->u.dir[0]);
354        1387      d->u.dir[1] = lfs_fromle32(d->u.dir[1]);
355        1387  }
356
357         210  static void lfs_entry_tole32(struct lfs_disk_entry *d) {
358         210      d->u.dir[0] = lfs_tole32(d->u.dir[0]);
359         210      d->u.dir[1] = lfs_tole32(d->u.dir[1]);
360         210  }
361
362           4  static void lfs_superblock_fromle32(struct lfs_disk_superblock *d) {
363           4      d->root[0]     = lfs_fromle32(d->root[0]);
364           4      d->root[1]     = lfs_fromle32(d->root[1]);
365           4      d->block_size  = lfs_fromle32(d->block_size);
366           4      d->block_count = lfs_fromle32(d->block_count);
367           4      d->version     = lfs_fromle32(d->version);
368           4  }
369
370           4  static void lfs_superblock_tole32(struct lfs_disk_superblock *d) {
371           4      d->root[0]     = lfs_tole32(d->root[0]);
372           4      d->root[1]     = lfs_tole32(d->root[1]);
373           4      d->block_size  = lfs_tole32(d->block_size);
374           4      d->block_count = lfs_tole32(d->block_count);
375           4      d->version     = lfs_tole32(d->version);
376           4  }
377
378
379              /// Metadata pair and directory operations ///
380         326  static inline void lfs_pairswap(lfs_block_t pair[2]) {
381         326      lfs_block_t t = pair[0];
382         326      pair[0] = pair[1];
383         326      pair[1] = t;
384         326  }
385
386         635  static inline bool lfs_pairisnull(const lfs_block_t pair[2]) {
387  ✓✓✗✓   635      return pair[0] == 0xffffffff || pair[1] == 0xffffffff;
388              }
```

```
389
390          204  static inline int lfs_paircmp(
391                       const lfs_block_t paira[2],
392                       const lfs_block_t pairb[2]) {
393  ✓✓✓✗    454      return !(paira[0] == pairb[0] || paira[1] == pairb[1] ||
394  ✓✓✓✗    250               paira[0] == pairb[1] || paira[1] == pairb[0]);
395                   }
396
397            4  static inline bool lfs_pairsync(
398                       const lfs_block_t paira[2],
399                       const lfs_block_t pairb[2]) {
400  ✓✗✗✓      8      return (paira[0] == pairb[0] && paira[1] == pairb[1]) ||
401  ✗✗                      (paira[0] == pairb[1] && paira[1] == pairb[0]);
402                   }
403
404         2304  static inline lfs_size_t lfs_entry_size(const lfs_entry_t *entry) {
405         2304      return 4 + entry->d.elen + entry->d.alen + entry->d.nlen;
406                   }
407
408           12  static int lfs_dir_alloc(lfs_t *lfs, lfs_dir_t *dir) {
409                   // allocate pair of dir blocks
410  ✓✓       36      for (int i = 0; i < 2; i++) {
411           24          int err = lfs_alloc(lfs, &dir->pair[i]);
412  ✗✓       24          if (err) {
413                           return err;
414                       }
415                   }
416
417                   // rather than clobbering one of the blocks we just pretend
418                   // the revision may be valid
419           12      int err = lfs_bd_read(lfs, dir->pair[0], 0, &dir->d.rev, 4);
420  ✗✓✗✗     12      if (err && err != LFS_ERR_CORRUPT) {
421                       return err;
422                   }
423
424  ✓✗       12      if (err != LFS_ERR_CORRUPT) {
425           12          dir->d.rev = lfs_fromle32(dir->d.rev);
426                   }
```

```
427
428                 // set defaults
429      12         dir->d.rev += 1;
430      12         dir->d.size = sizeof(dir->d)+4;
431      12         dir->d.tail[0] = 0xffffffff;
432      12         dir->d.tail[1] = 0xffffffff;
433      12         dir->off = sizeof(dir->d);
434
435                 // don't write out yet, let caller take care of that
436      12         return 0;
437              }
438
439    1714      static int lfs_dir_fetch(lfs_t *lfs,
440                     lfs_dir_t *dir, const lfs_block_t pair[2]) {
441                 // copy out pair, otherwise may be aliasing dir
442    1714         const lfs_block_t tpair[2] = {pair[0], pair[1]};
443    1714         bool valid = false;
444
445                 // check both blocks for the most recent revision
446  ✓✓ 5142         for (int i = 0; i < 2; i++) {
447                     struct lfs_disk_dir test;
448    3428             int err = lfs_bd_read(lfs, tpair[i], 0, &test, sizeof(test));
449    3428             lfs_dir_fromle32(&test);
450  ✗✓ 3428             if (err) {
451                         if (err == LFS_ERR_CORRUPT) {
452     614                     continue;
453                         }
454                         return err;
455                     }
456
457 ✓✓✓✓ 3428             if (valid && lfs_scmp(test.rev, dir->d.rev) < 0) {
458     598                 continue;
459                     }
460
461 ✓✓✗✓ 5644             if ((0x7fffffff & test.size) < sizeof(test)+4 ||
462    2814                 (0x7fffffff & test.size) > lfs->cfg->block_size) {
463      16                 continue;
464                     }
465
466    2814             uint32_t crc = 0xffffffff;
```

```
467                 lfs_dir_tole32(&test);
468      2814       lfs_crc(&crc, &test, sizeof(test));
469      2814       lfs_dir_fromle32(&test);
470      2814       err = lfs_bd_crc(lfs, tpair[i], sizeof(test),
471      2814               (0x7fffffff & test.size) - sizeof(test), &crc);
472   ✗✓ 2814       if (err) {
473                     if (err == LFS_ERR_CORRUPT) {
474                         continue;
475                     }
476                     return err;
477                 }
478
479   ✗✓ 2814       if (crc != 0) {
480                     continue;
481                 }
482
483      2814       valid = true;
484
485                 // setup dir in case it's valid
486      2814       dir->pair[0] = tpair[(i+0) % 2];
487      2814       dir->pair[1] = tpair[(i+1) % 2];
488      2814       dir->off = sizeof(dir->d);
489      2814       dir->d = test;
490             }
491
492   ✓✓ 1714       if (!valid) {
493         8           LFS_ERROR("Corrupted dir pair at %" PRIu32 " %" PRIu32 ,
494                             tpair[0], tpair[1]);
495         8           return LFS_ERR_CORRUPT;
496             }
497
498      1706       return 0;
499         }
500
501         struct lfs_region {
502             lfs_off_t oldoff;
503             lfs_size_t oldlen;
504             const void *newdata;
505             lfs_size_t newlen;
506         };
```

```
507                                
508          326  static int lfs_dir_commit(lfs_t *lfs, lfs_dir_t *dir,
509                           const struct lfs_region *regions, int count) {
510                       // increment revision count
511          326          dir->d.rev += 1;
512                                
513                       // keep pairs in order such that pair[0] is most recent
514          326          lfs_pairswap(dir->pair);
515      ✓✓  751          for (int i = 0; i < count; i++) {
516          425              dir->d.size += regions[i].newlen - regions[i].oldlen;
517                       }
518                                
519          326          const lfs_block_t oldpair[2] = {dir->pair[0], dir->pair[1]};
520          326          bool relocated = false;
521                                
522                       while (true) {
523                           if (true) {
524          326                  int err = lfs_bd_erase(lfs, dir->pair[0]);
525      ✗✓  326                  if (err) {
526                                   if (err == LFS_ERR_CORRUPT) {
527                                       goto relocate;
528                                   }
529                                   return err;
530                               }
531                                
532          326                  uint32_t crc = 0xffffffff;
533          326                  lfs_dir_tole32(&dir->d);
534          326                  lfs_crc(&crc, &dir->d, sizeof(dir->d));
535          326                  err = lfs_bd_prog(lfs, dir->pair[0], 0, &dir->d, sizeof(dir->d));
536          326                  lfs_dir_fromle32(&dir->d);
537      ✗✓  326                  if (err) {
538                                   if (err == LFS_ERR_CORRUPT) {
539                                       goto relocate;
540                                   }
541                                   return err;
542                               }
543                                
544          326                  int i = 0;
545          326                  lfs_off_t oldoff = sizeof(dir->d);
546          326                  lfs_off_t newoff = sizeof(dir->d);
```

```
547  ✓✓  1355              while (newoff < (0x7fffffff & dir->d.size)-4) {
548  ✓✓✓✓  703                  if (i < count && regions[i].oldoff == oldoff) {
549        325                      lfs_crc(&crc, regions[i].newdata, regions[i].newlen);
550        650                      err = lfs_bd_prog(lfs, dir->pair[0],
551        650                              newoff, regions[i].newdata, regions[i].newlen);
552  ✗✓    325                      if (err) {
553                                     if (err == LFS_ERR_CORRUPT) {
554                                         goto relocate;
555                                     }
556                                     return err;
557                                 }
558
559        325                      oldoff += regions[i].oldlen;
560        325                      newoff += regions[i].newlen;
561        325                      i += 1;
562                             } else {
563                                 uint8_t data;
564        378                      err = lfs_bd_read(lfs, oldpair[1], oldoff, &data, 1);
565  ✗✓    378                      if (err) {
566                                     return err;
567                                 }
568
569        378                      lfs_crc(&crc, &data, 1);
570        378                      err = lfs_bd_prog(lfs, dir->pair[0], newoff, &data, 1);
571  ✗✓    378                      if (err) {
572                                     if (err == LFS_ERR_CORRUPT) {
573                                         goto relocate;
574                                     }
575                                     return err;
576                                 }
577
578        378                      oldoff += 1;
579        378                      newoff += 1;
580                             }
581                         }
582
583        326                  crc = lfs_tole32(crc);
584        326                  err = lfs_bd_prog(lfs, dir->pair[0], newoff, &crc, 4);
585        326                  crc = lfs_fromle32(crc);
586  ✗✓    326                  if (err) {
```

```
587                     if (err == LFS_ERR_CORRUPT) {
588                         goto relocate;
589                     }
590                     return err;
591                 }
592
593     326         err = lfs_bd_sync(lfs);
594  ✗✓ 326         if (err) {
595                     if (err == LFS_ERR_CORRUPT) {
596                         goto relocate;
597                     }
598                     return err;
599                 }
600
601                 // successful commit, check checksum to make sure
602     326         uint32_t ncrc = 0xffffffff;
603     326         err = lfs_bd_crc(lfs, dir->pair[0], 0,
604     326                 (0x7fffffff & dir->d.size)-4, &ncrc);
605  ✗✓ 326         if (err) {
606                     return err;
607                 }
608
609  ✗✓ 326         if (ncrc != crc) {
610                     goto relocate;
611                 }
612             }
613
614     326     break;
615 relocate:
616             //commit was corrupted
617             LFS_DEBUG("Bad block at %" PRIu32, dir->pair[0]);
618
619             // drop caches and prepare to relocate block
620             relocated = true;
621             lfs_cache_drop(lfs, &lfs->pcache);
622
623             // can't relocate superblock, filesystem is now frozen
624             if (lfs_paircmp(oldpair, (const lfs_block_t[2]){0, 1}) == 0) {
625                 LFS_WARN("Superblock %" PRIu32 " has become unwritable",
626                         oldpair[0]);
```

```
627                 return LFS_ERR_CORRUPT;
628             }
629
630             // relocate half of pair
631             int err = lfs_alloc(lfs, &dir->pair[0]);
632             if (err) {
633                 return err;
634             }
635         }
636
637 ✗✓ 326      if (relocated) {
638             // update references if we relocated
639             LFS_DEBUG("Relocating %" PRIu32 " %" PRIu32 " to %" PRIu32 " %" PRIu32,
640                     oldpair[0], oldpair[1], dir->pair[0], dir->pair[1]);
641             int err = lfs_relocate(lfs, oldpair, dir->pair);
642             if (err) {
643                 return err;
644             }
645         }
646
647         // shift over any directories that are affected
648 ✗✓ 326      for (lfs_dir_t *d = lfs->dirs; d; d = d->next) {
649             if (lfs_paircmp(d->pair, dir->pair) == 0) {
650                 d->pair[0] = dir->pair[0];
651                 d->pair[1] = dir->pair[1];
652             }
653         }
654
655 326         return 0;
656     }
657
658 103 static int lfs_dir_update(lfs_t *lfs, lfs_dir_t *dir,
659             lfs_entry_t *entry, const void *data) {
660 103         lfs_entry_tole32(&entry->d);
661 ✗✓ 618      int err = lfs_dir_commit(lfs, dir, (struct lfs_region[]){
662 206             {entry->off, sizeof(entry->d), &entry->d, sizeof(entry->d)},
663 309             {entry->off+sizeof(entry->d), entry->d.nlen, data, entry->d.nlen}
664             }, data ? 2 : 1);
665 103         lfs_entry_fromle32(&entry->d);
666 103         return err;
```

```
667          }
668
669    107   static int lfs_dir_append(lfs_t *lfs, lfs_dir_t *dir,
670                  lfs_entry_t *entry, const void *data) {
671              // check if we fit, if top bit is set we do not and move on
672              while (true) {
673    ✓✗    107           if (dir->d.size + lfs_entry_size(entry) <= lfs->cfg->block_size) {
674    107               entry->off = dir->d.size - 4;
675
676    107               lfs_entry_tole32(&entry->d);
677    428               int err = lfs_dir_commit(lfs, dir, (struct lfs_region[]){
678    214                       {entry->off, 0, &entry->d, sizeof(entry->d)},
679    214                       {entry->off, 0, data, entry->d.nlen}
680                          }, 2);
681    107               lfs_entry_fromle32(&entry->d);
682    107               return err;
683                  }
684
685                  // we need to allocate a new dir block
686                  if (!(0x80000000 & dir->d.size)) {
687                      lfs_dir_t olddir = *dir;
688                      int err = lfs_dir_alloc(lfs, dir);
689                      if (err) {
690                          return err;
691                      }
692
693                      dir->d.tail[0] = olddir.d.tail[0];
694                      dir->d.tail[1] = olddir.d.tail[1];
695                      entry->off = dir->d.size - 4;
696                      lfs_entry_tole32(&entry->d);
697                      err = lfs_dir_commit(lfs, dir, (struct lfs_region[]){
698                              {entry->off, 0, &entry->d, sizeof(entry->d)},
699                              {entry->off, 0, data, entry->d.nlen}
700                          }, 2);
701                      lfs_entry_fromle32(&entry->d);
702                      if (err) {
703                          return err;
704                      }
705
706                      olddir.d.size |= 0x80000000;
```

```
707                 olddir.d.tail[0] = dir->pair[0];
708                 olddir.d.tail[1] = dir->pair[1];
709                 return lfs_dir_commit(lfs, &olddir, NULL, 0);
710             }
711
712             int err = lfs_dir_fetch(lfs, dir, dir->d.tail);
713             if (err) {
714                 return err;
715             }
716         }
717     }
718
719 100 static int lfs_dir_remove(lfs_t *lfs, lfs_dir_t *dir, lfs_entry_t *entry) {
720         // check if we should just drop the directory block
721 ✓✗ 200     if ((dir->d.size & 0x7fffffff) == sizeof(dir->d)+4
722 100             + lfs_entry_size(entry)) {
723             lfs_dir_t pdir;
724 100         int res = lfs_pred(lfs, dir->pair, &pdir);
725 ✗✓ 100         if (res < 0) {
726                 return res;
727             }
728
729 ✗✓ 100         if (pdir.d.size & 0x80000000) {
730                 pdir.d.size &= dir->d.size | 0x7fffffff;
731                 pdir.d.tail[0] = dir->d.tail[0];
732                 pdir.d.tail[1] = dir->d.tail[1];
733                 return lfs_dir_commit(lfs, &pdir, NULL, 0);
734             }
735         }
736
737         // shift out the entry
738 200     int err = lfs_dir_commit(lfs, dir, (struct lfs_region[]){
739 200             {entry->off, lfs_entry_size(entry), NULL, 0},
740             }, 1);
741 ✗✓ 100     if (err) {
742             return err;
743         }
744
745         // shift over any files/directories that are affected
746 ✗✓ 100     for (lfs_file_t *f = lfs->files; f; f = f->next) {
```

```
747                 if (lfs_paircmp(f->pair, dir->pair) == 0) {
748                     if (f->poff == entry->off) {
749                         f->pair[0] = 0xffffffff;
750                         f->pair[1] = 0xffffffff;
751                     } else if (f->poff > entry->off) {
752                         f->poff -= lfs_entry_size(entry);
753                     }
754                 }
755             }
756
✗✓  100     for (lfs_dir_t *d = lfs->dirs; d; d = d->next) {
758                 if (lfs_paircmp(d->pair, dir->pair) == 0) {
759                     if (d->off > entry->off) {
760                         d->off -= lfs_entry_size(entry);
761                         d->pos -= lfs_entry_size(entry);
762                     }
763                 }
764             }
765
766  100     return 0;
767         }
768
769  1146   static int lfs_dir_next(lfs_t *lfs, lfs_dir_t *dir, lfs_entry_t *entry) {
✓✓  2292       while (dir->off + sizeof(entry->d) > (0x7fffffff & dir->d.size)-4) {
✓✗   223            if (!(0x80000000 & dir->d.size)) {
772  223                entry->off = dir->off;
773  223                return LFS_ERR_NOENT;
774             }
775
776             int err = lfs_dir_fetch(lfs, dir, dir->d.tail);
777             if (err) {
778                 return err;
779             }
780
781             dir->off = sizeof(dir->d);
782             dir->pos += sizeof(dir->d) + 4;
783         }
784
785  923    int err = lfs_bd_read(lfs, dir->pair[0], dir->off,
786  923            &entry->d, sizeof(entry->d));
```

```
787        923      lfs_entry_fromle32(&entry->d);
788  ✗✓    923      if (err) {
789               return err;
790               }
791
792        923      entry->off = dir->off;
793        923      dir->off += lfs_entry_size(entry);
794        923      dir->pos += lfs_entry_size(entry);
795        923      return 0;
796               }
797
798        618   static int lfs_dir_find(lfs_t *lfs, lfs_dir_t *dir,
799                      lfs_entry_t *entry, const char **path) {
800        618      const char *pathname = *path;
801               size_t pathlen;
802        618      entry->d.type = LFS_TYPE_DIR;
803        618      entry->d.elen = sizeof(entry->d) - 4;
804        618      entry->d.alen = 0;
805        618      entry->d.nlen = 0;
806        618      entry->d.u.dir[0] = lfs->root[0];
807        618      entry->d.u.dir[1] = lfs->root[1];
808
809        911      while (true) {
810       1529   nextname:
811                   // skip slashes
812       1529         pathname += strspn(pathname, "/");
813       1529         pathlen = strcspn(pathname, "/");
814
815                   // skip '.' and root '..'
816  ✗✓✗✗ 1529         if ((pathlen == 1 && memcmp(pathname, ".", 1) == 0) ||
     ✗✓
817                       (pathlen == 2 && memcmp(pathname, "..", 2) == 0)) {
818                       pathname += pathlen;
819                       goto nextname;
820                   }
821
822                   // skip if matched by '..' in name
823       1529         const char *suffix = pathname + pathlen;
824                   size_t sufflen;
825       1529         int depth = 1;
```

```
826                 while (true) {
827 2543                suffix += strspn(suffix, "/");
828 2036                sufflen = strcspn(suffix, "/");
829 ✓✓ 2036              if (sufflen == 0) {
830 1529                    break;
831                     }
832
833 ✗✓✗✗ 507            if (sufflen == 2 && memcmp(suffix, "..", 2) == 0) {
834                         depth -= 1;
835                         if (depth == 0) {
836                             pathname = suffix + sufflen;
837                             goto nextname;
838                         }
839                     } else {
840 507                     depth += 1;
841                     }
842
843 507                 suffix += sufflen;
844                 }
845
846                 // found path
847 ✓✓ 1529          if (pathname[0] == '\0') {
848 404                 return 0;
849                 }
850
851                 // update what we've found
852 1125            *path = pathname;
853
854                 // continue on if we hit a directory
855 ✗✓ 1125          if (entry->d.type != LFS_TYPE_DIR) {
856                     return LFS_ERR_NOTDIR;
857                 }
858
859 1125            int err = lfs_dir_fetch(lfs, dir, entry->d.u.dir);
860 ✗✓ 1125          if (err) {
861                     return err;
862                 }
863
864                 // find entry matching name
865 2               while (true) {
```

```
866          1127             err = lfs_dir_next(lfs, dir, entry);
867    ✓✓    1127             if (err) {
868           214                 return err;
869                             }
870
871    ✓✓✓✗  1523             if (((0x7f & entry->d.type) != LFS_TYPE_REG &&
872    ✓✓    1523                  (0x7f & entry->d.type) != LFS_TYPE_DIR) ||
873           913               entry->d.nlen != pathlen) {
874             2                 continue;
875                             }
876
877          1822             int res = lfs_bd_cmp(lfs, dir->pair[0],
878           911                     entry->off + 4+entry->d.elen+entry->d.alen,
879                                   pathname, pathlen);
880    ✗✓    911              if (res < 0) {
881                                 return res;
882                             }
883
884                             // found match
885    ✓✗    911              if (res) {
886           911                 break;
887                             }
888                         }
889
890                         // check that entry has not been moved
891    ✓✗✓   911              if (!lfs->moving && entry->d.type & 0x80) {
892                                 int moved = lfs_moved(lfs, &entry->d.u);
893                                 if (moved < 0 || moved) {
894                                     return (moved < 0) ? moved : LFS_ERR_NOENT;
895                                 }
896
897                                 entry->d.type &= ~0x80;
898                             }
899
900                             // to next name
901           911               pathname += pathlen;
902                         }
903                     }
904
905
```

```
906                 /// Top level directory operations ///
907          4      int lfs_mkdir(lfs_t *lfs, const char *path) {
908                     // deorphan if we haven't yet, needed at most once after poweron
909      ✓✗  4          if (!lfs->deorphaned) {
910          4              int err = lfs_deorphan(lfs);
911      ✗✓  4              if (err) {
912                             return err;
913                         }
914                     }
915
916                     // fetch parent directory
917                     lfs_dir_t cwd;
918                     lfs_entry_t entry;
919          4          int err = lfs_dir_find(lfs, &cwd, &entry, &path);
920    ✓✗✗✓  4          if (err != LFS_ERR_NOENT || strchr(path, '/') != NULL) {
921                         return err ? err : LFS_ERR_EXIST;
922                     }
923
924                     // build up new directory
925          4          lfs_alloc_ack(lfs);
926
927                     lfs_dir_t dir;
928          4          err = lfs_dir_alloc(lfs, &dir);
929      ✗✓  4          if (err) {
930                         return err;
931                     }
932          4          dir.d.tail[0] = cwd.d.tail[0];
933          4          dir.d.tail[1] = cwd.d.tail[1];
934
935          4          err = lfs_dir_commit(lfs, &dir, NULL, 0);
936      ✗✓  4          if (err) {
937                         return err;
938                     }
939
940          4          entry.d.type = LFS_TYPE_DIR;
941          4          entry.d.elen = sizeof(entry.d) - 4;
942          4          entry.d.alen = 0;
943          4          entry.d.nlen = strlen(path);
944          4          entry.d.u.dir[0] = dir.pair[0];
945          4          entry.d.u.dir[1] = dir.pair[1];
```

```
946
947      4      cwd.d.tail[0] = dir.pair[0];
948      4      cwd.d.tail[1] = dir.pair[1];
949
950      4      err = lfs_dir_append(lfs, &cwd, &entry, path);
951   ✗✓ 4      if (err) {
952              return err;
953              }
954
955      4      lfs_alloc_ack(lfs);
956      4      return 0;
957            }
958
959    107   int lfs_dir_open(lfs_t *lfs, lfs_dir_t *dir, const char *path) {
960    107       dir->pair[0] = lfs->root[0];
961    107       dir->pair[1] = lfs->root[1];
962
963              lfs_entry_t entry;
964    107       int err = lfs_dir_find(lfs, dir, &entry, &path);
965   ✓✓ 107     if (err) {
966      4          return err;
967   ✗✓ 103     } else if (entry.d.type != LFS_TYPE_DIR) {
968              return LFS_ERR_NOTDIR;
969              }
970
971    103       err = lfs_dir_fetch(lfs, dir, entry.d.u.dir);
972   ✗✓ 103     if (err) {
973              return err;
974              }
975
976              // setup head dir
977              // special offset for '.' and '..'
978    103       dir->head[0] = dir->pair[0];
979    103       dir->head[1] = dir->pair[1];
980    103       dir->pos = sizeof(dir->d) - 2;
981    103       dir->off = sizeof(dir->d);
982
983              // add to list of directories
984    103       dir->next = lfs->dirs;
985    103       lfs->dirs = dir;
```

```
986
987  103      return 0;
988           }
989
990  103      int lfs_dir_close(lfs_t *lfs, lfs_dir_t *dir) {
991           // remove from list of directories
992  ✓✗ 103      for (lfs_dir_t **p = &lfs->dirs; *p; p = &(*p)->next) {
993  ✓✗ 103          if (*p == dir) {
994  103                 *p = dir->next;
995  103                 break;
996                  }
997           }
998
999  103      return 0;
1000           }
1001
1002 5        int lfs_dir_read(lfs_t *lfs, lfs_dir_t *dir, struct lfs_info *info) {
1003 5        memset(info, 0, sizeof(*info));
1004
1005          // special offset for '.' and '..'
1006 ✓✓ 5     if (dir->pos == sizeof(dir->d) - 2) {
1007 1           info->type = LFS_TYPE_DIR;
1008 1           strcpy(info->name, ".");
1009 1           dir->pos += 1;
1010 1           return 1;
1011 ✓✓ 4     } else if (dir->pos == sizeof(dir->d) - 1) {
1012 1           info->type = LFS_TYPE_DIR;
1013 1           strcpy(info->name, "..");
1014 1           dir->pos += 1;
1015 1           return 1;
1016          }
1017
1018          lfs_entry_t entry;
1019          while (true) {
1020 3           int err = lfs_dir_next(lfs, dir, &entry);
1021 ✓✓ 3        if (err) {
1022 ✗✓ 1            return (err == LFS_ERR_NOENT) ? 0 : err;
1023          }
1024
1025 ✗✓✗✗ 2      if ((0x7f & entry.d.type) != LFS_TYPE_REG &&
```

```
1026                 (0x7f & entry.d.type) != LFS_TYPE_DIR) {
1027                 continue;
1028             }
1029
1030             // check that entry has not been moved
1031 ✗✓      2   if (entry.d.type & 0x80) {
1032                 int moved = lfs_moved(lfs, &entry.d.u);
1033                 if (moved < 0) {
1034                     return moved;
1035                 }
1036
1037                 if (moved) {
1038                     continue;
1039                 }
1040
1041                 entry.d.type &= ~0x80;
1042             }
1043
1044         2   break;
1045         }
1046
1047         2   info->type = entry.d.type;
1048 ✓✗      2   if (info->type == LFS_TYPE_REG) {
1049         2       info->size = entry.d.u.file.size;
1050         }
1051
1052         4   int err = lfs_bd_read(lfs, dir->pair[0],
1053         2           entry.off + 4+entry.d.elen+entry.d.alen,
1054         4           info->name, entry.d.nlen);
1055 ✗✓      2   if (err) {
1056                 return err;
1057         }
1058
1059         2   return 1;
1060     }
1061
1062     int lfs_dir_seek(lfs_t *lfs, lfs_dir_t *dir, lfs_off_t off) {
1063         // simply walk from head dir
1064         int err = lfs_dir_rewind(lfs, dir);
1065         if (err) {
```

```
1066            return err;
1067        }
1068        dir->pos = off;
1069
1070        while (off > (0x7fffffff & dir->d.size)) {
1071            off -= 0x7fffffff & dir->d.size;
1072            if (!(0x80000000 & dir->d.size)) {
1073                return LFS_ERR_INVAL;
1074            }
1075
1076            err = lfs_dir_fetch(lfs, dir, dir->d.tail);
1077            if (err) {
1078                return err;
1079            }
1080        }
1081
1082        dir->off = off;
1083        return 0;
1084    }
1085
1086    lfs_soff_t lfs_dir_tell(lfs_t *lfs, lfs_dir_t *dir) {
1087        (void)lfs;
1088        return dir->pos;
1089    }
1090
1091    int lfs_dir_rewind(lfs_t *lfs, lfs_dir_t *dir) {
1092        // reload the head dir
1093        int err = lfs_dir_fetch(lfs, dir, dir->head);
1094        if (err) {
1095            return err;
1096        }
1097
1098        dir->pair[0] = dir->head[0];
1099        dir->pair[1] = dir->head[1];
1100        dir->pos = sizeof(dir->d) - 2;
1101        dir->off = sizeof(dir->d);
1102        return 0;
1103    }
1104
1105
```

```
1106          /// File index list operations ///
1107    604   static int lfs_ctz_index(lfs_t *lfs, lfs_off_t *off) {
1108    604       lfs_off_t size = *off;
1109    604       lfs_off_t b = lfs->cfg->block_size - 2*4;
1110    604       lfs_off_t i = size / b;
1111 ✓✗ 604       if (i == 0) {
1112    604           return 0;
1113              }
1114
1115              i = (size - 4*(lfs_popc(i-1)+2)) / b;
1116              *off = size - b*i - 4*lfs_popc(i);
1117              return i;
1118          }
1119
1120    302   static int lfs_ctz_find(lfs_t *lfs,
1121              lfs_cache_t *rcache, const lfs_cache_t *pcache,
1122              lfs_block_t head, lfs_size_t size,
1123              lfs_size_t pos, lfs_block_t *block, lfs_off_t *off) {
1124 ✗✓ 302       if (size == 0) {
1125              *block = 0xffffffff;
1126              *off = 0;
1127              return 0;
1128          }
1129
1130    302       lfs_off_t current = lfs_ctz_index(lfs, &(lfs_off_t){size-1});
1131    302       lfs_off_t target = lfs_ctz_index(lfs, &pos);
1132
1133 ✗✓ 604       while (current > target) {
1134              lfs_size_t skip = lfs_min(
1135                      lfs_npw2(current-target+1) - 1,
1136                      lfs_ctz(current));
1137
1138              int err = lfs_cache_read(lfs, rcache, pcache, head, 4*skip, &head, 4);
1139              head = lfs_fromle32(head);
1140              if (err) {
1141                  return err;
1142              }
1143
1144              LFS_ASSERT(head >= 2 && head <= lfs->cfg->block_count);
1145              current -= 1 << skip;
```

```
1146                 }
1147
1148     302         *block = head;
1149     302         *off = pos;
1150     302         return 0;
1151                 }
1152
1153     103         static int lfs_ctz_extend(lfs_t *lfs,
1154                         lfs_cache_t *rcache, lfs_cache_t *pcache,
1155                         lfs_block_t head, lfs_size_t size,
1156                         lfs_block_t *block, lfs_off_t *off) {
1157                     while (true) {
1158                         // go ahead and grab a block
1159                         lfs_block_t nblock;
1160     103                 int err = lfs_alloc(lfs, &nblock);
1161   ✗✓    103                 if (err) {
1162     103                     return err;
1163                         }
1164   ✓✗✓✗  103                 LFS_ASSERT(nblock >= 2 && nblock <= lfs->cfg->block_count);
1165
1166                         if (true) {
1167     103                     err = lfs_bd_erase(lfs, nblock);
1168   ✗✓    103                 if (err) {
1169                                 if (err == LFS_ERR_CORRUPT) {
1170                                     goto relocate;
1171                                 }
1172                                 return err;
1173                             }
1174
1175   ✓✗    103                 if (size == 0) {
1176     103                         *block = nblock;
1177     103                         *off = 0;
1178     103                         return 0;
1179                             }
1180
1181                             size -= 1;
1182                             lfs_off_t index = lfs_ctz_index(lfs, &size);
1183                             size += 1;
1184
1185                             // just copy out the last block if it is incomplete
```

```
1186                if (size != lfs->cfg->block_size) {
1187                    for (lfs_off_t i = 0; i < size; i++) {
1188                        uint8_t data;
1189                        err = lfs_cache_read(lfs, rcache, NULL,
1190                                head, i, &data, 1);
1191                        if (err) {
1192                            return err;
1193                        }
1194
1195                        err = lfs_cache_prog(lfs, pcache, rcache,
1196                                nblock, i, &data, 1);
1197                        if (err) {
1198                            if (err == LFS_ERR_CORRUPT) {
1199                                goto relocate;
1200                            }
1201                            return err;
1202                        }
1203                    }
1204
1205                    *block = nblock;
1206                    *off = size;
1207                    return 0;
1208                }
1209
1210                // append block
1211                index += 1;
1212                lfs_size_t skips = lfs_ctz(index) + 1;
1213
1214                for (lfs_off_t i = 0; i < skips; i++) {
1215                    head = lfs_tole32(head);
1216                    err = lfs_cache_prog(lfs, pcache, rcache,
1217                            nblock, 4*i, &head, 4);
1218                    head = lfs_fromle32(head);
1219                    if (err) {
1220                        if (err == LFS_ERR_CORRUPT) {
1221                            goto relocate;
1222                        }
1223                        return err;
1224                    }
1225
```

```
1226                    if (i != skips-1) {
1227                        err = lfs_cache_read(lfs, rcache, NULL,
1228                                head, 4*i, &head, 4);
1229                        head = lfs_fromle32(head);
1230                        if (err) {
1231                            return err;
1232                        }
1233                    }
1234
1235                    LFS_ASSERT(head >= 2 && head <= lfs->cfg->block_count);
1236                }
1237
1238                *block = nblock;
1239                *off = 4*skips;
1240                return 0;
1241            }
1242
1243    relocate:
1244            LFS_DEBUG("Bad block at %" PRIu32, nblock);
1245
1246            // just clear cache and try a new block
1247            lfs_cache_drop(lfs, &lfs->pcache);
1248        }
1249    }
1250
1251 49 static int lfs_ctz_traverse(lfs_t *lfs,
1252            lfs_cache_t *rcache, const lfs_cache_t *pcache,
1253            lfs_block_t head, lfs_size_t size,
1254            int (*cb)(void*, lfs_block_t), void *data) {
1255 ✓✗ 49     if (size == 0) {
1256 49         return 0;
1257        }
1258
1259        lfs_off_t index = lfs_ctz_index(lfs, &(lfs_off_t){size-1});
1260
1261        while (true) {
1262            int err = cb(data, head);
1263            if (err) {
1264                return err;
1265            }
```

```
1266
1267            if (index == 0) {
1268                return 0;
1269            }
1270
1271            lfs_block_t heads[2];
1272            int count = 2 - (index & 1);
1273            err = lfs_cache_read(lfs, rcache, pcache, head, 0, &heads, count*4);
1274            heads[0] = lfs_fromle32(heads[0]);
1275            heads[1] = lfs_fromle32(heads[1]);
1276            if (err) {
1277                return err;
1278            }
1279
1280            for (int i = 0; i < count-1; i++) {
1281                err = cb(data, heads[i]);
1282                if (err) {
1283                    return err;
1284                }
1285            }
1286
1287            head = heads[count-1];
1288            index -= count;
1289        }
1290    }
1291
1292
1293    /// Top level file operations ///
1294 407 int lfs_file_opencfg(lfs_t *lfs, lfs_file_t *file,
1295            const char *path, int flags,
1296            const struct lfs_file_config *cfg) {
1297        // deorphan if we haven't yet, needed at most once after poweron
1298 ✓✓✗✓ 407    if ((flags & 3) != LFS_O_RDONLY && !lfs->deorphaned) {
1299            int err = lfs_deorphan(lfs);
1300            if (err) {
1301                return err;
1302            }
1303        }
1304
1305        // allocate entry for file if it doesn't exist
```

```
1306                 lfs_dir_t cwd;
1307                 lfs_entry_t entry;
1308        407      int err = lfs_dir_find(lfs, &cwd, &entry, &path);
1309 ✓✓✓✗    407      if (err && (err != LFS_ERR_NOENT || strchr(path, '/') != NULL)) {
       ✗✓
1310                     return err;
1311                 }
1312
1313    ✓✓   407      if (err == LFS_ERR_NOENT) {
1314    ✓✓   206          if (!(flags & LFS_O_CREAT)) {
1315         103              return LFS_ERR_NOENT;
1316                     }
1317
1318                     // create entry to remember name
1319         103         entry.d.type = LFS_TYPE_REG;
1320         103         entry.d.elen = sizeof(entry.d) - 4;
1321         103         entry.d.alen = 0;
1322         103         entry.d.nlen = strlen(path);
1323         103         entry.d.u.file.head = 0xffffffff;
1324         103         entry.d.u.file.size = 0;
1325         103         err = lfs_dir_append(lfs, &cwd, &entry, path);
1326    ✗✓   103         if (err) {
1327                         return err;
1328                     }
1329    ✗✓   201     } else if (entry.d.type == LFS_TYPE_DIR) {
1330                     return LFS_ERR_ISDIR;
1331    ✗✓   201     } else if (flags & LFS_O_EXCL) {
1332                     return LFS_ERR_EXIST;
1333                 }
1334
1335                 // setup file struct
1336         304     file->cfg = cfg;
1337         304     file->pair[0] = cwd.pair[0];
1338         304     file->pair[1] = cwd.pair[1];
1339         304     file->poff = entry.off;
1340         304     file->head = entry.d.u.file.head;
1341         304     file->size = entry.d.u.file.size;
1342         304     file->flags = flags;
1343         304     file->pos = 0;
1344
```

```
1345  ✓✓  304      if (flags & LFS_O_TRUNC) {
1346  ✗✓  103          if (file->size != 0) {
1347                       file->flags |= LFS_F_DIRTY;
1348                   }
1349      103          file->head = 0xffffffff;
1350      103          file->size = 0;
1351               }
1352
1353               // allocate buffer if needed
1354      304      file->cache.block = 0xffffffff;
1355  ✗✓✗ 304      if (file->cfg && file->cfg->buffer) {
1356                   file->cache.buffer = file->cfg->buffer;
1357  ✗✓  304      } else if (lfs->cfg->file_buffer) {
1358                   if (lfs->files) {
1359                       // already in use
1360                       return LFS_ERR_NOMEM;
1361                   }
1362                   file->cache.buffer = lfs->cfg->file_buffer;
1363  ✓✓  304      } else if ((file->flags & 3) == LFS_O_RDONLY) {
1364      201          file->cache.buffer = lfs_malloc(lfs->cfg->read_size);
1365  ✗✓  201          if (!file->cache.buffer) {
1366                       return LFS_ERR_NOMEM;
1367                   }
1368               } else {
1369      103          file->cache.buffer = lfs_malloc(lfs->cfg->prog_size);
1370  ✗✓  103          if (!file->cache.buffer) {
1371                       return LFS_ERR_NOMEM;
1372                   }
1373               }
1374
1375               // zero to avoid information leak
1376      304      lfs_cache_drop(lfs, &file->cache);
1377  ✓✓  304      if ((file->flags & 3) != LFS_O_RDONLY) {
1378      103          lfs_cache_zero(lfs, &file->cache);
1379               }
1380
1381               // add to list of files
1382      304      file->next = lfs->files;
1383      304      lfs->files = file;
1384
```

```
1385          304      return 0;
1386                   }
1387
1388          407  int lfs_file_open(lfs_t *lfs, lfs_file_t *file,
1389                           const char *path, int flags) {
1390          407      return lfs_file_opencfg(lfs, file, path, flags, NULL);
1391                   }
1392
1393          304  int lfs_file_close(lfs_t *lfs, lfs_file_t *file) {
1394          304      int err = lfs_file_sync(lfs, file);
1395
1396                   // remove from list of files
1397    ✓✗    304      for (lfs_file_t **p = &lfs->files; *p; p = &(*p)->next) {
1398    ✓✗    304          if (*p == file) {
1399          304              *p = file->next;
1400          304              break;
1401                       }
1402                   }
1403
1404                   // clean up memory
1405 ✗✓✗✗ ✓✗   304      if (!(file->cfg && file->cfg->buffer) && !lfs->cfg->file_buffer) {
1406          304          lfs_free(file->cache.buffer);
1407                   }
1408
1409          304      return err;
1410                   }
1411
1412                   static int lfs_file_relocate(lfs_t *lfs, lfs_file_t *file) {
1413                   relocate:
1414                       LFS_DEBUG("Bad block at %" PRIu32, file->block);
1415
1416                       // just relocate what exists into new block
1417                       lfs_block_t nblock;
1418                       int err = lfs_alloc(lfs, &nblock);
1419                       if (err) {
1420                           return err;
1421                       }
1422
1423                       err = lfs_bd_erase(lfs, nblock);
```

```
1424                     if (err) {
1425                         if (err == LFS_ERR_CORRUPT) {
1426                             goto relocate;
1427                         }
1428                         return err;
1429                     }
1430 
1431                     // either read from dirty cache or disk
1432                     for (lfs_off_t i = 0; i < file->off; i++) {
1433                         uint8_t data;
1434                         err = lfs_cache_read(lfs, &lfs->rcache, &file->cache,
1435                                 file->block, i, &data, 1);
1436                         if (err) {
1437                             return err;
1438                         }
1439 
1440                         err = lfs_cache_prog(lfs, &lfs->pcache, &lfs->rcache,
1441                                 nblock, i, &data, 1);
1442                         if (err) {
1443                             if (err == LFS_ERR_CORRUPT) {
1444                                 goto relocate;
1445                             }
1446                             return err;
1447                         }
1448                     }
1449 
1450                     // copy over new state of file
1451                     memcpy(file->cache.buffer, lfs->pcache.buffer, lfs->cfg->prog_size);
1452                     file->cache.block = lfs->pcache.block;
1453                     file->cache.off = lfs->pcache.off;
1454                     lfs_cache_zero(lfs, &lfs->pcache);
1455 
1456                     file->block = nblock;
1457                     return 0;
1458                 }
1459 
1460     405     static int lfs_file_flush(lfs_t *lfs, lfs_file_t *file) {
1461 ✓✓  405         if (file->flags & LFS_F_READING) {
1462                         // just drop read cache
1463     302         lfs_cache_drop(lfs, &file->cache);
```

```
1464   302          file->flags &= ~LFS_F_READING;
1465              }
1466
1467  ✓✓  405      if (file->flags & LFS_F_WRITING) {
1468   103          lfs_off_t pos = file->pos;
1469
1470               // copy over anything after current branch
1471   412          lfs_file_t orig = {
1472   103              .head = file->head,
1473   103              .size = file->size,
1474                    .flags = LFS_O_RDONLY,
1475   103              .pos = file->pos,
1476                    .cache = lfs->rcache,
1477               };
1478   103          lfs_cache_drop(lfs, &lfs->rcache);
1479
1480  ✗✓  103          while (file->pos < file->size) {
1481                       // copy over a byte at a time, leave it up to caching
1482                       // to make this efficient
1483                       uint8_t data;
1484                       lfs_ssize_t res = lfs_file_read(lfs, &orig, &data, 1);
1485                       if (res < 0) {
1486                           return res;
1487                       }
1488
1489                       res = lfs_file_write(lfs, file, &data, 1);
1490                       if (res < 0) {
1491                           return res;
1492                       }
1493
1494                       // keep our reference to the rcache in sync
1495                       if (lfs->rcache.block != 0xffffffff) {
1496                           lfs_cache_drop(lfs, &orig.cache);
1497                           lfs_cache_drop(lfs, &lfs->rcache);
1498                       }
1499               }
1500
1501               // write out what we have
1502               while (true) {
1503   103             int err = lfs_cache_flush(lfs, &file->cache, &lfs->rcache);
```

```
1504  ✗✓  103              if (err) {
1505                           if (err == LFS_ERR_CORRUPT) {
1506                               goto relocate;
1507                           }
1508                           return err;
1509                       }
1510
1511      103              break;
1512                   relocate:
1513                       err = lfs_file_relocate(lfs, file);
1514                       if (err) {
1515                           return err;
1516                       }
1517                   }
1518
1519                   // actual file updates
1520      103          file->head = file->block;
1521      103          file->size = file->pos;
1522      103          file->flags &= ~LFS_F_WRITING;
1523      103          file->flags |= LFS_F_DIRTY;
1524
1525      103          file->pos = pos;
1526               }
1527
1528      405      return 0;
1529           }
1530
1531      304  int lfs_file_sync(lfs_t *lfs, lfs_file_t *file) {
1532      304      int err = lfs_file_flush(lfs, file);
1533  ✗✓  304      if (err) {
1534                   return err;
1535               }
1536
1537  ✓✓✓✗ 407      if ((file->flags & LFS_F_DIRTY) &&
1538   ✓✗  206              !(file->flags & LFS_F_ERRED) &&
1539      103              !lfs_pairisnull(file->pair)) {
1540                   // update dir entry
1541                   lfs_dir_t cwd;
1542      103          err = lfs_dir_fetch(lfs, &cwd, file->pair);
1543  ✗✓  103          if (err) {
```

```
1544                 return err;
1545             }
1546
1547     103     lfs_entry_t entry = {.off = file->poff};
1548     103     err = lfs_bd_read(lfs, cwd.pair[0], entry.off,
1549                     &entry.d, sizeof(entry.d));
1550     103     lfs_entry_fromle32(&entry.d);
1551 ✗✓  103     if (err) {
1552                 return err;
1553             }
1554
1555 ✗✓  103     LFS_ASSERT(entry.d.type == LFS_TYPE_REG);
1556     103     entry.d.u.file.head = file->head;
1557     103     entry.d.u.file.size = file->size;
1558
1559     103     err = lfs_dir_update(lfs, &cwd, &entry, NULL);
1560 ✗✓  103     if (err) {
1561                 return err;
1562             }
1563
1564     103     file->flags &= ~LFS_F_DIRTY;
1565         }
1566
1567     304     return 0;
1568     }
1569
1570     302 lfs_ssize_t lfs_file_read(lfs_t *lfs, lfs_file_t *file,
1571             void *buffer, lfs_size_t size) {
1572     302     uint8_t *data = buffer;
1573     302     lfs_size_t nsize = size;
1574
1575 ✗✓  302     if ((file->flags & 3) == LFS_O_WRONLY) {
1576                 return LFS_ERR_BADF;
1577             }
1578
1579 ✗✓  302     if (file->flags & LFS_F_WRITING) {
1580                 // flush out any writes
1581                 int err = lfs_file_flush(lfs, file);
1582                 if (err) {
1583                     return err;
```

```
1584                }
1585            }
1586
1587 ✗✓  302    if (file->pos >= file->size) {
1588                // eof if past end
1589                return 0;
1590            }
1591
1592        302 size = lfs_min(size, file->size - file->pos);
1593        302 nsize = size;
1594
1595 ✓✓  906    while (nsize > 0) {
1596                // check if we need a new block
1597 ✗✓✗✗ 302        if (!(file->flags & LFS_F_READING) ||
1598                        file->off == lfs->cfg->block_size) {
1599        302         int err = lfs_ctz_find(lfs, &file->cache, NULL,
1600                                file->head, file->size,
1601                                file->pos, &file->block, &file->off);
1602 ✗✓  302        if (err) {
1603                    return err;
1604                }
1605
1606        302         file->flags |= LFS_F_READING;
1607                }
1608
1609                // read as much as we can in current block
1610        302     lfs_size_t diff = lfs_min(nsize, lfs->cfg->block_size - file->off);
1611        302     int err = lfs_cache_read(lfs, &file->cache, NULL,
1612                        file->block, file->off, data, diff);
1613 ✗✓  302        if (err) {
1614                    return err;
1615                }
1616
1617        302         file->pos += diff;
1618        302         file->off += diff;
1619        302         data += diff;
1620        302         nsize -= diff;
1621            }
1622
1623        302 return size;
```

```
1624                  }
1625
1626          206     lfs_ssize_t lfs_file_write(lfs_t *lfs, lfs_file_t *file,
1627                          const void *buffer, lfs_size_t size) {
1628          206         const uint8_t *data = buffer;
1629          206         lfs_size_t nsize = size;
1630
1631      ✗✓  206         if ((file->flags & 3) == LFS_O_RDONLY) {
1632                          return LFS_ERR_BADF;
1633                      }
1634
1635      ✗✓  206         if (file->flags & LFS_F_READING) {
1636                          // drop any reads
1637                          int err = lfs_file_flush(lfs, file);
1638                          if (err) {
1639                              return err;
1640                          }
1641                      }
1642
1643  ✗✓✗✗  206         if ((file->flags & LFS_O_APPEND) && file->pos < file->size) {
1644                          file->pos = file->size;
1645                      }
1646
1647      ✗✓  206         if (file->pos + size > LFS_FILE_MAX) {
1648                          // larger than file limit?
1649                          return LFS_ERR_FBIG;
1650                      }
1651
1652  ✓✓✗✓  206         if (!(file->flags & LFS_F_WRITING) && file->pos > file->size) {
1653                          // fill with zeros
1654                          lfs_off_t pos = file->pos;
1655                          file->pos = file->size;
1656
1657                          while (file->pos < pos) {
1658                              lfs_ssize_t res = lfs_file_write(lfs, file, &(uint8_t){0}, 1);
1659                              if (res < 0) {
1660                                  return res;
1661                              }
1662                          }
1663                      }
```

```
1664                                
1665      ✓✓        618      while (nsize > 0) {
1666                                     // check if we need a new block
1667    ✓✓✗✓        309          if (!(file->flags & LFS_F_WRITING) ||
1668                103                  file->off == lfs->cfg->block_size) {
1669    ✓✗✗✓        103              if (!(file->flags & LFS_F_WRITING) && file->pos > 0) {
1670                                         // find out which block we're extending from
1671                                         int err = lfs_ctz_find(lfs, &file->cache, NULL,
1672                                                 file->head, file->size,
1673                                                 file->pos-1, &file->block, &file->off);
1674                                         if (err) {
1675                                             file->flags |= LFS_F_ERRED;
1676                                             return err;
1677                                         }
1678                                
1679                                         // mark cache as dirty since we may have read data into it
1680                                         lfs_cache_zero(lfs, &file->cache);
1681                                     }
1682                                
1683                                     // extend file with new blocks
1684                103                  lfs_alloc_ack(lfs);
1685                103                  int err = lfs_ctz_extend(lfs, &lfs->rcache, &file->cache,
1686                                                 file->block, file->pos,
1687                                                 &file->block, &file->off);
1688      ✗✓        103                  if (err) {
1689                                         file->flags |= LFS_F_ERRED;
1690                                         return err;
1691                                     }
1692                                
1693                103                  file->flags |= LFS_F_WRITING;
1694                                 }
1695                                
1696                                 // program as much as we can in current block
1697                206                  lfs_size_t diff = lfs_min(nsize, lfs->cfg->block_size - file->off);
1698                                     while (true) {
1699                206                      int err = lfs_cache_prog(lfs, &file->cache, &lfs->rcache,
1700                                                 file->block, file->off, data, diff);
1701      ✗✓        206                      if (err) {
1702                                             if (err == LFS_ERR_CORRUPT) {
1703                                                 goto relocate;
```

```
1704                     }
1705                     file->flags |= LFS_F_ERRED;
1706                     return err;
1707                 }
1708
1709     206         break;
1710             relocate:
1711                 err = lfs_file_relocate(lfs, file);
1712                 if (err) {
1713                     file->flags |= LFS_F_ERRED;
1714                     return err;
1715                 }
1716             }
1717
1718     206     file->pos += diff;
1719     206     file->off += diff;
1720     206     data += diff;
1721     206     nsize -= diff;
1722
1723     206     lfs_alloc_ack(lfs);
1724         }
1725
1726     206 file->flags &= ~LFS_F_ERRED;
1727     206 return size;
1728         }
1729
1730     101 lfs_soff_t lfs_file_seek(lfs_t *lfs, lfs_file_t *file,
1731             lfs_soff_t off, int whence) {
1732         // write out everything beforehand, may be noop if rdonly
1733     101 int err = lfs_file_flush(lfs, file);
1734 ✗✓  101 if (err) {
1735             return err;
1736         }
1737
1738         // find new pos
1739     101 lfs_soff_t npos = file->pos;
1740 ✓✗  101 if (whence == LFS_SEEK_SET) {
1741     101     npos = off;
1742         } else if (whence == LFS_SEEK_CUR) {
1743             npos = file->pos + off;
```

```
1744                    } else if (whence == LFS_SEEK_END) {
1745                        npos = file->size + off;
1746                    }
1747
1748 ✗✓          101     if (npos < 0 || npos > LFS_FILE_MAX) {
1749                        // file position out of range
1750                        return LFS_ERR_INVAL;
1751                    }
1752
1753                    // update pos
1754             101    file->pos = npos;
1755             101    return npos;
1756                }
1757
1758                int lfs_file_truncate(lfs_t *lfs, lfs_file_t *file, lfs_off_t size) {
1759                    if ((file->flags & 3) == LFS_O_RDONLY) {
1760                        return LFS_ERR_BADF;
1761                    }
1762
1763                    lfs_off_t oldsize = lfs_file_size(lfs, file);
1764                    if (size < oldsize) {
1765                        // need to flush since directly changing metadata
1766                        int err = lfs_file_flush(lfs, file);
1767                        if (err) {
1768                            return err;
1769                        }
1770
1771                        // lookup new head in ctz skip list
1772                        err = lfs_ctz_find(lfs, &file->cache, NULL,
1773                                file->head, file->size,
1774                                size, &file->head, &(lfs_off_t){0});
1775                        if (err) {
1776                            return err;
1777                        }
1778
1779                        file->size = size;
1780                        file->flags |= LFS_F_DIRTY;
1781                    } else if (size > oldsize) {
1782                        lfs_off_t pos = file->pos;
1783
```

```
1784            // flush+seek if not already at end
1785            if (file->pos != oldsize) {
1786                int err = lfs_file_seek(lfs, file, 0, LFS_SEEK_END);
1787                if (err < 0) {
1788                    return err;
1789                }
1790            }
1791
1792            // fill with zeros
1793            while (file->pos < size) {
1794                lfs_ssize_t res = lfs_file_write(lfs, file, &(uint8_t){0}, 1);
1795                if (res < 0) {
1796                    return res;
1797                }
1798            }
1799
1800            // restore pos
1801            int err = lfs_file_seek(lfs, file, pos, LFS_SEEK_SET);
1802            if (err < 0) {
1803                return err;
1804            }
1805        }
1806
1807        return 0;
1808    }
1809
1810    lfs_soff_t lfs_file_tell(lfs_t *lfs, lfs_file_t *file) {
1811        (void)lfs;
1812        return file->pos;
1813    }
1814
1815    int lfs_file_rewind(lfs_t *lfs, lfs_file_t *file) {
1816        lfs_soff_t res = lfs_file_seek(lfs, file, 0, LFS_SEEK_SET);
1817        if (res < 0) {
1818            return res;
1819        }
1820
1821        return 0;
1822    }
1823
```

```
1824        101  lfs_soff_t lfs_file_size(lfs_t *lfs, lfs_file_t *file) {
1825             (void)lfs;
1826  ✗✓    101      if (file->flags & LFS_F_WRITING) {
1827                     return lfs_max(file->pos, file->size);
1828             } else {
1829        101          return file->size;
1830             }
1831         }
1832
1833
1834         /// General fs operations ///
1835         int lfs_stat(lfs_t *lfs, const char *path, struct lfs_info *info) {
1836             lfs_dir_t cwd;
1837             lfs_entry_t entry;
1838             int err = lfs_dir_find(lfs, &cwd, &entry, &path);
1839             if (err) {
1840                 return err;
1841             }
1842
1843             memset(info, 0, sizeof(*info));
1844             info->type = entry.d.type;
1845             if (info->type == LFS_TYPE_REG) {
1846                 info->size = entry.d.u.file.size;
1847             }
1848
1849             if (lfs_paircmp(entry.d.u.dir, lfs->root) == 0) {
1850                 strcpy(info->name, "/");
1851             } else {
1852                 err = lfs_bd_read(lfs, cwd.pair[0],
1853                         entry.off + 4+entry.d.elen+entry.d.alen,
1854                         info->name, entry.d.nlen);
1855                 if (err) {
1856                     return err;
1857                 }
1858             }
1859
1860             return 0;
1861         }
1862
1863        100  int lfs_remove(lfs_t *lfs, const char *path) {
```

```
1864                    // deorphan if we haven't yet, needed at most once after poweron
1865 ✗✓ 100            if (!lfs->deorphaned) {
1866                        int err = lfs_deorphan(lfs);
1867                        if (err) {
1868                            return err;
1869                        }
1870                    }
1871
1872                    lfs_dir_t cwd;
1873                    lfs_entry_t entry;
1874 100                int err = lfs_dir_find(lfs, &cwd, &entry, &path);
1875 ✗✓ 100             if (err) {
1876                        return err;
1877                    }
1878
1879                    lfs_dir_t dir;
1880 ✗✓ 100             if (entry.d.type == LFS_TYPE_DIR) {
1881                        // must be empty before removal, checking size
1882                        // without masking top bit checks for any case where
1883                        // dir is not empty
1884                        err = lfs_dir_fetch(lfs, &dir, entry.d.u.dir);
1885                        if (err) {
1886                            return err;
1887                        } else if (dir.d.size != sizeof(dir.d)+4) {
1888                            return LFS_ERR_NOTEMPTY;
1889                        }
1890                    }
1891
1892                    // remove the entry
1893 100                err = lfs_dir_remove(lfs, &cwd, &entry);
1894 ✗✓ 100             if (err) {
1895                        return err;
1896                    }
1897
1898                    // if we were a directory, find pred, replace tail
1899 ✗✓ 100             if (entry.d.type == LFS_TYPE_DIR) {
1900                        int res = lfs_pred(lfs, dir.pair, &cwd);
1901                        if (res < 0) {
1902                            return res;
1903                        }
```

```
1904
1905            LFS_ASSERT(res); // must have pred
1906            cwd.d.tail[0] = dir.d.tail[0];
1907            cwd.d.tail[1] = dir.d.tail[1];
1908
1909            err = lfs_dir_commit(lfs, &cwd, NULL, 0);
1910            if (err) {
1911                return err;
1912            }
1913        }
1914
1915  100   return 0;
1916    }
1917
1918    int lfs_rename(lfs_t *lfs, const char *oldpath, const char *newpath) {
1919        // deorphan if we haven't yet, needed at most once after poweron
1920        if (!lfs->deorphaned) {
1921            int err = lfs_deorphan(lfs);
1922            if (err) {
1923                return err;
1924            }
1925        }
1926
1927        // find old entry
1928        lfs_dir_t oldcwd;
1929        lfs_entry_t oldentry;
1930        int err = lfs_dir_find(lfs, &oldcwd, &oldentry, &(const char *){oldpath});
1931        if (err) {
1932            return err;
1933        }
1934
1935        // mark as moving
1936        oldentry.d.type |= 0x80;
1937        err = lfs_dir_update(lfs, &oldcwd, &oldentry, NULL);
1938        if (err) {
1939            return err;
1940        }
1941
1942        // allocate new entry
1943        lfs_dir_t newcwd;
```

```
1944        lfs_entry_t preventry;
1945        err = lfs_dir_find(lfs, &newcwd, &preventry, &newpath);
1946        if (err && (err != LFS_ERR_NOENT || strchr(newpath, '/') != NULL)) {
1947            return err;
1948        }
1949
1950        // must have same type
1951        bool prevexists = (err != LFS_ERR_NOENT);
1952        if (prevexists && preventry.d.type != (0x7f & oldentry.d.type)) {
1953            return LFS_ERR_ISDIR;
1954        }
1955
1956        lfs_dir_t dir;
1957        if (prevexists && preventry.d.type == LFS_TYPE_DIR) {
1958            // must be empty before removal, checking size
1959            // without masking top bit checks for any case where
1960            // dir is not empty
1961            err = lfs_dir_fetch(lfs, &dir, preventry.d.u.dir);
1962            if (err) {
1963                return err;
1964            } else if (dir.d.size != sizeof(dir.d)+4) {
1965                return LFS_ERR_NOTEMPTY;
1966            }
1967        }
1968
1969        // move to new location
1970        lfs_entry_t newentry = preventry;
1971        newentry.d = oldentry.d;
1972        newentry.d.type &= ~0x80;
1973        newentry.d.nlen = strlen(newpath);
1974
1975        if (prevexists) {
1976            err = lfs_dir_update(lfs, &newcwd, &newentry, newpath);
1977            if (err) {
1978                return err;
1979            }
1980        } else {
1981            err = lfs_dir_append(lfs, &newcwd, &newentry, newpath);
1982            if (err) {
1983                return err;
```

```
1984                }
1985            }
1986
1987            // fetch old pair again in case dir block changed
1988            lfs->moving = true;
1989            err = lfs_dir_find(lfs, &oldcwd, &oldentry, &oldpath);
1990            if (err) {
1991                return err;
1992            }
1993            lfs->moving = false;
1994
1995            // remove old entry
1996            err = lfs_dir_remove(lfs, &oldcwd, &oldentry);
1997            if (err) {
1998                return err;
1999            }
2000
2001            // if we were a directory, find pred, replace tail
2002            if (prevexists && preventry.d.type == LFS_TYPE_DIR) {
2003                int res = lfs_pred(lfs, dir.pair, &newcwd);
2004                if (res < 0) {
2005                    return res;
2006                }
2007
2008                LFS_ASSERT(res); // must have pred
2009                newcwd.d.tail[0] = dir.d.tail[0];
2010                newcwd.d.tail[1] = dir.d.tail[1];
2011
2012                err = lfs_dir_commit(lfs, &newcwd, NULL, 0);
2013                if (err) {
2014                    return err;
2015                }
2016            }
2017
2018            return 0;
2019        }
2020
2021
2022        /// Filesystem operations ///
2023    16  static void lfs_deinit(lfs_t *lfs) {
```

```
2024          |          |     // free allocated memory
2025    ✓✗    |       16 |     if (!lfs->cfg->read_buffer) {
2026          |       16 |         lfs_free(lfs->rcache.buffer);
2027          |          |     }
2028          |          |
2029    ✓✗    |       16 |     if (!lfs->cfg->prog_buffer) {
2030          |       16 |         lfs_free(lfs->pcache.buffer);
2031          |          |     }
2032          |          |
2033    ✓✗    |       16 |     if (!lfs->cfg->lookahead_buffer) {
2034          |       16 |         lfs_free(lfs->free.buffer);
2035          |          |     }
2036          |       16 | }
2037          |          |
2038          |       16 | static int lfs_init(lfs_t *lfs, const struct lfs_config *cfg) {
2039          |       16 |     lfs->cfg = cfg;
2040          |          |
2041          |          |     // setup read cache
2042    ✗✓    |       16 |     if (lfs->cfg->read_buffer) {
2043          |          |         lfs->rcache.buffer = lfs->cfg->read_buffer;
2044          |          |     } else {
2045          |       16 |         lfs->rcache.buffer = lfs_malloc(lfs->cfg->read_size);
2046    ✗✓    |       16 |         if (!lfs->rcache.buffer) {
2047          |          |             goto cleanup;
2048          |          |         }
2049          |          |     }
2050          |          |
2051          |          |     // setup program cache
2052    ✗✓    |       16 |     if (lfs->cfg->prog_buffer) {
2053          |          |         lfs->pcache.buffer = lfs->cfg->prog_buffer;
2054          |          |     } else {
2055          |       16 |         lfs->pcache.buffer = lfs_malloc(lfs->cfg->prog_size);
2056    ✗✓    |       16 |         if (!lfs->pcache.buffer) {
2057          |          |             goto cleanup;
2058          |          |         }
2059          |          |     }
2060          |          |
2061          |          |     // zero to avoid information leaks
2062          |       16 |     lfs_cache_zero(lfs, &lfs->pcache);
2063          |       16 |     lfs_cache_drop(lfs, &lfs->rcache);
```

```
2064                
2065                    // setup lookahead, round down to nearest 32-bits
2066 ✗✓         16     LFS_ASSERT(lfs->cfg->lookahead % 32 == 0);
2067 ✗✓         16     LFS_ASSERT(lfs->cfg->lookahead > 0);
2068 ✗✓         16     if (lfs->cfg->lookahead_buffer) {
2069                        lfs->free.buffer = lfs->cfg->lookahead_buffer;
2070                    } else {
2071         16             lfs->free.buffer = lfs_malloc(lfs->cfg->lookahead/8);
2072 ✗✓      16             if (!lfs->free.buffer) {
2073                            goto cleanup;
2074                        }
2075                    }
2076                
2077                    // check that program and read sizes are multiples of the block size
2078 ✗✓         16     LFS_ASSERT(lfs->cfg->prog_size % lfs->cfg->read_size == 0);
2079 ✗✓         16     LFS_ASSERT(lfs->cfg->block_size % lfs->cfg->prog_size == 0);
2080                
2081                    // check that the block size is large enough to fit ctz pointers
2082 ✗✓         16     LFS_ASSERT(4*lfs_npw2(0xffffffff / (lfs->cfg->block_size-2*4))
2083                            <= lfs->cfg->block_size);
2084                
2085                    // setup default state
2086         16         lfs->root[0] = 0xffffffff;
2087         16         lfs->root[1] = 0xffffffff;
2088         16         lfs->files = NULL;
2089         16         lfs->dirs = NULL;
2090         16         lfs->deorphaned = false;
2091         16         lfs->moving = false;
2092                
2093         16         return 0;
2094                
2095                cleanup:
2096                    lfs_deinit(lfs);
2097                    return LFS_ERR_NOMEM;
2098                }
2099                
2100          4     int lfs_format(lfs_t *lfs, const struct lfs_config *cfg) {
2101          4         int err = 0;
2102                    if (true) {
2103          4             err = lfs_init(lfs, cfg);
```

```
2104  ✗✓  4          if (err) {
2105              4          return err;
2106              4          }
2107              4
2108              4          // create free lookahead
2109              4          memset(lfs->free.buffer, 0, lfs->cfg->lookahead/8);
2110              4          lfs->free.off = 0;
2111              4          lfs->free.size = lfs_min(lfs->cfg->lookahead, lfs->cfg->block_count);
2112              4          lfs->free.i = 0;
2113              4          lfs_alloc_ack(lfs);
2114              4
2115              4          // create superblock dir
2116              4          lfs_dir_t superdir;
2117              4          err = lfs_dir_alloc(lfs, &superdir);
2118  ✗✓          4          if (err) {
2119              4          goto cleanup;
2120              4          }
2121              4
2122              4          // write root directory
2123              4          lfs_dir_t root;
2124              4          err = lfs_dir_alloc(lfs, &root);
2125  ✗✓          4          if (err) {
2126              4          goto cleanup;
2127              4          }
2128              4
2129              4          err = lfs_dir_commit(lfs, &root, NULL, 0);
2130  ✗✓          4          if (err) {
2131              4          goto cleanup;
2132              4          }
2133              4
2134              4          lfs->root[0] = root.pair[0];
2135              4          lfs->root[1] = root.pair[1];
2136              4
2137              4          // write superblocks
2138             20          lfs_superblock_t superblock = {
2139                             .off = sizeof(superdir.d),
2140                             .d.type = LFS_TYPE_SUPERBLOCK,
2141                             .d.elen = sizeof(superblock.d) - sizeof(superblock.d.magic) - 4,
2142                             .d.nlen = sizeof(superblock.d.magic),
2143                             .d.version = LFS_DISK_VERSION,
```

```
2144                .d.magic = {"littlefs"},
2145        4       .d.block_size  = lfs->cfg->block_size,
2146        4       .d.block_count = lfs->cfg->block_count,
2147        8       .d.root = {lfs->root[0], lfs->root[1]},
2148            };
2149        4   superdir.d.tail[0] = root.pair[0];
2150        4   superdir.d.tail[1] = root.pair[1];
2151        4   superdir.d.size = sizeof(superdir.d) + sizeof(superblock.d) + 4;
2152
2153            // write both pairs to be safe
2154        4   lfs_superblock_tole32(&superblock.d);
2155        4   bool valid = false;
2156    ✓✓ 12   for (int i = 0; i < 2; i++) {
2157        8       err = lfs_dir_commit(lfs, &superdir, (struct lfs_region[]){
2158                        {sizeof(superdir.d), sizeof(superblock.d),
2159                         &superblock.d, sizeof(superblock.d)}
2160                    }, 1);
2161 ✗✓✗✗    8       if (err && err != LFS_ERR_CORRUPT) {
2162                        goto cleanup;
2163                    }
2164
2165 ✓✓✓✗    8       valid = valid || !err;
2166            }
2167
2168    ✗✓      4   if (!valid) {
2169                    err = LFS_ERR_CORRUPT;
2170                    goto cleanup;
2171            }
2172
2173            // sanity check that fetch works
2174        4   err = lfs_dir_fetch(lfs, &superdir, (const lfs_block_t[2]){0, 1});
2175    ✗✓      4   if (err) {
2176                    goto cleanup;
2177            }
2178
2179        4   lfs_alloc_ack(lfs);
2180        }
2181
2182        4   cleanup:
2183        4       lfs_deinit(lfs);
```

```
2184          4      return err;
2185                 }
2186
2187         12  int lfs_mount(lfs_t *lfs, const struct lfs_config *cfg) {
2188         12      int err = 0;
2189                 if (true) {
2190         12          err = lfs_init(lfs, cfg);
2191    ✗✓   12          if (err) {
2192          4              return err;
2193                      }
2194
2195                      // setup free lookahead
2196         12          lfs->free.off = 0;
2197         12          lfs->free.size = 0;
2198         12          lfs->free.i = 0;
2199         12          lfs_alloc_ack(lfs);
2200
2201                      // load superblock
2202                      lfs_dir_t dir;
2203                      lfs_superblock_t superblock;
2204         12          err = lfs_dir_fetch(lfs, &dir, (const lfs_block_t[2]){0, 1});
2205  ✓✓✗✓   12          if (err && err != LFS_ERR_CORRUPT) {
2206          8              goto cleanup;
2207                      }
2208
2209    ✓✓   12          if (!err) {
2210          4              err = lfs_bd_read(lfs, dir.pair[0], sizeof(dir.d),
2211                                 &superblock.d, sizeof(superblock.d));
2212          4              lfs_superblock_fromle32(&superblock.d);
2213    ✗✓    4              if (err) {
2214                             goto cleanup;
2215                          }
2216
2217          4              lfs->root[0] = superblock.d.root[0];
2218          4              lfs->root[1] = superblock.d.root[1];
2219                      }
2220
2221  ✓✓✗✓   12          if (err || memcmp(superblock.d.magic, "littlefs", 8) != 0) {
2222          8              LFS_ERROR("Invalid superblock at %d %d", 0, 1);
2223          8              err = LFS_ERR_CORRUPT;
```

```
2224        8              goto cleanup;
2225                     }
2226
2227        4            uint16_t major_version = (0xffff & (superblock.d.version >> 16));
2228        4            uint16_t minor_version = (0xffff & (superblock.d.version >>  0));
2229 ✓✗✗✓   4            if ((major_version != LFS_DISK_VERSION_MAJOR ||
2230                          minor_version > LFS_DISK_VERSION_MINOR)) {
2231                         LFS_ERROR("Invalid version %d.%d", major_version, minor_version);
2232                         err = LFS_ERR_INVAL;
2233                         goto cleanup;
2234                     }
2235
2236        4            return 0;
2237                 }
2238
2239        8        cleanup:
2240
2241        8            lfs_deinit(lfs);
2242        8            return err;
2243                 }
2244
2245        4        int lfs_unmount(lfs_t *lfs) {
2246        4            lfs_deinit(lfs);
2247        4            return 0;
2248                 }
2249
2250
2251                 /// Littlefs specific operations ///
2252       53        int lfs_traverse(lfs_t *lfs, int (*cb)(void*, lfs_block_t), void *data) {
2253  ✗✓   53            if (lfs_pairisnull(lfs->root)) {
2254                         return 0;
2255                     }
2256
2257                     // iterate over metadata pairs
2258                     lfs_dir_t dir;
2259                     lfs_entry_t entry;
2260       53            lfs_block_t cwd[2] = {0, 1};
2261
2262      102            while (true) {
2263  ✓✓  465                for (int i = 0; i < 2; i++) {
```

```
2264          310             int err = cb(data, cwd[i]);
2265    ✗✓    310             if (err) {
2266          ---                 return err;
2267          ---             }
2268          ---         }
2269          ---
2270          155         int err = lfs_dir_fetch(lfs, &dir, cwd);
2271    ✗✓    155         if (err) {
2272          ---             return err;
2273          ---         }
2274          ---
2275          ---         // iterate over contents
2276    ✓✓    461         while (dir.off + sizeof(entry.d) <= (0x7fffffff & dir.d.size)-4) {
2277          151             err = lfs_bd_read(lfs, dir.pair[0], dir.off,
2278          ---                     &entry.d, sizeof(entry.d));
2279          151             lfs_entry_fromle32(&entry.d);
2280    ✗✓    151             if (err) {
2281          ---                 return err;
2282          ---             }
2283          ---
2284          151             dir.off += lfs_entry_size(&entry);
2285    ✓✓    151             if ((0x70 & entry.d.type) == (0x70 & LFS_TYPE_REG)) {
2286           49                 err = lfs_ctz_traverse(lfs, &lfs->rcache, NULL,
2287          ---                         entry.d.u.file.head, entry.d.u.file.size, cb, data);
2288    ✗✓     49                 if (err) {
2289          ---                     return err;
2290          ---                 }
2291          ---             }
2292          ---         }
2293          ---
2294          155         cwd[0] = dir.d.tail[0];
2295          155         cwd[1] = dir.d.tail[1];
2296          ---
2297    ✓✓    155         if (lfs_pairisnull(cwd)) {
2298           53             break;
2299          ---         }
2300          ---     }
2301          ---
2302          ---     // iterate over any open files
2303    ✓✓    102     for (lfs_file_t *f = lfs->files; f; f = f->next) {
```

```
2304 ✗✓  49         if (f->flags & LFS_F_DIRTY) {
2305                     int err = lfs_ctz_traverse(lfs, &lfs->rcache, &f->cache,
2306                             f->head, f->size, cb, data);
2307                     if (err) {
2308                         return err;
2309                     }
2310                 }
2311
2312 ✗✓  49         if (f->flags & LFS_F_WRITING) {
2313                     int err = lfs_ctz_traverse(lfs, &lfs->rcache, &f->cache,
2314                             f->block, f->pos, cb, data);
2315                     if (err) {
2316                         return err;
2317                     }
2318                 }
2319             }
2320
2321      53     return 0;
2322          }
2323
2324     100 static int lfs_pred(lfs_t *lfs, const lfs_block_t dir[2], lfs_dir_t *pdir) {
2325 ✗✓  100     if (lfs_pairisnull(lfs->root)) {
2326                 return 0;
2327             }
2328
2329             // iterate over all directory directory entries
2330     100     int err = lfs_dir_fetch(lfs, pdir, (const lfs_block_t[2]){0, 1});
2331 ✗✓  100     if (err) {
2332                 return err;
2333             }
2334
2335 ✓✗  300     while (!lfs_pairisnull(pdir->d.tail)) {
2336 ✓✓  200         if (lfs_paircmp(pdir->d.tail, dir) == 0) {
2337     100             return true;
2338             }
2339
2340     100         err = lfs_dir_fetch(lfs, pdir, pdir->d.tail);
2341 ✗✓  100         if (err) {
2342                     return err;
2343             }
```

```
2344                }
2345
2346                return false;
2347            }
2348
2349      4     static int lfs_parent(lfs_t *lfs, const lfs_block_t dir[2],
2350                    lfs_dir_t *parent, lfs_entry_t *entry) {
2351  ✗✓  4         if (lfs_pairisnull(lfs->root)) {
2352                    return 0;
2353                }
2354
2355      4         parent->d.tail[0] = 0;
2356      4         parent->d.tail[1] = 1;
2357
2358                // iterate over all directory directory entries
2359  ✓✗  8         while (!lfs_pairisnull(parent->d.tail)) {
2360      4             int err = lfs_dir_fetch(lfs, parent, parent->d.tail);
2361  ✗✓  4             if (err) {
2362                        return err;
2363                    }
2364
2365                    while (true) {
2366      4                 err = lfs_dir_next(lfs, parent, entry);
2367  ✗✓✗✗  4               if (err && err != LFS_ERR_NOENT) {
2368                            return err;
2369                        }
2370
2371  ✗✓  4                 if (err == LFS_ERR_NOENT) {
2372                            break;
2373                        }
2374
2375  ✓✗✓✗  8               if (((0x70 & entry->d.type) == (0x70 & LFS_TYPE_DIR)) &&
2376      4                     lfs_paircmp(entry->d.u.dir, dir) == 0) {
2377      4                 return true;
2378                    }
2379                }
2380            }
2381
2382                return false;
2383            }
```

```
2384
2385 static int lfs_moved(lfs_t *lfs, const void *e) {
2386     if (lfs_pairisnull(lfs->root)) {
2387         return 0;
2388     }
2389
2390     // skip superblock
2391     lfs_dir_t cwd;
2392     int err = lfs_dir_fetch(lfs, &cwd, (const lfs_block_t[2]){0, 1});
2393     if (err) {
2394         return err;
2395     }
2396
2397     // iterate over all directory directory entries
2398     lfs_entry_t entry;
2399     while (!lfs_pairisnull(cwd.d.tail)) {
2400         err = lfs_dir_fetch(lfs, &cwd, cwd.d.tail);
2401         if (err) {
2402             return err;
2403         }
2404
2405         while (true) {
2406             err = lfs_dir_next(lfs, &cwd, &entry);
2407             if (err && err != LFS_ERR_NOENT) {
2408                 return err;
2409             }
2410
2411             if (err == LFS_ERR_NOENT) {
2412                 break;
2413             }
2414
2415             if (!(0x80 & entry.d.type) &&
2416                  memcmp(&entry.d.u, e, sizeof(entry.d.u)) == 0) {
2417                 return true;
2418             }
2419         }
2420     }
2421
2422     return false;
2423 }
```

```
2424
2425 static int lfs_relocate(lfs_t *lfs,
2426         const lfs_block_t oldpair[2], const lfs_block_t newpair[2]) {
2427     // find parent
2428     lfs_dir_t parent;
2429     lfs_entry_t entry;
2430     int res = lfs_parent(lfs, oldpair, &parent, &entry);
2431     if (res < 0) {
2432         return res;
2433     }
2434
2435     if (res) {
2436         // update disk, this creates a desync
2437         entry.d.u.dir[0] = newpair[0];
2438         entry.d.u.dir[1] = newpair[1];
2439
2440         int err = lfs_dir_update(lfs, &parent, &entry, NULL);
2441         if (err) {
2442             return err;
2443         }
2444
2445         // update internal root
2446         if (lfs_paircmp(oldpair, lfs->root) == 0) {
2447             LFS_DEBUG("Relocating root %" PRIu32 " %" PRIu32,
2448                     newpair[0], newpair[1]);
2449             lfs->root[0] = newpair[0];
2450             lfs->root[1] = newpair[1];
2451         }
2452
2453         // clean up bad block, which should now be a desync
2454         return lfs_deorphan(lfs);
2455     }
2456
2457     // find pred
2458     res = lfs_pred(lfs, oldpair, &parent);
2459     if (res < 0) {
2460         return res;
2461     }
2462
2463     if (res) {
```

```
2464                 // just replace bad pair, no desync can occur
2465                 parent.d.tail[0] = newpair[0];
2466                 parent.d.tail[1] = newpair[1];
2467
2468                 return lfs_dir_commit(lfs, &parent, NULL, 0);
2469             }
2470
2471             // couldn't find dir, must be new
2472             return 0;
2473         }
2474
2475       4 int lfs_deorphan(lfs_t *lfs) {
2476       4     lfs->deorphaned = true;
2477
2478   ✗✓  4     if (lfs_pairisnull(lfs->root)) {
2479               return 0;
2480           }
2481
2482       4     lfs_dir_t pdir = {.d.size = 0x80000000};
2483       4     lfs_dir_t cwd = {.d.tail[0] = 0, .d.tail[1] = 1};
2484
2485           // iterate over all directory directory entries
2486   ✓✗ 24     for (lfs_size_t i = 0; i < lfs->cfg->block_count; i++) {
2487   ✓✓ 12         if (lfs_pairisnull(cwd.d.tail)) {
2488    8                return 0;
2489             }
2490
2491    8         int err = lfs_dir_fetch(lfs, &cwd, cwd.d.tail);
2492   ✗✓ 8         if (err) {
2493               return err;
2494             }
2495
2496             // check head blocks for orphans
2497   ✓✓ 8         if (!(0x80000000 & pdir.d.size)) {
2498                 // check if we have a parent
2499                 lfs_dir_t parent;
2500                 lfs_entry_t entry;
2501    4            int res = lfs_parent(lfs, pdir.d.tail, &parent, &entry);
2502   ✗✓ 4            if (res < 0) {
2503                     return res;
```

```
2504                }
2505
2506 ✗✓      4      if (!res) {
2507                    // we are an orphan
2508                    LFS_DEBUG("Found orphan %" PRIu32 " %" PRIu32,
2509                            pdir.d.tail[0], pdir.d.tail[1]);
2510
2511                    pdir.d.tail[0] = cwd.d.tail[0];
2512                    pdir.d.tail[1] = cwd.d.tail[1];
2513
2514                    err = lfs_dir_commit(lfs, &pdir, NULL, 0);
2515                    if (err) {
2516                        return err;
2517                    }
2518
2519                    return 0;
2520                }
2521
2522 ✗✓      4      if (!lfs_pairsync(entry.d.u.dir, pdir.d.tail)) {
2523                    // we have desynced
2524                    LFS_DEBUG("Found desync %" PRIu32 " %" PRIu32,
2525                            entry.d.u.dir[0], entry.d.u.dir[1]);
2526
2527                    pdir.d.tail[0] = entry.d.u.dir[0];
2528                    pdir.d.tail[1] = entry.d.u.dir[1];
2529
2530                    err = lfs_dir_commit(lfs, &pdir, NULL, 0);
2531                    if (err) {
2532                        return err;
2533                    }
2534
2535                    return 0;
2536                }
2537            }
2538
2539            // check entries for moves
2540            lfs_entry_t entry;
2541            while (true) {
2542     16         err = lfs_dir_next(lfs, &cwd, &entry);
2543 ✓✓✗✓    12         if (err && err != LFS_ERR_NOENT) {
```

```
2544                    return err;
2545                }
2546
2547 ✓✓ 12             if (err == LFS_ERR_NOENT) {
2548  8                     break;
2549                }
2550
2551                // found moved entry
2552 ✗✓  4             if (entry.d.type & 0x80) {
2553                    int moved = lfs_moved(lfs, &entry.d.u);
2554                    if (moved < 0) {
2555                        return moved;
2556                    }
2557
2558                    if (moved) {
2559                        LFS_DEBUG("Found move %" PRIu32 " %" PRIu32,
2560                                entry.d.u.dir[0], entry.d.u.dir[1]);
2561                        err = lfs_dir_remove(lfs, &cwd, &entry);
2562                        if (err) {
2563                            return err;
2564                        }
2565                    } else {
2566                        LFS_DEBUG("Found partial move %" PRIu32 " %" PRIu32,
2567                                entry.d.u.dir[0], entry.d.u.dir[1]);
2568                        entry.d.type &= ~0x80;
2569                        err = lfs_dir_update(lfs, &cwd, &entry, NULL);
2570                        if (err) {
2571                            return err;
2572                        }
2573                    }
2574                }
2575            }
2576
2577  8         memcpy(&pdir, &cwd, sizeof(pdir));
2578        }
2579
2580        // If we reached here, we have more directory pairs than blocks in the
2581        // filesystem... So something must be horribly wrong
2582        return LFS_ERR_CORRUPT;
2583    }
```