

1691c07ff4

...

src / sys / kern / vfs_syscalls.c



laffer1 Sync with freebsd

History

1 contributor

4368 lines (4034 sloc) 94 KB

...

```
1  /*-
2   * Copyright (c) 1989, 1993
3   *   The Regents of the University of California. All rights reserved.
4   * (c) UNIX System Laboratories, Inc.
5   * All or some portions of this file are derived from material licensed
6   * to the University of California by American Telephone and Telegraph
7   * Co. or Unix System Laboratories, Inc. and are reproduced herein with
8   * the permission of UNIX System Laboratories, Inc.
9   *
10  * Redistribution and use in source and binary forms, with or without
11  * modification, are permitted provided that the following conditions
12  * are met:
13  * 1. Redistributions of source code must retain the above copyright
14  *   notice, this list of conditions and the following disclaimer.
15  * 2. Redistributions in binary form must reproduce the above copyright
16  *   notice, this list of conditions and the following disclaimer in the
17  *   documentation and/or other materials provided with the distribution.
18  * 4. Neither the name of the University nor the names of its contributors
19  *   may be used to endorse or promote products derived from this software
20  *   without specific prior written permission.
21  *
22  * THIS SOFTWARE IS PROVIDED BY THE REGENTS AND CONTRIBUTORS ``AS IS'' AND
23  * ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
24  * IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
25  * ARE DISCLAIMED. IN NO EVENT SHALL THE REGENTS OR CONTRIBUTORS BE LIABLE
26  * FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
27  * DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
28  * OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
29  * HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
30  * LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
31  * OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
32  * SUCH DAMAGE.
33  *
34  *   @(#)vfs_syscalls.c      8.13 (Berkeley) 4/15/94
35  */
36
37 #include <sys/cdefs.h>
38 __FBSDID("$FreeBSD: stable/11/sys/kern/vfs_syscalls.c 338987 2018-09-27 18:54:41Z gordon $");
39
40 #include "opt_capsicum.h"
41 #include "opt_compat.h"
42 #include "opt_ktrace.h"
43
44 #include <sys/param.h>
45 #include <sys/systm.h>
46 #include <sys/bio.h>
47 #include <sys/buf.h>
48 #include <sys/capsicum.h>
49 #include <sys/disk.h>
50 #include <sys/sysent.h>
51 #include <sys/malloc.h>
52 #include <sys/mount.h>
53 #include <sys/mutex.h>
54 #include <sys/sysproto.h>
55 #include <sys/namei.h>
56 #include <sys/filedesc.h>
57 #include <sys/kernel.h>
58 #include <sys/fcntl.h>
59 #include <sys/file.h>
60 #include <sys/filio.h>
61 #include <sys/limits.h>
62 #include <sys/linker.h>
63 #include <sys/rwlock.h>
64 #include <sys/sdt.h>
65 #include <sys/stat.h>
66 #include <sys/sx.h>
67 #include <sys/unistd.h>
68 #include <sys/vnode.h>
69 #include <sys/priv.h>
70 #include <sys/proc.h>
71 #include <sys/dirent.h>
72 #include <sys/jail.h>
73 #include <sys/syscallsubr.h>
74 #include <sys/sysctl.h>
75 #ifdef KTRACE
76 #include <sys/ktrace.h>
77 #endif
78
```

```

79 #include <machine/stdarg.h>
80
81 #include <security/audit/audit.h>
82 #include <security/mac/mac_framework.h>
83
84 #include <vm/vm.h>
85 #include <vm/vm_object.h>
86 #include <vm/vm_page.h>
87 #include <vm/uma.h>
88
89 #include <ufs/ufs/quota.h>
90
91 MALLOC_DEFINE(M_FADVISE, "Fadvise", "posix_fadvise(2) information");
92
93 SDT_PROVIDER_DEFINE(vfs);
94 SDT_PROBE_DEFINE2(vfs, , stat, mode, "char *", "int");
95 SDT_PROBE_DEFINE2(vfs, , stat, reg, "char *", "int");
96
97 static int kern_chflagsat(struct thread *td, int fd, const char *path,
98     enum uio_seg pathseg, u_long flags, int atflag);
99 static int setfflags(struct thread *td, struct vnode *, u_long);
100 static int getutimes(const struct timeval *, enum uio_seg, struct timespec *);
101 static int getutimens(const struct timespec *, enum uio_seg,
102     struct timespec *, int *);
103 static int setutimes(struct thread *td, struct vnode *,
104     const struct timespec *, int, int);
105 static int vn_access(struct vnode *vp, int user_flags, struct ucred *cred,
106     struct thread *td);
107
108 /*
109  * Sync each mounted filesystem.
110  */
111 #ifndef _SYS_SYSPROTO_H_
112 struct sync_args {
113     int    dummy;
114 };
115 #endif
116 /* ARGSUSED */
117 int
118 sys_sync(struct thread *td, struct sync_args *uap)
119 {
120     struct mount *mp, *nmp;
121     int save;
122
123     mtx_lock(&mountlist_mtx);
124     for (mp = TAILQ_FIRST(&mountlist); mp != NULL; mp = nmp) {
125         if (vfs_busy(mp, MBF_NOWAIT | MBF_MNTLSTLOCK)) {
126             nmp = TAILQ_NEXT(mp, mnt_list);
127             continue;
128         }
129         if ((mp->mnt_flag & MNT_RDONLY) == 0 &&
130             vn_start_write(NULL, &mp, V_NOWAIT) == 0) {
131             save = curthread_pflags_set(TDP_SYNCIO);
132             vfs_msync(mp, MNT_NOWAIT);
133             VFS_SYNC(mp, MNT_NOWAIT);
134             curthread_pflags_restore(save);
135             vn_finished_write(mp);
136         }
137         mtx_lock(&mountlist_mtx);
138         nmp = TAILQ_NEXT(mp, mnt_list);
139         vfs_unbusy(mp);
140     }
141     mtx_unlock(&mountlist_mtx);
142     return (0);
143 }
144
145 /*
146  * Change filesystem quotas.
147  */
148 #ifndef _SYS_SYSPROTO_H_
149 struct quotactl_args {
150     char *path;
151     int cmd;
152     int uid;
153     caddr_t arg;
154 };
155 #endif
156 int
157 sys_quotactl(struct thread *td, struct quotactl_args *uap)
158 {
159     struct mount *mp;
160     struct nameidata nd;
161     int error;
162
163     AUDIT_ARG_CMD(uap->cmd);
164     AUDIT_ARG_UID(uap->uid);
165     if (!prison_allow(td->td_ucred, PR_ALLOW_QUOTAS))
166         return (EPERM);
167     NDINIT(&nd, LOOKUP, FOLLOW | LOCKLEAF | AUDITVNODE1, UIO_USERSPACE,
168         uap->path, td);
169     if ((error = namei(&nd)) != 0)
170         return (error);
171     NDFREE(&nd, NDF_ONLY_PNBUF);
172     mp = nd.ni_vp->v_mount;
173     vfs_ref(mp);
174     vput(nd.ni_vp);
175     error = vfs_busy(mp, 0);
176     vfs_rel(mp);

```

```

177     if (error != 0)
178         return (error);
179     error = VFS_QUOTACTL(mp, uap->cmd, uap->uid, uap->arg);
180
181     /*
182     * Since quota on operation typically needs to open quota
183     * file, the Q_QUOTAON handler needs to unbusy the mount point
184     * before calling into namei. Otherwise, unmount might be
185     * started between two vfs_busy() invocations (first is our,
186     * second is from mount point cross-walk code in lookup()),
187     * causing deadlock.
188     *
189     * Require that Q_QUOTAON handles the vfs_busy() reference on
190     * its own, always returning with ubusied mount point.
191     */
192     if ((uap->cmd >> SUBCMDSHIFT) != Q_QUOTAON &&
193         (uap->cmd >> SUBCMDSHIFT) != Q_QUOTAOFF)
194         vfs_unbusy(mp);
195     return (error);
196 }
197
198 /*
199 * Used by statfs conversion routines to scale the block size up if
200 * necessary so that all of the block counts are <= 'max_size'. Note
201 * that 'max_size' should be a bitmask, i.e. 2^n - 1 for some non-zero
202 * value of 'n'.
203 */
204 void
205 statfs_scale_blocks(struct statfs *sf, long max_size)
206 {
207     uint64_t count;
208     int shift;
209
210     KASSERT(powerof2(max_size + 1), ("%s: invalid max_size", __func__));
211
212     /*
213     * Attempt to scale the block counts to give a more accurate
214     * overview to userland of the ratio of free space to used
215     * space. To do this, find the largest block count and compute
216     * a divisor that lets it fit into a signed integer <= max_size.
217     */
218     if (sf->f_bavail < 0)
219         count = -sf->f_bavail;
220     else
221         count = sf->f_bavail;
222     count = MAX(sf->f_blocks, MAX(sf->f_bfree, count));
223     if (count <= max_size)
224         return;
225
226     count >>= flsl(max_size);
227     shift = 0;
228     while (count > 0) {
229         shift++;
230         count >>= 1;
231     }
232
233     sf->f_bsize <= shift;
234     sf->f_blocks >>= shift;
235     sf->f_bfree >>= shift;
236     sf->f_bavail >>= shift;
237 }
238
239 static int
240 kern_do_statfs(struct thread *td, struct mount *mp, struct statfs *buf)
241 {
242     struct statfs *sp;
243     int error;
244
245     if (mp == NULL)
246         return (EBADF);
247     error = vfs_busy(mp, 0);
248     vfs_rel(mp);
249     if (error != 0)
250         return (error);
251
252 #ifdef MAC
253     error = mac_mount_check_stat(td->td_ucred, mp);
254     if (error != 0)
255         goto out;
256 #endif
257
258     /*
259     * Set these in case the underlying filesystem fails to do so.
260     */
261     sp = &mp->mnt_stat;
262     sp->f_version = STATFS_VERSION;
263     sp->f_namemax = NAME_MAX;
264     sp->f_flags = mp->mnt_flag & MNT_VISFLAGMASK;
265     error = VFS_STATFS(mp, sp);
266     if (error != 0)
267         goto out;
268     *buf = *sp;
269     if (priv_check(td, PRIV_VFS_GENERATION)) {
270         buf->f_fsid.val[0] = buf->f_fsid.val[1] = 0;
271         prison_enforce_statfs(td->td_ucred, mp, buf);
272     }
273 out:
274     vfs_unbusy(mp);
275     return (error);
276 }

```

```

275
276 /*
277  * Get filesystem statistics.
278  */
279 #ifndef _SYS_SYSPROTO_H_
280 struct statfs_args {
281     char *path;
282     struct statfs *buf;
283 };
284 #endif
285 int
286 sys_statfs(struct thread *td, struct statfs_args *uap)
287 {
288     struct statfs *sfp;
289     int error;
290
291     sfp = malloc(sizeof(struct statfs), M_STATFS, M_WAITOK);
292     error = kern_statfs(td, uap->path, UIO_USERSPACE, sfp);
293     if (error == 0)
294         error = copyout(sfp, uap->buf, sizeof(struct statfs));
295     free(sfp, M_STATFS);
296     return (error);
297 }
298
299 int
300 kern_statfs(struct thread *td, char *path, enum uio_seg pathseg,
301             struct statfs *buf)
302 {
303     struct mount *mp;
304     struct nameidata nd;
305     int error;
306
307     NDINIT(&nd, LOOKUP, FOLLOW | LOCKSHARED | LOCKLEAF | AUDITVNODE1,
308           pathseg, path, td);
309     error = namei(&nd);
310     if (error != 0)
311         return (error);
312     mp = nd.ni_vp->v_mount;
313     vfs_ref(mp);
314     NDFREE(&nd, NDF_ONLY_PNBUF);
315     vput(nd.ni_vp);
316     return (kern_do_statfs(td, mp, buf));
317 }
318
319 /*
320  * Get filesystem statistics.
321  */
322 #ifndef _SYS_SYSPROTO_H_
323 struct fstatfs_args {
324     int fd;
325     struct statfs *buf;
326 };
327 #endif
328 int
329 sys_fstatfs(struct thread *td, struct fstatfs_args *uap)
330 {
331     struct statfs *sfp;
332     int error;
333
334     sfp = malloc(sizeof(struct statfs), M_STATFS, M_WAITOK);
335     error = kern_fstatfs(td, uap->fd, sfp);
336     if (error == 0)
337         error = copyout(sfp, uap->buf, sizeof(struct statfs));
338     free(sfp, M_STATFS);
339     return (error);
340 }
341
342 int
343 kern_fstatfs(struct thread *td, int fd, struct statfs *buf)
344 {
345     struct file *fp;
346     struct mount *mp;
347     struct vnode *vp;
348     cap_rights_t rights;
349     int error;
350
351     AUDIT_ARG_FD(fd);
352     error = getvnode(td, fd, cap_rights_init(&rights, CAP_FSTATFS), &fp);
353     if (error != 0)
354         return (error);
355     vp = fp->v_vnode;
356     vn_lock(vp, LK_SHARED | LK_RETRY);
357 #ifdef AUDIT
358     AUDIT_ARG_VNODE1(vp);
359 #endif
360     mp = vp->v_mount;
361     if (mp != NULL)
362         vfs_ref(mp);
363     VOP_UNLOCK(vp, 0);
364     fdrop(fp, td);
365     return (kern_do_statfs(td, mp, buf));
366 }
367
368 /*
369  * Get statistics on all filesystems.
370  */
371 #ifndef _SYS_SYSPROTO_H_
372 struct getfsstat_args {

```

```

373     struct statfs *buf;
374     long bufsize;
375     int mode;
376 };
377 #endif
378 int
379 sys_getfsstat(struct thread *td, struct getfsstat_args *uap)
380 {
381     size_t count;
382     int error;
383
384     if (uap->bufsize < 0 || uap->bufsize > SIZE_MAX)
385         return (EINVAL);
386     error = kern_getfsstat(td, &uap->buf, uap->bufsize, &count,
387         UIO_USERSPACE, uap->mode);
388     if (error == 0)
389         td->td_retval[0] = count;
390     return (error);
391 }
392
393 /*
394  * If (bufsize > 0 && bufseg == UIO_SYSSPACE)
395  *   The caller is responsible for freeing memory which will be allocated
396  *   in '*buf'.
397  */
398 int
399 kern_getfsstat(struct thread *td, struct statfs **buf, size_t bufsize,
400     size_t *countp, enum uio_seg bufseg, int mode)
401 {
402     struct mount *mp, *nmp;
403     struct statfs *sfsp, *sp, *sptmp, *tofree;
404     size_t count, maxcount;
405     int error;
406
407     switch (mode) {
408     case MNT_WAIT:
409     case MNT_NOWAIT:
410         break;
411     default:
412         return (EINVAL);
413     }
414 restart:
415     maxcount = bufsize / sizeof(struct statfs);
416     if (bufsize == 0) {
417         sfsp = NULL;
418         tofree = NULL;
419     } else if (bufseg == UIO_USERSPACE) {
420         sfsp = *buf;
421         tofree = NULL;
422     } else /* if (bufseg == UIO_SYSSPACE) */ {
423         count = 0;
424         mtx_lock(&mountlist_mtx);
425         TAILQ_FOREACH(mp, &mountlist, mnt_list) {
426             count++;
427         }
428         mtx_unlock(&mountlist_mtx);
429         if (maxcount > count)
430             maxcount = count;
431         tofree = sfsp = *buf = malloc(maxcount * sizeof(struct statfs),
432             M_STATFS, M_WAITOK);
433     }
434     count = 0;
435     mtx_lock(&mountlist_mtx);
436     for (mp = TAILQ_FIRST(&mountlist); mp != NULL; mp = nmp) {
437         if (prison_canseemount(td->td_ucred, mp) != 0) {
438             nmp = TAILQ_NEXT(mp, mnt_list);
439             continue;
440         }
441 #ifdef MAC
442         if (mac_mount_check_stat(td->td_ucred, mp) != 0) {
443             nmp = TAILQ_NEXT(mp, mnt_list);
444             continue;
445         }
446 #endif
447         if (mode == MNT_WAIT) {
448             if (vfs_busy(mp, MBF_MNTLSTLOCK) != 0) {
449                 /*
450                  * If vfs_busy() failed, and MBF_NOWAIT
451                  * wasn't passed, then the mp is gone.
452                  * Furthermore, because of MBF_MNTLSTLOCK,
453                  * the mountlist_mtx was dropped. We have
454                  * no other choice than to start over.
455                  */
456                 mtx_unlock(&mountlist_mtx);
457                 free(tofree, M_STATFS);
458                 goto restart;
459             }
460         } else {
461             if (vfs_busy(mp, MBF_NOWAIT | MBF_MNTLSTLOCK) != 0) {
462                 nmp = TAILQ_NEXT(mp, mnt_list);
463                 continue;
464             }
465         }
466         if (sfsp != NULL && count < maxcount) {
467             sp = &mp->mnt_stat;
468             /*
469              * Set these in case the underlying filesystem
470              * fails to do so.

```

```

471         */
472         sp->f_version = STATFS_VERSION;
473         sp->f_namemax = NAME_MAX;
474         sp->f_flags = mp->mnt_flag & MNT_VISFLAGMASK;
475         /*
476          * If MNT_NOWAIT is specified, do not refresh
477          * the fsstat cache.
478          */
479         if (mode != MNT_NOWAIT) {
480             error = VFS_STATFS(mp, sp);
481             if (error != 0) {
482                 mtx_lock(&mountlist_mtx);
483                 nmp = TAILQ_NEXT(mp, mnt_list);
484                 vfs_unbusy(mp);
485                 continue;
486             }
487         }
488         if (priv_check(td, PRIV_VFS_GENERATION)) {
489             sptmp = malloc(sizeof(struct statfs), M_STATFS,
490                 M_WAITOK);
491             *sptmp = *sp;
492             sptmp->f_fsid.val[0] = sptmp->f_fsid.val[1] = 0;
493             prison_enforce_statfs(td->td_ucred, mp, sptmp);
494             sp = sptmp;
495         } else
496             sptmp = NULL;
497         if (bufseg == UIO_SYSSPACE) {
498             bcopy(sp, sfsp, sizeof(*sp));
499             free(sptmp, M_STATFS);
500         } else /* if (bufseg == UIO_USERSPACE) */ {
501             error = copyout(sp, sfsp, sizeof(*sp));
502             free(sptmp, M_STATFS);
503             if (error != 0) {
504                 vfs_unbusy(mp);
505                 return (error);
506             }
507         }
508         sfsp++;
509     }
510     count++;
511     mtx_lock(&mountlist_mtx);
512     nmp = TAILQ_NEXT(mp, mnt_list);
513     vfs_unbusy(mp);
514 }
515 mtx_unlock(&mountlist_mtx);
516 if (sfsp != NULL && count > maxcount)
517     *countp = maxcount;
518 else
519     *countp = count;
520 return (0);
521 }
522
523 #ifdef COMPAT_FREEBSD4
524 /*
525  * Get old format filesystem statistics.
526  */
527 static void cvtstatfs(struct statfs *, struct ostatfs *);
528
529 #ifndef _SYS_SYSPROTO_H_
530 struct freebsd4_statfs_args {
531     char *path;
532     struct ostatfs *buf;
533 };
534 #endif
535 int
536 freebsd4_statfs(struct thread *td, struct freebsd4_statfs_args *uap)
537 {
538     struct ostatfs osb;
539     struct statfs *sfp;
540     int error;
541
542     sfp = malloc(sizeof(struct statfs), M_STATFS, M_WAITOK);
543     error = kern_statfs(td, uap->path, UIO_USERSPACE, sfp);
544     if (error == 0) {
545         cvtstatfs(sfp, &osb);
546         error = copyout(&osb, uap->buf, sizeof(osb));
547     }
548     free(sfp, M_STATFS);
549     return (error);
550 }
551
552 /*
553  * Get filesystem statistics.
554  */
555 #ifndef _SYS_SYSPROTO_H_
556 struct freebsd4_fstatfs_args {
557     int fd;
558     struct ostatfs *buf;
559 };
560 #endif
561 int
562 freebsd4_fstatfs(struct thread *td, struct freebsd4_fstatfs_args *uap)
563 {
564     struct ostatfs osb;
565     struct statfs *sfp;
566     int error;
567
568     sfp = malloc(sizeof(struct statfs), M_STATFS, M_WAITOK);

```

```

569     error = kern_fstatfs(td, uap->fd, sfp);
570     if (error == 0) {
571         cvtstatfs(sfp, &osb);
572         error = copyout(&osb, uap->buf, sizeof(osb));
573     }
574     free(sfp, M_STATFS);
575     return (error);
576 }
577
578 /*
579  * Get statistics on all filesystems.
580  */
581 #ifndef _SYS_SYSPROTO_H_
582 struct freebsd4_getfsstat_args {
583     struct ostatfs *buf;
584     long bufsize;
585     int mode;
586 };
587 #endif
588 int
589 freebsd4_getfsstat(struct thread *td, struct freebsd4_getfsstat_args *uap)
590 {
591     struct statfs *buf, *sp;
592     struct ostatfs osb;
593     size_t count, size;
594     int error;
595
596     if (uap->bufsize < 0)
597         return (EINVAL);
598     count = uap->bufsize / sizeof(struct ostatfs);
599     if (count > SIZE_MAX / sizeof(struct statfs))
600         return (EINVAL);
601     size = count * sizeof(struct statfs);
602     error = kern_getfsstat(td, &buf, size, &count, UIO_SYSSPACE,
603         uap->mode);
604     if (buf == NULL)
605         return (EINVAL);
606     td->td_retval[0] = count;
607     if (size != 0) {
608         sp = buf;
609         while (count != 0 && error == 0) {
610             cvtstatfs(sp, &osb);
611             error = copyout(&osb, uap->buf, sizeof(osb));
612             sp++;
613             uap->buf++;
614             count--;
615         }
616         free(buf, M_STATFS);
617     }
618     return (error);
619 }
620
621 /*
622  * Implement fstatfs() for (NFS) file handles.
623  */
624 #ifndef _SYS_SYSPROTO_H_
625 struct freebsd4_fhstatfs_args {
626     struct fhandle *u_fhp;
627     struct ostatfs *buf;
628 };
629 #endif
630 int
631 freebsd4_fhstatfs(struct thread *td, struct freebsd4_fhstatfs_args *uap)
632 {
633     struct ostatfs osb;
634     struct statfs *sfp;
635     fhandle_t fh;
636     int error;
637
638     error = copyin(uap->u_fhp, &fh, sizeof(fh));
639     if (error != 0)
640         return (error);
641     sfp = malloc(sizeof(struct statfs), M_STATFS, M_WAITOK);
642     error = kern_fhstatfs(td, fh, sfp);
643     if (error == 0) {
644         cvtstatfs(sfp, &osb);
645         error = copyout(&osb, uap->buf, sizeof(osb));
646     }
647     free(sfp, M_STATFS);
648     return (error);
649 }
650
651 /*
652  * Convert a new format statfs structure to an old format statfs structure.
653  */
654 static void
655 cvtstatfs(struct statfs *nsp, struct ostatfs *osp)
656 {
657
658     statfs_scale_blocks(nsp, LONG_MAX);
659     bzero(osp, sizeof(*osp));
660     osp->f_bsize = nsp->f_bsize;
661     osp->f_iosize = MIN(nsp->f_iosize, LONG_MAX);
662     osp->f_blocks = nsp->f_blocks;
663     osp->f_bfree = nsp->f_bfree;
664     osp->f_bavail = nsp->f_bavail;
665     osp->f_files = MIN(nsp->f_files, LONG_MAX);
666     osp->f_ffree = MIN(nsp->f_ffree, LONG_MAX);

```

```

667     osp->f_owner = nsp->f_owner;
668     osp->f_type = nsp->f_type;
669     osp->f_flags = nsp->f_flags;
670     osp->f_syncwrites = MIN(nsp->f_syncwrites, LONG_MAX);
671     osp->f_asyncwrites = MIN(nsp->f_asyncwrites, LONG_MAX);
672     osp->f_syncreads = MIN(nsp->f_syncreads, LONG_MAX);
673     osp->f_asyncreads = MIN(nsp->f_asyncreads, LONG_MAX);
674     strcpy(osp->f_stypename, nsp->f_stypename,
675            MIN(MFNSNAMELEN, OMFSNAMELEN));
676     strcpy(osp->f_mntonname, nsp->f_mntonname,
677            MIN(MNAMELEN, OMNAMELEN));
678     strcpy(osp->f_mntfromname, nsp->f_mntfromname,
679            MIN(MNAMELEN, OMNAMELEN));
680     osp->f_fsid = nsp->f_fsid;
681 }
682 #endif /* COMPAT_FREEBSD4 */
683
684 /*
685  * Change current working directory to a given file descriptor.
686  */
687 #ifndef _SYS_SYSPROTO_H_
688 struct fchdir_args {
689     int    fd;
690 };
691 #endif
692 int
693 sys_fchdir(struct thread *td, struct fchdir_args *uap)
694 {
695     struct vnode *vp, *tdp;
696     struct mount *mp;
697     struct file *fp;
698     cap_rights_t rights;
699     int error;
700
701     AUDIT_ARG_FD(uap->fd);
702     error = getvnode(td, uap->fd, cap_rights_init(&rights, CAP_FCHDIR),
703                    &fp);
704     if (error != 0)
705         return (error);
706     vp = fp->f_vnode;
707     vrefact(vp);
708     fdrop(fp, td);
709     vn_lock(vp, LK_SHARED | LK_RETRY);
710     AUDIT_ARG_VNODE1(vp);
711     error = change_dir(vp, td);
712     while (!error && (mp = vp->v_mountedhere) != NULL) {
713         if (vfs_busy(mp, 0))
714             continue;
715         error = VFS_ROOT(mp, LK_SHARED, &tdp);
716         vfs_unbusy(mp);
717         if (error != 0)
718             break;
719         vput(vp);
720         vp = tdp;
721     }
722     if (error != 0) {
723         vput(vp);
724         return (error);
725     }
726     VOP_UNLOCK(vp, 0);
727     pwd_chdir(td, vp);
728     return (0);
729 }
730
731 /*
732  * Change current working directory ('`.`').
733  */
734 #ifndef _SYS_SYSPROTO_H_
735 struct chdir_args {
736     char    *path;
737 };
738 #endif
739 int
740 sys_chdir(struct thread *td, struct chdir_args *uap)
741 {
742
743     return (kern_chdir(td, uap->path, UIO_USERSPACE));
744 }
745
746 int
747 kern_chdir(struct thread *td, char *path, enum uio_seg pathseg)
748 {
749     struct nameidata nd;
750     int error;
751
752     NDINIT(&nd, LOOKUP, FOLLOW | LOCKSHARED | LOCKLEAF | AUDITVNODE1,
753            pathseg, path, td);
754     if ((error = name1(&nd)) != 0)
755         return (error);
756     if ((error = change_dir(nd.ni_vp, td)) != 0) {
757         vput(nd.ni_vp);
758         NDFREE(&nd, NDF_ONLY_PNBUF);
759         return (error);
760     }
761     VOP_UNLOCK(nd.ni_vp, 0);
762     NDFREE(&nd, NDF_ONLY_PNBUF);
763     pwd_chdir(td, nd.ni_vp);
764     return (0);

```



```

765 }
766
767 /*
768  * Change notion of root (``/``) directory.
769  */
770 #ifndef _SYS_SYSPROTO_H_
771 struct chroot_args {
772     char    *path;
773 };
774 #endif
775 int
776 sys_chroot(struct thread *td, struct chroot_args *uap)
777 {
778     struct nameidata nd;
779     int error;
780
781     error = priv_check(td, PRIV_VFS_CHROOT);
782     if (error != 0)
783         return (error);
784     NDINIT(&nd, LOOKUP, FOLLOW | LOCKSHARED | LOCKLEAF | AUDITVNODE1,
785         UIO_USERSPACE, uap->path, td);
786     error = namei(&nd);
787     if (error != 0)
788         goto error;
789     error = change_dir(nd.ni_vp, td);
790     if (error != 0)
791         goto e_vunlock;
792 #ifdef MAC
793     error = mac_vnode_check_chroot(td->td_ucred, nd.ni_vp);
794     if (error != 0)
795         goto e_vunlock;
796 #endif
797     VOP_UNLOCK(nd.ni_vp, 0);
798     error = pwd_chroot(td, nd.ni_vp);
799     vrele(nd.ni_vp);
800     NDFREE(&nd, NDF_ONLY_PNBUF);
801     return (error);
802 e_vunlock:
803     vput(nd.ni_vp);
804 error:
805     NDFREE(&nd, NDF_ONLY_PNBUF);
806     return (error);
807 }
808
809 /*
810  * Common routine for chroot and chdir. Callers must provide a locked vnode
811  * instance.
812  */
813 int
814 change_dir(struct vnode *vp, struct thread *td)
815 {
816 #ifdef MAC
817     int error;
818 #endif
819
820     ASSERT_VOP_LOCKED(vp, "change_dir(): vp not locked");
821     if (vp->v_type != VDIR)
822         return (ENOTDIR);
823 #ifdef MAC
824     error = mac_vnode_check_chdir(td->td_ucred, vp);
825     if (error != 0)
826         return (error);
827 #endif
828     return (VOP_ACCESS(vp, VEXEC, td->td_ucred, td));
829 }
830
831 static __inline void
832 flags_to_rights(int flags, cap_rights_t *rightsp)
833 {
834
835     if (flags & O_EXEC) {
836         cap_rights_set(rightsp, CAP_FEXECVE);
837     } else {
838         switch ((flags & O_ACCMODE)) {
839             case O_RDONLY:
840                 cap_rights_set(rightsp, CAP_READ);
841                 break;
842             case O_RDWR:
843                 cap_rights_set(rightsp, CAP_READ);
844                 /* FALLTHROUGH */
845             case O_WRONLY:
846                 cap_rights_set(rightsp, CAP_WRITE);
847                 if (!(flags & (O_APPEND | O_TRUNC)))
848                     cap_rights_set(rightsp, CAP_SEEK);
849                 break;
850         }
851     }
852
853     if (flags & O_CREAT)
854         cap_rights_set(rightsp, CAP_CREATE);
855
856     if (flags & O_TRUNC)
857         cap_rights_set(rightsp, CAP_FTRUNCATE);
858
859     if (flags & (O_SYNC | O_FSYNC))
860         cap_rights_set(rightsp, CAP_FSYNC);
861
862     if (flags & (O_EXLOCK | O_SHLOCK))

```

```

864 }
865
866 /*
867  * Check permissions, allocate an open file structure, and call the device
868  * open routine if any.
869  */
870 #ifndef _SYS_SYSPROTO_H_
871 struct open_args {
872     char *path;
873     int flags;
874     int mode;
875 };
876 #endif
877 int
878 sys_open(struct thread *td, struct open_args *uap)
879 {
880
881     return (kern_openat(td, AT_FDCWD, uap->path, UIO_USERSPACE,
882                        uap->flags, uap->mode));
883 }
884
885 #ifndef _SYS_SYSPROTO_H_
886 struct openat_args {
887     int fd;
888     char *path;
889     int flag;
890     int mode;
891 };
892 #endif
893 int
894 sys_openat(struct thread *td, struct openat_args *uap)
895 {
896
897     AUDIT_ARG_FD(uap->fd);
898     return (kern_openat(td, uap->fd, uap->path, UIO_USERSPACE, uap->flag,
899                        uap->mode));
900 }
901
902 int
903 kern_openat(struct thread *td, int fd, char *path, enum uio_seg pathseg,
904             int flags, int mode)
905 {
906     struct proc *p = td->td_proc;
907     struct filedesc *fdp = p->p_fdp;
908     struct file *fp;
909     struct vnode *vp;
910     struct nameidata nd;
911     cap_rights_t rights;
912     int cmode, error, indx;
913
914     indx = -1;
915
916     AUDIT_ARG_FFLAGS(flags);
917     AUDIT_ARG_MODE(mode);
918     cap_rights_init(&rights, CAP_LOOKUP);
919     flags_to_rights(flags, &rights);
920     /*
921      * Only one of the O_EXEC, O_RDONLY, O_WRONLY and O_RDWR flags
922      * may be specified.
923      */
924     if (flags & O_EXEC) {
925         if (flags & O_ACCMODE)
926             return (EINVAL);
927     } else if ((flags & O_ACCMODE) == O_ACCMODE) {
928         return (EINVAL);
929     } else {
930         flags = FFLAGS(flags);
931     }
932
933     /*
934      * Allocate a file structure. The descriptor to reference it
935      * is allocated and set by finstall() below.
936      */
937     error = falloc_noinstall(td, &fp);
938     if (error != 0)
939         return (error);
940
941     /* An extra reference on `fp' has been held for us by
942      * falloc_noinstall().
943      */
944     /* Set the flags early so the finit in devfs can pick them up. */
945     fp->f_flag = flags & FMASK;
946     cmode = ((mode & ~fdp->fd_cmask) & ALLPERMS) & ~S_ISTXT;
947     NDINIT_ATRIGHTS(&nd, LOOKUP, FOLLOW | AUDITVNODE1, pathseg, path, fd,
948                    &rights, td);
949     td->td_dupfd = -1; /* XXX check for fdopen */
950     error = vn_open(&nd, &flags, cmode, fp);
951     if (error != 0) {
952         /*
953          * If the vn_open replaced the method vector, something
954          * wonderful happened deep below and we just pass it up
955          * pretending we know what we do.
956          */
957         if (error == ENXIO && fp->f_ops != &badfileops)
958             goto success;
959
960         /*

```

```

961     * Handle special fdopen() case. bleh.
962     *
963     * Don't do this for relative (capability) lookups; we don't
964     * understand exactly what would happen, and we don't think
965     * that it ever should.
966     */
967     if ((nd.ni_lcf & NI_LCF_STRICTRELATIVE) == 0 &&
968         (error == ENODEV || error == ENXIO) &&
969         td->td_dupfd >= 0) {
970         error = dupfdopen(td, fd, td->td_dupfd, flags, error,
971             &indx);
972         if (error == 0)
973             goto success;
974     }
975
976     goto bad;
977 }
978 td->td_dupfd = 0;
979 NDFREE(&nd, NDF_ONLY_PNBUF);
980 vp = nd.ni_vp;
981
982 /*
983  * Store the vnode, for any f_type. Typically, the vnode use
984  * count is decremented by direct call to vn_closefile() for
985  * files that switched type in the cdevsw fdopen() method.
986  */
987 fp->f_vnode = vp;
988 /*
989  * If the file wasn't claimed by devfs bind it to the normal
990  * vnode operations here.
991  */
992 if (fp->f_ops == &badfileops) {
993     KASSERT(vp->v_type != VFIFO, ("Unexpected fifo."));
994     fp->f_seqcount = 1;
995     finit(fp, (flags & FMASK) | (fp->f_flag & FHASLOCK),
996         DTYPE_VNODE, vp, &vnops);
997 }
998
999 VOP_UNLOCK(vp, 0);
1000 if (flags & O_TRUNC) {
1001     error = fo_truncate(fp, 0, td->td_ucred, td);
1002     if (error != 0)
1003         goto bad;
1004 }
1005 success:
1006 /*
1007  * If we haven't already installed the FD (for dupfdopen), do so now.
1008  */
1009 if (indx == -1) {
1010     struct filecaps *fcaps;
1011
1012     #ifdef CAPABILITIES
1013     if ((nd.ni_lcf & NI_LCF_STRICTRELATIVE) != 0)
1014         fcaps = &nd.ni_filecaps;
1015     else
1016     #endif
1017         fcaps = NULL;
1018     error = finstall(td, fp, &indx, flags, fcaps);
1019     /* On success finstall() consumes fcaps. */
1020     if (error != 0) {
1021         filecaps_free(&nd.ni_filecaps);
1022         goto bad;
1023     }
1024 } else {
1025     filecaps_free(&nd.ni_filecaps);
1026 }
1027
1028 /*
1029  * Release our private reference, leaving the one associated with
1030  * the descriptor table intact.
1031  */
1032 fdrop(fp, td);
1033 td->td_retval[0] = indx;
1034 return (0);
1035 bad:
1036 KASSERT(indx == -1, ("indx=%d, should be -1", indx));
1037 fdrop(fp, td);
1038 return (error);
1039 }
1040
1041 #ifdef COMPAT_43
1042 /*
1043  * Create a file.
1044  */
1045 #ifndef _SYS_SYSPROTO_H_
1046 struct ocreat_args {
1047     char    *path;
1048     int     mode;
1049 };
1050 #endif
1051 int
1052 ocreat(struct thread *td, struct ocreat_args *uap)
1053 {
1054
1055     return (kern_openat(td, AT_FDCWD, uap->path, UIO_USERSPACE,
1056         O_WRONLY | O_CREAT | O_TRUNC, uap->mode));
1057 }
1058 #endif /* COMPAT_43 */

```

```

1059
1060 /*
1061  * Create a special file.
1062  */
1063 #ifndef _SYS_SYSPROTO_H_
1064 struct mknod_args {
1065     char    *path;
1066     int     mode;
1067     int     dev;
1068 };
1069 #endif
1070 int
1071 sys_mknod(struct thread *td, struct mknod_args *uap)
1072 {
1073
1074     return (kern_mknodat(td, AT_FDCWD, uap->path, UIO_USERSPACE,
1075         uap->mode, uap->dev));
1076 }
1077
1078 #ifndef _SYS_SYSPROTO_H_
1079 struct mknodat_args {
1080     int     fd;
1081     char    *path;
1082     mode_t  mode;
1083     dev_t   dev;
1084 };
1085 #endif
1086 int
1087 sys_mknodat(struct thread *td, struct mknodat_args *uap)
1088 {
1089
1090     return (kern_mknodat(td, uap->fd, uap->path, UIO_USERSPACE, uap->mode,
1091         uap->dev));
1092 }
1093
1094 int
1095 kern_mknodat(struct thread *td, int fd, char *path, enum uio_seg pathseg,
1096     int mode, int dev)
1097 {
1098     struct vnode *vp;
1099     struct mount *mp;
1100     struct vattr vattr;
1101     struct nameidata nd;
1102     cap_rights_t rights;
1103     int error, whiteout = 0;
1104
1105     AUDIT_ARG_MODE(mode);
1106     AUDIT_ARG_DEV(dev);
1107     switch (mode & S_IFMT) {
1108     case S_IFCHR:
1109     case S_IFBLK:
1110         error = priv_check(td, PRIV_VFS_MKNOD_DEV);
1111         if (error == 0 && dev == VNOVAL)
1112             error = EINVAL;
1113         break;
1114     case S_IFWHT:
1115         error = priv_check(td, PRIV_VFS_MKNOD_WHT);
1116         break;
1117     case S_IFIFO:
1118         if (dev == 0)
1119             return (kern_mkfifoat(td, fd, path, pathseg, mode));
1120         /* FALLTHROUGH */
1121     default:
1122         error = EINVAL;
1123         break;
1124     }
1125     if (error != 0)
1126         return (error);
1127 restart:
1128     bwrite();
1129     NDINIT_ATRIGHTS(&nd, CREATE, LOCKPARENT | SAVENAME | AUDITVNODE1 |
1130         NOCACHE, pathseg, path, fd, cap_rights_init(&rights, CAP_MKNODAT),
1131     td);
1132     if ((error = namei(&nd)) != 0)
1133         return (error);
1134     vp = nd.ni_vp;
1135     if (vp != NULL) {
1136         NDFREE(&nd, NDF_ONLY_PNBUF);
1137         if (vp == nd.ni_dvp)
1138             vrele(nd.ni_dvp);
1139         else
1140             vput(nd.ni_dvp);
1141         vrele(vp);
1142         return (EEXIST);
1143     } else {
1144         VATTR_NULL(&vattr);
1145         vattr.va_mode = (mode & ALLPERMS) &
1146             ~td->td_proc->p_fd->fd_cmask;
1147         vattr.va_rdev = dev;
1148         whiteout = 0;
1149
1150         switch (mode & S_IFMT) {
1151         case S_IFCHR:
1152             vattr.va_type = VCHR;
1153             break;
1154         case S_IFBLK:
1155             vattr.va_type = VBLK;
1156             break;

```

```

1157         case S_IFWHT:
1158             whiteout = 1;
1159             break;
1160         default:
1161             panic("kern_mknod: invalid mode");
1162     }
1163 }
1164 if (vn_start_write(nd.ni_dvp, &mp, V_NOWAIT) != 0) {
1165     NDFREE(&nd, NDF_ONLY_PNBUF);
1166     vput(nd.ni_dvp);
1167     if ((error = vn_start_write(NULL, &mp, V_XSLEEP | PCATCH)) != 0)
1168         return (error);
1169     goto restart;
1170 }
1171 #ifdef MAC
1172 if (error == 0 && !whiteout)
1173     error = mac_vnode_check_create(td->td_ucred, nd.ni_dvp,
1174         &nd.ni_cnd, &vattr);
1175 #endif
1176 if (error == 0) {
1177     if (whiteout)
1178         error = VOP_WHITEOUT(nd.ni_dvp, &nd.ni_cnd, CREATE);
1179     else {
1180         error = VOP_MKNOD(nd.ni_dvp, &nd.ni_vp,
1181             &nd.ni_cnd, &vattr);
1182         if (error == 0)
1183             vput(nd.ni_vp);
1184     }
1185 }
1186 NDFREE(&nd, NDF_ONLY_PNBUF);
1187 vput(nd.ni_dvp);
1188 vn_finished_write(mp);
1189 return (error);
1190 }
1191
1192 /*
1193  * Create a named pipe.
1194  */
1195 #ifdef _SYS_SYSPROTO_H_
1196 struct mkfifo_args {
1197     char *path;
1198     int mode;
1199 };
1200 #endif
1201 int
1202 sys_mkfifo(struct thread *td, struct mkfifo_args *uap)
1203 {
1204     return (kern_mkfifo(td, AT_FDCWD, uap->path, UIO_USERSPACE,
1205         uap->mode));
1206 }
1207
1208 #ifdef _SYS_SYSPROTO_H_
1209 struct mkfifoat_args {
1210     int fd;
1211     char *path;
1212     mode_t mode;
1213 };
1214 #endif
1215 int
1216 sys_mkfifoat(struct thread *td, struct mkfifoat_args *uap)
1217 {
1218     return (kern_mkfifoat(td, uap->fd, uap->path, UIO_USERSPACE,
1219         uap->mode));
1220 }
1221
1222 int
1223 kern_mkfifoat(struct thread *td, int fd, char *path, enum uio_seg pathseg,
1224     int mode)
1225 {
1226     struct mount *mp;
1227     struct vattr vattr;
1228     struct nameidata nd;
1229     cap_rights_t rights;
1230     int error;
1231
1232     AUDIT_ARG_MODE(mode);
1233 restart:
1234     bwrite();
1235     NDINIT_ATRIGHTS(&nd, CREATE, LOCKPARENT | SAVENAME | AUDITVNODE1 |
1236         NOCACHE, pathseg, path, fd, cap_rights_init(&rights, CAP_MKFIFOAT),
1237         td);
1238     if ((error = namei(&nd)) != 0)
1239         return (error);
1240     if (nd.ni_vp != NULL) {
1241         NDFREE(&nd, NDF_ONLY_PNBUF);
1242         if (nd.ni_vp == nd.ni_dvp)
1243             vrele(nd.ni_dvp);
1244         else
1245             vput(nd.ni_dvp);
1246         vrele(nd.ni_vp);
1247         return (EEXIST);
1248     }
1249     if (vn_start_write(nd.ni_dvp, &mp, V_NOWAIT) != 0) {
1250         NDFREE(&nd, NDF_ONLY_PNBUF);
1251         vput(nd.ni_dvp);
1252         if ((error = vn_start_write(NULL, &mp, V_XSLEEP | PCATCH)) != 0)

```

```

1255         return (error);
1256         goto restart;
1257     }
1258     VATTR_NULL(&vattr);
1259     vattr.va_type = VFIFO;
1260     vattr.va_mode = (mode & ALLPERMS) & ~td->td_proc->p_fd->fd_cmask;
1261 #ifdef MAC
1262     error = mac_vnode_check_create(td->td_ucred, nd.ni_dvp, &nd.ni_cnd,
1263         &vattr);
1264     if (error != 0)
1265         goto out;
1266 #endif
1267     error = VOP_MKNOD(nd.ni_dvp, &nd.ni_vp, &nd.ni_cnd, &vattr);
1268     if (error == 0)
1269         vput(nd.ni_vp);
1270 #ifdef MAC
1271 out:
1272 #endif
1273     vput(nd.ni_dvp);
1274     vn_finished_write(mp);
1275     NDFREE(&nd, NDF_ONLY_PNBUF);
1276     return (error);
1277 }
1278
1279 /*
1280  * Make a hard file link.
1281  */
1282 #ifdef _SYS_SYSPROTO_H_
1283 struct link_args {
1284     char *path;
1285     char *link;
1286 };
1287 #endif
1288 int
1289 sys_link(struct thread *td, struct link_args *uap)
1290 {
1291     return (kern_linkat(td, AT_FDCWD, AT_FDCWD, uap->path, uap->link,
1292         UIO_USERSPACE, FOLLOW));
1293 }
1294
1295 #ifdef _SYS_SYSPROTO_H_
1296 struct linkat_args {
1297     int fd1;
1298     char *path1;
1299     int fd2;
1300     char *path2;
1301     int flag;
1302 };
1303 #endif
1304 int
1305 sys_linkat(struct thread *td, struct linkat_args *uap)
1306 {
1307     int flag;
1308
1309     flag = uap->flag;
1310     if (flag & ~AT_SYMLINK_FOLLOW)
1311         return (EINVAL);
1312
1313     return (kern_linkat(td, uap->fd1, uap->fd2, uap->path1, uap->path2,
1314         UIO_USERSPACE, (flag & AT_SYMLINK_FOLLOW) ? FOLLOW : NOFOLLOW));
1315 }
1316
1317 int hardlink_check_uid = 0;
1318 SYSCTL_INT(_security_bsd, OID_AUTO, hardlink_check_uid, CTLFLAG_RW,
1319     &hardlink_check_uid, 0,
1320     "Unprivileged processes cannot create hard links to files owned by other "
1321     "users");
1322 static int hardlink_check_gid = 0;
1323 SYSCTL_INT(_security_bsd, OID_AUTO, hardlink_check_gid, CTLFLAG_RW,
1324     &hardlink_check_gid, 0,
1325     "Unprivileged processes cannot create hard links to files owned by other "
1326     "groups");
1327
1328 static int
1329 can_hardlink(struct vnode *vp, struct ucred *cred)
1330 {
1331     struct vattr va;
1332     int error;
1333
1334     if (!hardlink_check_uid && !hardlink_check_gid)
1335         return (0);
1336
1337     error = VOP_GETATTR(vp, &va, cred);
1338     if (error != 0)
1339         return (error);
1340
1341     if (hardlink_check_uid && cred->cr_uid != va.va_uid) {
1342         error = priv_check_cred(cred, PRIV_VFS_LINK, 0);
1343         if (error != 0)
1344             return (error);
1345     }
1346
1347     if (hardlink_check_gid && !groupmember(va.va_gid, cred)) {
1348         error = priv_check_cred(cred, PRIV_VFS_LINK, 0);
1349         if (error != 0)
1350             return (error);
1351     }
1352 }

```

```

1353
1354     return (0);
1355 }
1356
1357 int
1358 kern_linkat(struct thread *td, int fd1, int fd2, char *path1, char *path2,
1359     enum uio_seg segflg, int follow)
1360 {
1361     struct vnode *vp;
1362     struct mount *mp;
1363     struct nameidata nd;
1364     cap_rights_t rights;
1365     int error;
1366
1367     again:
1368     bwrite();
1369     NDINIT_ATRIGHTS(&nd, LOOKUP, follow | AUDITVNODE1, segflg, path1, fd1,
1370         cap_rights_init(&rights, CAP_LINKAT_SOURCE), td);
1371
1372     if ((error = namei(&nd)) != 0)
1373         return (error);
1374     NDFREE(&nd, NDF_ONLY_PNBUF);
1375     vp = nd.ni_vp;
1376     if (vp->v_type == VDIR) {
1377         vrele(vp);
1378         return (EPERM);        /* POSIX */
1379     }
1380     NDINIT_ATRIGHTS(&nd, CREATE,
1381         LOCKPARENT | SAVENAME | AUDITVNODE2 | NOCACHE, segflg, path2, fd2,
1382         cap_rights_init(&rights, CAP_LINKAT_TARGET), td);
1383     if ((error = namei(&nd)) == 0) {
1384         if (nd.ni_vp != NULL) {
1385             NDFREE(&nd, NDF_ONLY_PNBUF);
1386             if (nd.ni_dvp == nd.ni_vp)
1387                 vrele(nd.ni_dvp);
1388             else
1389                 vput(nd.ni_dvp);
1390             vrele(nd.ni_vp);
1391             vrele(vp);
1392             return (EEXIST);
1393         } else if (nd.ni_dvp->v_mount != vp->v_mount) {
1394             /*
1395              * Cross-device link. No need to recheck
1396              * vp->v_type, since it cannot change, except
1397              * to VBAD.
1398              */
1399             NDFREE(&nd, NDF_ONLY_PNBUF);
1400             vput(nd.ni_dvp);
1401             vrele(vp);
1402             return (EXDEV);
1403         } else if ((error = vn_lock(vp, LK_EXCLUSIVE)) == 0) {
1404             error = can_hardlink(vp, td->td_ucred);
1405
1406             #ifdef MAC
1407             if (error == 0)
1408                 error = mac_vnode_check_link(td->td_ucred,
1409                     nd.ni_dvp, vp, &nd.ni_cnd);
1410
1411             #endif
1412
1413             if (error != 0) {
1414                 vput(vp);
1415                 vput(nd.ni_dvp);
1416                 NDFREE(&nd, NDF_ONLY_PNBUF);
1417                 return (error);
1418             }
1419             error = vn_start_write(vp, &mp, V_NOWAIT);
1420             if (error != 0) {
1421                 vput(vp);
1422                 vput(nd.ni_dvp);
1423                 NDFREE(&nd, NDF_ONLY_PNBUF);
1424                 error = vn_start_write(NULL, &mp,
1425                     V_XSLEEP | PCATCH);
1426                 if (error != 0)
1427                     return (error);
1428                 goto again;
1429             }
1430             error = VOP_LINK(nd.ni_dvp, vp, &nd.ni_cnd);
1431             VOP_UNLOCK(vp, 0);
1432             vput(nd.ni_dvp);
1433             vn_finished_write(mp);
1434             NDFREE(&nd, NDF_ONLY_PNBUF);
1435         } else {
1436             vput(nd.ni_dvp);
1437             NDFREE(&nd, NDF_ONLY_PNBUF);
1438             vrele(vp);
1439             goto again;
1440         }
1441     }
1442     vrele(vp);
1443     return (error);
1444 }
1445
1446 /*
1447  * Make a symbolic link.
1448  */
1449 #ifndef _SYS_SYSPROTO_H_
1450 struct symlink_args {
1451     char *path;
1452     char *link;
1453 };

```

```

1451 #endif
1452 int
1453 sys_symlink(struct thread *td, struct symlink_args *uap)
1454 {
1455
1456     return (kern_symlinkat(td, uap->path, AT_FDCWD, uap->link,
1457         UIO_USERSPACE));
1458 }
1459
1460 #ifndef _SYS_SYSPROTO_H_
1461 struct symlinkat_args {
1462     char    *path;
1463     int     fd;
1464     char    *path2;
1465 };
1466 #endif
1467 int
1468 sys_symlinkat(struct thread *td, struct symlinkat_args *uap)
1469 {
1470
1471     return (kern_symlinkat(td, uap->path1, uap->fd, uap->path2,
1472         UIO_USERSPACE));
1473 }
1474
1475 int
1476 kern_symlinkat(struct thread *td, char *path1, int fd, char *path2,
1477     enum uio_seg segflg)
1478 {
1479     struct mount *mp;
1480     struct vattr vattr;
1481     char *syspath;
1482     struct nameidata nd;
1483     int error;
1484     cap_rights_t rights;
1485
1486     if (segflg == UIO_SYSSPACE) {
1487         syspath = path1;
1488     } else {
1489         syspath = uma_zalloc(namei_zone, M_WAITOK);
1490         if ((error = copyinstr(path1, syspath, MAXPATHLEN, NULL)) != 0)
1491             goto out;
1492     }
1493     AUDIT_ARG_TEXT(syspath);
1494 restart:
1495     bwrite();
1496     NDINIT(&nd, CREATE, LOCKPARENT | SAVENAME | AUDITVNODE1 |
1497         NOCACHE, segflg, path2, fd, cap_rights_init(&rights, CAP_SYMLINKAT),
1498         td);
1499     if ((error = namei(&nd)) != 0)
1500         goto out;
1501     if (nd.ni_vp) {
1502         NDFREE(&nd, NDF_ONLY_PNBUF);
1503         if (nd.ni_vp == nd.ni_dvp)
1504             vrele(nd.ni_dvp);
1505         else
1506             vput(nd.ni_dvp);
1507         vrele(nd.ni_vp);
1508         error = EEXIST;
1509         goto out;
1510     }
1511     if (vn_start_write(nd.ni_dvp, &mp, V_NOWAIT) != 0) {
1512         NDFREE(&nd, NDF_ONLY_PNBUF);
1513         vput(nd.ni_dvp);
1514         if ((error = vn_start_write(NULL, &mp, V_XSLEEP | PCATCH)) != 0)
1515             goto out;
1516         goto restart;
1517     }
1518     VATTR_NULL(&vattr);
1519     vattr.va_mode = ACCESSPERMS &~ td->td_proc->p_fd->fd_cmask;
1520 #ifdef MAC
1521     vattr.va_type = VLNK;
1522     error = mac_vnode_check_create(td->td_ucred, nd.ni_dvp, &nd.ni_cnd,
1523         &vattr);
1524     if (error != 0)
1525         goto out2;
1526 #endif
1527     error = VOP_SYMLINK(nd.ni_dvp, &nd.ni_vp, &nd.ni_cnd, &vattr, syspath);
1528     if (error == 0)
1529         vput(nd.ni_vp);
1530 #ifdef MAC
1531 out2:
1532 #endif
1533     NDFREE(&nd, NDF_ONLY_PNBUF);
1534     vput(nd.ni_dvp);
1535     vn_finished_write(mp);
1536 out:
1537     if (segflg != UIO_SYSSPACE)
1538         uma_zfree(namei_zone, syspath);
1539     return (error);
1540 }
1541
1542 /*
1543  * Delete a whiteout from the filesystem.
1544  */
1545 #ifndef _SYS_SYSPROTO_H_
1546 struct undelete_args {
1547     char *path;
1548 };

```



```

1549 #endif
1550 int
1551 sys_undelete(struct thread *td, struct undelete_args *uap)
1552 {
1553     struct mount *mp;
1554     struct nameidata nd;
1555     int error;
1556
1557 restart:
1558     bwrite();
1559     NDINIT(&nd, DELETE, LOCKPARENT | DOMHIDEOUT | AUDITVNODE1,
1560         UIO_USERSPACE, uap->path, td);
1561     error = namei(&nd);
1562     if (error != 0)
1563         return (error);
1564
1565     if (nd.ni_vp != NULLVP || !(nd.ni_cnd.cn_flags & ISWHITEOUT)) {
1566         NDFREE(&nd, NDF_ONLY_PNBUF);
1567         if (nd.ni_vp == nd.ni_dvp)
1568             vrelease(nd.ni_dvp);
1569         else
1570             vput(nd.ni_dvp);
1571         if (nd.ni_vp)
1572             vrelease(nd.ni_vp);
1573         return (EEXIST);
1574     }
1575     if (vn_start_write(nd.ni_dvp, &mp, V_NOWAIT) != 0) {
1576         NDFREE(&nd, NDF_ONLY_PNBUF);
1577         vput(nd.ni_dvp);
1578         if ((error = vn_start_write(NULL, &mp, V_XSLEEP | PCATCH)) != 0)
1579             return (error);
1580         goto restart;
1581     }
1582     error = VOP_WHITEOUT(nd.ni_dvp, &nd.ni_cnd, DELETE);
1583     NDFREE(&nd, NDF_ONLY_PNBUF);
1584     vput(nd.ni_dvp);
1585     vn_finished_write(mp);
1586     return (error);
1587 }
1588
1589 /*
1590  * Delete a name from the filesystem.
1591  */
1592 #ifndef _SYS_SYSPROTO_H_
1593 struct unlink_args {
1594     char *path;
1595 };
1596 #endif
1597 int
1598 sys_unlink(struct thread *td, struct unlink_args *uap)
1599 {
1600     return (kern_unlinkat(td, AT_FDCWD, uap->path, UIO_USERSPACE, 0));
1601 }
1602
1603 #ifndef _SYS_SYSPROTO_H_
1604 struct unlinkat_args {
1605     int fd;
1606     char *path;
1607     int flag;
1608 };
1609 #endif
1610 int
1611 sys_unlinkat(struct thread *td, struct unlinkat_args *uap)
1612 {
1613     int flag = uap->flag;
1614     int fd = uap->fd;
1615     char *path = uap->path;
1616
1617     if (flag & ~AT_REMOVEDIR)
1618         return (EINVAL);
1619
1620     if (flag & AT_REMOVEDIR)
1621         return (kern_rmdirat(td, fd, path, UIO_USERSPACE));
1622     else
1623         return (kern_unlinkat(td, fd, path, UIO_USERSPACE, 0));
1624 }
1625
1626 int
1627 kern_unlinkat(struct thread *td, int fd, char *path, enum uio_seg pathseg,
1628     ino_t oldinum)
1629 {
1630     struct mount *mp;
1631     struct vnode *vp;
1632     struct nameidata nd;
1633     struct stat sb;
1634     cap_rights_t rights;
1635     int error;
1636
1637 restart:
1638     bwrite();
1639     NDINIT_ATRIGHTS(&nd, DELETE, LOCKPARENT | LOCKLEAF | AUDITVNODE1,
1640         pathseg, path, fd, cap_rights_init(&rights, CAP_UNLINKAT), td);
1641     if ((error = namei(&nd)) != 0)
1642         return (error == EINVAL ? EPERM : error);
1643     vp = nd.ni_vp;
1644     if (vp->v_type == VDIR && oldinum == 0) {
1645         error = EPERM; /* POSIX */

```

```

1647     } else if (oldinum != 0 &&
1648               ((error = vn_stat(vp, &sb, td->td_ucred, NOCRED, td)) == 0) &&
1649               sb.st_ino != oldinum) {
1650         error = EIDRM; /* Identifier removed */
1651     } else {
1652         /*
1653          * The root of a mounted filesystem cannot be deleted.
1654          *
1655          * XXX: can this only be a VDIR case?
1656          */
1657         if (vp->v_vflag & VW_ROOT)
1658             error = EBUSY;
1659     }
1660     if (error == 0) {
1661         if (vn_start_write(nd.ni_dvp, &mp, V_NOMAIT) != 0) {
1662             NDFREE(&nd, NDF_ONLY_PNBUF);
1663             vput(nd.ni_dvp);
1664             if (vp == nd.ni_dvp)
1665                 vrele(vp);
1666             else
1667                 vput(vp);
1668             if ((error = vn_start_write(NULL, &mp,
1669                                       V_XSLEEP | PCATCH)) != 0)
1670                 return (error);
1671             goto restart;
1672         }
1673 #ifdef MAC
1674         error = mac_vnode_check_unlink(td->td_ucred, nd.ni_dvp, vp,
1675                                       &nd.ni_cnd);
1676         if (error != 0)
1677             goto out;
1678 #endif
1679         vfs_notify_upper(vp, VFS_NOTIFY_UPPER_UNLINK);
1680         error = VOP_REMOVE(nd.ni_dvp, vp, &nd.ni_cnd);
1681 #ifdef MAC
1682 out:
1683 #endif
1684         vn_finished_write(mp);
1685     }
1686     NDFREE(&nd, NDF_ONLY_PNBUF);
1687     vput(nd.ni_dvp);
1688     if (vp == nd.ni_dvp)
1689         vrele(vp);
1690     else
1691         vput(vp);
1692     return (error);
1693 }
1694
1695 /*
1696  * Reposition read/write file offset.
1697  */
1698 #ifndef _SYS_SYSPROTO_H_
1699 struct lseek_args {
1700     int    fd;
1701     int    pad;
1702     off_t  offset;
1703     int    whence;
1704 };
1705 #endif
1706 int
1707 sys_lseek(struct thread *td, struct lseek_args *uap)
1708 {
1709     return (kern_lseek(td, uap->fd, uap->offset, uap->whence));
1710 }
1711
1712 int
1713 kern_lseek(struct thread *td, int fd, off_t offset, int whence)
1714 {
1715     struct file *fp;
1716     cap_rights_t rights;
1717     int error;
1718
1719     AUDIT_ARG_FD(fd);
1720     error = fget(td, fd, cap_rights_init(&rights, CAP_SEEK), &fp);
1721     if (error != 0)
1722         return (error);
1723     error = (fp->f_ops->fo_flags & DFLAG_SEEKABLE) != 0 ?
1724         fo_seek(fp, offset, whence, td) : ESPIPE;
1725     fdrop(fp, td);
1726     return (error);
1727 }
1728
1729 #if defined(COMPAT_43)
1730 /*
1731  * Reposition read/write file offset.
1732  */
1733 #ifndef _SYS_SYSPROTO_H_
1734 struct olseek_args {
1735     int    fd;
1736     long   offset;
1737     int    whence;
1738 };
1739 #endif
1740 int
1741 olseek(struct thread *td, struct olseek_args *uap)
1742 {
1743
1744

```

```

1745         return (kern_lseek(td, uap->fd, uap->offset, uap->whence));
1746     }
1747     #endif /* COMPAT_43 */
1748
1749     #if defined(COMPAT_FREEBSD6)
1750     /* Version with the 'pad' argument */
1751     int
1752     freebsd6_lseek(struct thread *td, struct freebsd6_lseek_args *uap)
1753     {
1754
1755         return (kern_lseek(td, uap->fd, uap->offset, uap->whence));
1756     }
1757     #endif
1758
1759     /*
1760     * Check access permissions using passed credentials.
1761     */
1762     static int
1763     vn_access(struct vnode *vp, int user_flags, struct ucred *cred,
1764              struct thread *td)
1765     {
1766         accmode_t accmode;
1767         int error;
1768
1769         /* Flags == 0 means only check for existence. */
1770         if (user_flags == 0)
1771             return (0);
1772
1773         accmode = 0;
1774         if (user_flags & R_OK)
1775             accmode |= VREAD;
1776         if (user_flags & W_OK)
1777             accmode |= VWRITE;
1778         if (user_flags & X_OK)
1779             accmode |= VEXEC;
1780
1781         #ifdef MAC
1782         error = mac_vnode_check_access(cred, vp, accmode);
1783         if (error != 0)
1784             return (error);
1785         #endif
1786
1787         if ((accmode & VWRITE) == 0 || (error = vn_writechk(vp)) == 0)
1788             error = VOP_ACCESS(vp, accmode, cred, td);
1789         return (error);
1790     }
1791
1792     /*
1793     * Check access permissions using "real" credentials.
1794     */
1795     #ifndef _SYS_SYSPROTO_H_
1796     struct access_args {
1797         char *path;
1798         int amode;
1799     };
1800     #endif
1801     int
1802     sys_access(struct thread *td, struct access_args *uap)
1803     {
1804
1805         return (kern_accessat(td, AT_FDCWD, uap->path, UIO_USERSPACE,
1806                               0, uap->amode));
1807     }
1808
1809     #ifndef _SYS_SYSPROTO_H_
1810     struct faccessat_args {
1811         int dirfd;
1812         char *path;
1813         int amode;
1814         int flag;
1815     };
1816     #endif
1817     int
1818     sys_faccessat(struct thread *td, struct faccessat_args *uap)
1819     {
1820
1821         return (kern_accessat(td, uap->fd, uap->path, UIO_USERSPACE, uap->flag,
1822                               uap->amode));
1823     }
1824
1825     int
1826     kern_accessat(struct thread *td, int fd, char *path, enum uio_seg pathseg,
1827                  int flag, int amode)
1828     {
1829         struct ucred *cred, *ucred;
1830         struct vnode *vp;
1831         struct nameidata nd;
1832         cap_rights_t rights;
1833         int error;
1834
1835         if (flag & ~AT_EACCESS)
1836             return (EINVAL);
1837
1838         if (amode != F_OK && (amode & ~(R_OK | W_OK | X_OK)) != 0)
1839             return (EINVAL);
1840
1841         /*
1842         * Create and modify a temporary credential instead of one that
1843         * is potentially shared (if we need one).
1844         */
1845         cred = td->td_ucred;

```

```

1843     if ((flag & AT_EACCESS) == 0 &&
1844         ((cred->cr_uid != cred->cr_ruid ||
1845          cred->cr_rgid != cred->cr_groups[0]))) {
1846         usecred = crdup(cred);
1847         usecred->cr_uid = cred->cr_ruid;
1848         usecred->cr_groups[0] = cred->cr_rgid;
1849         td->td_ucred = usecred;
1850     } else
1851         usecred = cred;
1852     AUDIT_ARG_VALUE(amode);
1853     NDINIT_ATRIGHTS(&nd, LOOKUP, FOLLOW | LOCKSHARED | LOCKLEAF |
1854                    AUDITVNODE1, pathseg, path, fd, cap_rights_init(&rights, CAP_FSTAT),
1855                    td);
1856     if ((error = namei(&nd)) != 0)
1857         goto out;
1858     vp = nd.ni_vp;
1859
1860     error = vn_access(vp, amode, usecred, td);
1861     NDFREE(&nd, NDF_ONLY_PNBUF);
1862     vput(vp);
1863 out:
1864     if (usecred != cred) {
1865         td->td_ucred = cred;
1866         crfree(usecred);
1867     }
1868     return (error);
1869 }
1870
1871 /*
1872  * Check access permissions using "effective" credentials.
1873  */
1874 #ifndef _SYS_SYSPROTO_H_
1875 struct eaccess_args {
1876     char    *path;
1877     int     amode;
1878 };
1879 #endif
1880 int
1881 sys_eaccess(struct thread *td, struct eaccess_args *uap)
1882 {
1883
1884     return (kern_accessat(td, AT_FDCWD, uap->path, UIO_USERSPACE,
1885                          AT_EACCESS, uap->amode));
1886 }
1887
1888 #if defined(COMPAT_43)
1889 /*
1890  * Get file status; this version follows links.
1891  */
1892 #ifndef _SYS_SYSPROTO_H_
1893 struct ostat_args {
1894     char    *path;
1895     struct  ostat *ub;
1896 };
1897 #endif
1898 int
1899 ostat(struct thread *td, struct ostat_args *uap)
1900 {
1901     struct stat sb;
1902     struct ostat osb;
1903     int error;
1904
1905     error = kern_statat(td, 0, AT_FDCWD, uap->path, UIO_USERSPACE,
1906                       &sb, NULL);
1907     if (error != 0)
1908         return (error);
1909     cvtstat(&sb, &osb);
1910     return (copyout(&osb, uap->ub, sizeof (osb)));
1911 }
1912
1913 /*
1914  * Get file status; this version does not follow links.
1915  */
1916 #ifndef _SYS_SYSPROTO_H_
1917 struct olstat_args {
1918     char    *path;
1919     struct  ostat *ub;
1920 };
1921 #endif
1922 int
1923 olstat(struct thread *td, struct olstat_args *uap)
1924 {
1925     struct stat sb;
1926     struct ostat osb;
1927     int error;
1928
1929     error = kern_statat(td, AT_SYMLINK_NOFOLLOW, AT_FDCWD, uap->path,
1930                       UIO_USERSPACE, &sb, NULL);
1931     if (error != 0)
1932         return (error);
1933     cvtstat(&sb, &osb);
1934     return (copyout(&osb, uap->ub, sizeof (osb)));
1935 }
1936
1937 /*
1938  * Convert from an old to a new stat structure.
1939  */
1940 void

```

```

1941 cvtstat(struct stat *st, struct ostat *ost)
1942 {
1943
1944     bzero(ost, sizeof(*ost));
1945     ost->st_dev = st->st_dev;
1946     ost->st_ino = st->st_ino;
1947     ost->st_mode = st->st_mode;
1948     ost->st_nlink = st->st_nlink;
1949     ost->st_uid = st->st_uid;
1950     ost->st_gid = st->st_gid;
1951     ost->st_rdev = st->st_rdev;
1952     if (st->st_size < (quad_t)1 << 32)
1953         ost->st_size = st->st_size;
1954     else
1955         ost->st_size = -2;
1956     ost->st_atim = st->st_atim;
1957     ost->st_mtim = st->st_mtim;
1958     ost->st_ctim = st->st_ctim;
1959     ost->st_blksize = st->st_blksize;
1960     ost->st_blocks = st->st_blocks;
1961     ost->st_flags = st->st_flags;
1962     ost->st_gen = st->st_gen;
1963 }
1964 #endif /* COMPAT_A3 */
1965
1966 /*
1967  * Get file status; this version follows links.
1968  */
1969 #ifndef _SYS_SYSPROTO_H_
1970 struct stat_args {
1971     char    *path;
1972     struct stat *ub;
1973 };
1974 #endif
1975 int
1976 sys_stat(struct thread *td, struct stat_args *uap)
1977 {
1978     struct stat sb;
1979     int error;
1980
1981     error = kern_statat(td, 0, AT_FDCWD, uap->path, UIO_USERSPACE,
1982         &sb, NULL);
1983     if (error == 0)
1984         error = copyout(&sb, uap->ub, sizeof (sb));
1985     return (error);
1986 }
1987
1988 #ifndef _SYS_SYSPROTO_H_
1989 struct fstatat_args {
1990     int    fd;
1991     char    *path;
1992     struct stat *buf;
1993     int    flag;
1994 };
1995 #endif
1996 int
1997 sys_fstatat(struct thread *td, struct fstatat_args *uap)
1998 {
1999     struct stat sb;
2000     int error;
2001
2002     error = kern_statat(td, uap->flag, uap->fd, uap->path,
2003         UIO_USERSPACE, &sb, NULL);
2004     if (error == 0)
2005         error = copyout(&sb, uap->buf, sizeof (sb));
2006     return (error);
2007 }
2008
2009 int
2010 kern_statat(struct thread *td, int flag, int fd, char *path,
2011     enum uio_seg pathseg, struct stat *sbp,
2012     void (*hook)(struct vnode *vp, struct stat *sbp))
2013 {
2014     struct nameidata nd;
2015     struct stat sb;
2016     cap_rights_t rights;
2017     int error;
2018
2019     if (flag & ~AT_SYMLINK_NOFOLLOW)
2020         return (EINVAL);
2021
2022     NDINIT_ATRIGHTS(&nd, LOOKUP, ((flag & AT_SYMLINK_NOFOLLOW) ? NOFOLLOW :
2023         FOLLOW) | LOCKSHARED | LOCKLEAF | AUDITVNODE1, pathseg, path, fd,
2024         cap_rights_init(&rights, CAP_FSTAT), td);
2025
2026     if ((error = namei(&nd)) != 0)
2027         return (error);
2028     error = vn_stat(nd.ni_vp, &sb, td->td_ucred, NOCRED, td);
2029     if (error == 0) {
2030         SDT_PROBE2(vfs, , stat, mode, path, sb.st_mode);
2031         if (S_ISREG(sb.st_mode))
2032             SDT_PROBE2(vfs, , stat, reg, path, pathseg);
2033         if (__predict_false(hook != NULL))
2034             hook(nd.ni_vp, &sb);
2035     }
2036     NDFREE(&nd, NDF_ONLY_PNBUF);
2037     vput(nd.ni_vp);
2038     if (error != 0)

```

```

2039         return (error);
2040     *sbp = sb;
2041 #ifdef KTRACE
2042     if (KTRPOINT(td, KTR_STRUCT))
2043         ktrstat(&sb);
2044 #endif
2045     return (0);
2046 }
2047
2048 /*
2049  * Get file status; this version does not follow links.
2050  */
2051 #ifndef _SYS_SYSPROTO_H_
2052 struct lstat_args {
2053     char *path;
2054     struct stat *ub;
2055 };
2056 #endif
2057 int
2058 sys_lstat(struct thread *td, struct lstat_args *uap)
2059 {
2060     struct stat sb;
2061     int error;
2062
2063     error = kern_statat(td, AT_SYMLINK_NOFOLLOW, AT_FDCWD, uap->path,
2064         UIO_USERSPACE, &sb, NULL);
2065     if (error == 0)
2066         error = copyout(&sb, uap->ub, sizeof (sb));
2067     return (error);
2068 }
2069
2070 /*
2071  * Implementation of the NetBSD [l]stat() functions.
2072  */
2073 void
2074 cvtnstat( struct stat *sb, struct nstat *nsb)
2075 {
2076     bzero(nsb, sizeof *nsb);
2077     nsb->st_dev = sb->st_dev;
2078     nsb->st_ino = sb->st_ino;
2079     nsb->st_mode = sb->st_mode;
2080     nsb->st_nlink = sb->st_nlink;
2081     nsb->st_uid = sb->st_uid;
2082     nsb->st_gid = sb->st_gid;
2083     nsb->st_rdev = sb->st_rdev;
2084     nsb->st_atim = sb->st_atim;
2085     nsb->st_mtim = sb->st_mtim;
2086     nsb->st_ctim = sb->st_ctim;
2087     nsb->st_size = sb->st_size;
2088     nsb->st_blocks = sb->st_blocks;
2089     nsb->st_blksize = sb->st_blksize;
2090     nsb->st_flags = sb->st_flags;
2091     nsb->st_gen = sb->st_gen;
2092     nsb->st_birthtim = sb->st_birthtim;
2093 }
2094
2095 #ifndef _SYS_SYSPROTO_H_
2096 struct nstat_args {
2097     char *path;
2098     struct nstat *ub;
2099 };
2100 #endif
2101 int
2102 sys_nstat(struct thread *td, struct nstat_args *uap)
2103 {
2104     struct stat sb;
2105     struct nstat nsb;
2106     int error;
2107
2108     error = kern_statat(td, 0, AT_FDCWD, uap->path, UIO_USERSPACE,
2109         &sb, NULL);
2110     if (error != 0)
2111         return (error);
2112     cvtnstat(&sb, &nsb);
2113     return (copyout(&nsb, uap->ub, sizeof (nsb)));
2114 }
2115
2116 /*
2117  * NetBSD lstat. Get file status; this version does not follow links.
2118  */
2119 #ifndef _SYS_SYSPROTO_H_
2120 struct lstat_args {
2121     char *path;
2122     struct stat *ub;
2123 };
2124 #endif
2125 int
2126 sys_nlstat(struct thread *td, struct nlstat_args *uap)
2127 {
2128     struct stat sb;
2129     struct nstat nsb;
2130     int error;
2131
2132     error = kern_statat(td, AT_SYMLINK_NOFOLLOW, AT_FDCWD, uap->path,
2133         UIO_USERSPACE, &sb, NULL);
2134     if (error != 0)
2135         return (error);

```

```

2137     cvtnstat(&sb, &nsb);
2138     return (copyout(&nsb, uap->ub, sizeof (nsb)));
2139 }
2140
2141 /*
2142  * Get configurable pathname variables.
2143  */
2144 #ifndef _SYS_SYSPROTO_H_
2145 struct pathconf_args {
2146     char    *path;
2147     int     name;
2148 };
2149 #endif
2150 int
2151 sys_pathconf(struct thread *td, struct pathconf_args *uap)
2152 {
2153
2154     return (kern_pathconf(td, uap->path, UIO_USERSPACE, uap->name, FOLLOW));
2155 }
2156
2157 #ifndef _SYS_SYSPROTO_H_
2158 struct lpathconf_args {
2159     char    *path;
2160     int     name;
2161 };
2162 #endif
2163 int
2164 sys_lpathconf(struct thread *td, struct lpathconf_args *uap)
2165 {
2166
2167     return (kern_pathconf(td, uap->path, UIO_USERSPACE, uap->name,
2168         NOFOLLOW));
2169 }
2170
2171 int
2172 kern_pathconf(struct thread *td, char *path, enum uio_seg pathseg, int name,
2173     u_long flags)
2174 {
2175     struct nameidata nd;
2176     int error;
2177
2178     NDINIT(&nd, LOOKUP, LOCKSHARED | LOCKLEAF | AUDITVNODE1 | flags,
2179         pathseg, path, td);
2180     if ((error = namei(&nd)) != 0)
2181         return (error);
2182     NDFREE(&nd, NDF_ONLY_PNBUF);
2183
2184     error = VOP_PATHCONF(nd.ni_vp, name, td->td_retval);
2185     vput(nd.ni_vp);
2186     return (error);
2187 }
2188
2189 /*
2190  * Return target name of a symbolic link.
2191  */
2192 #ifndef _SYS_SYSPROTO_H_
2193 struct readlink_args {
2194     char    *path;
2195     char    *buf;
2196     size_t  count;
2197 };
2198 #endif
2199 int
2200 sys_readlink(struct thread *td, struct readlink_args *uap)
2201 {
2202
2203     return (kern_readlinkat(td, AT_FDCWD, uap->path, UIO_USERSPACE,
2204         uap->buf, UIO_USERSPACE, uap->count));
2205 }
2206 #ifndef _SYS_SYSPROTO_H_
2207 struct readlinkat_args {
2208     int     fd;
2209     char    *path;
2210     char    *buf;
2211     size_t  bufsize;
2212 };
2213 #endif
2214 int
2215 sys_readlinkat(struct thread *td, struct readlinkat_args *uap)
2216 {
2217
2218     return (kern_readlinkat(td, uap->fd, uap->path, UIO_USERSPACE,
2219         uap->buf, UIO_USERSPACE, uap->bufsize));
2220 }
2221
2222 int
2223 kern_readlinkat(struct thread *td, int fd, char *path, enum uio_seg pathseg,
2224     char *buf, enum uio_seg bufseg, size_t count)
2225 {
2226     struct vnode *vp;
2227     struct iovec aiov;
2228     struct uio auio;
2229     struct nameidata nd;
2230     int error;
2231
2232     if (count > IOSIZE_MAX)
2233         return (EINVAL);
2234

```

```

2235     NDINIT_AT(&nd, LOOKUP, NOFOLLOW | LOCKSHARED | LOCKLEAF | AUDITVNODE1,
2236             pathseg, path, fd, td);
2237
2238     if ((error = namei(&nd)) != 0)
2239         return (error);
2240     NDFREE(&nd, NDF_ONLY_PNBUF);
2241     vp = nd.ni_vp;
2242 #ifdef MAC
2243     error = mac_vnode_check_readlink(td->td_ucred, vp);
2244     if (error != 0) {
2245         vput(vp);
2246         return (error);
2247     }
2248 #endif
2249     if (vp->v_type != VLNK && (vp->v_vflag & VV_READLINK) == 0)
2250         error = EINVAL;
2251     else {
2252         aiov.iov_base = buf;
2253         aiov.iov_len = count;
2254         auio.uio_iov = &aiov;
2255         auio.uio_iovcnt = 1;
2256         auio.uio_offset = 0;
2257         auio.uio_rw = UIO_READ;
2258         auio.uio_segflg = bufseg;
2259         auio.uio_td = td;
2260         auio.uio_resid = count;
2261         error = VOP_READLINK(vp, &auio, td->td_ucred);
2262         td->td_retval[0] = count - auio.uio_resid;
2263     }
2264     vput(vp);
2265     return (error);
2266 }
2267
2268 /*
2269  * Common implementation code for chflags() and fchflags().
2270  */
2271 static int
2272 setfflags(struct thread *td, struct vnode *vp, u_long flags)
2273 {
2274     struct mount *mp;
2275     struct vattr vattr;
2276     int error;
2277
2278     /* We can't support the value matching VNOVAL. */
2279     if (flags == VNOVAL)
2280         return (EOPNOTSUPP);
2281
2282     /*
2283      * Prevent non-root users from setting flags on devices. When
2284      * a device is reused, users can retain ownership of the device
2285      * if they are allowed to set flags and programs assume that
2286      * chown can't fail when done as root.
2287      */
2288     if (vp->v_type == VCHR || vp->v_type == VBLK) {
2289         error = priv_check(td, PRIV_VFS_CHFLAGS_DEV);
2290         if (error != 0)
2291             return (error);
2292     }
2293
2294     if ((error = vn_start_write(vp, &mp, V_WAIT | PCATCH)) != 0)
2295         return (error);
2296     VATTR_NULL(&vattr);
2297     vattr.va_flags = flags;
2298     vn_lock(vp, LK_EXCLUSIVE | LK_RETRY);
2299 #ifdef MAC
2300     error = mac_vnode_check_setflags(td->td_ucred, vp, vattr.va_flags);
2301     if (error == 0)
2302 #endif
2303         error = VOP_SETATTR(vp, &vattr, td->td_ucred);
2304     VOP_UNLOCK(vp, 0);
2305     vn_finished_write(mp);
2306     return (error);
2307 }
2308
2309 /*
2310  * Change flags of a file given a path name.
2311  */
2312 #ifdef _SYS_SYSPROTO_H_
2313 struct chflags_args {
2314     const char *path;
2315     u_long flags;
2316 };
2317 #endif
2318 int
2319 sys_chflags(struct thread *td, struct chflags_args *uap)
2320 {
2321
2322     return (kern_chflagsat(td, AT_FDCWD, uap->path, UIO_USERSPACE,
2323         uap->flags, 0));
2324 }
2325
2326 #ifdef _SYS_SYSPROTO_H_
2327 struct chflagsat_args {
2328     int fd;
2329     const char *path;
2330     u_long flags;
2331     int atflag;
2332 }
2333

```



```

2333 #endif
2334 int
2335 sys_chflagsat(struct thread *td, struct chflagsat_args *uap)
2336 {
2337     int fd = uap->fd;
2338     const char *path = uap->path;
2339     u_long flags = uap->flags;
2340     int atflag = uap->atflag;
2341
2342     if (atflag & ~AT_SYMLINK_NOFOLLOW)
2343         return (EINVAL);
2344
2345     return (kern_chflagsat(td, fd, path, UIO_USERSPACE, flags, atflag));
2346 }
2347
2348 /*
2349  * Same as chflags() but doesn't follow symlinks.
2350  */
2351 #ifndef _SYS_SYSPROTO_H_
2352 struct lchflags_args {
2353     const char *path;
2354     u_long flags;
2355 };
2356 #endif
2357 int
2358 sys_lchflags(struct thread *td, struct lchflags_args *uap)
2359 {
2360
2361     return (kern_chflagsat(td, AT_FDCWD, uap->path, UIO_USERSPACE,
2362         uap->flags, AT_SYMLINK_NOFOLLOW));
2363 }
2364
2365 static int
2366 kern_chflagsat(struct thread *td, int fd, const char *path,
2367     enum uio_seg pathseg, u_long flags, int atflag)
2368 {
2369     struct nameidata nd;
2370     cap_rights_t rights;
2371     int error, follow;
2372
2373     AUDIT_ARG_FFLAGS(flags);
2374     follow = (atflag & AT_SYMLINK_NOFOLLOW) ? NOFOLLOW : FOLLOW;
2375     NDINIT_ATRIGHTS(&nd, LOOKUP, follow | AUDITVNODE1, pathseg, path, fd,
2376         cap_rights_init(&rights, CAP_FCHFLAGS), td);
2377     if ((error = namei(&nd)) != 0)
2378         return (error);
2379     NDFREE(&nd, NDF_ONLY_PNBUF);
2380     error = setfflags(td, nd.ni_vp, flags);
2381     vrele(nd.ni_vp);
2382     return (error);
2383 }
2384
2385 /*
2386  * Change flags of a file given a file descriptor.
2387  */
2388 #ifndef _SYS_SYSPROTO_H_
2389 struct fchflags_args {
2390     int fd;
2391     u_long flags;
2392 };
2393 #endif
2394 int
2395 sys_fchflags(struct thread *td, struct fchflags_args *uap)
2396 {
2397     struct file *fp;
2398     cap_rights_t rights;
2399     int error;
2400
2401     AUDIT_ARG_FD(uap->fd);
2402     AUDIT_ARG_FFLAGS(uap->flags);
2403     error = getvnode(td, uap->fd, cap_rights_init(&rights, CAP_FCHFLAGS),
2404         &fp);
2405     if (error != 0)
2406         return (error);
2407 #ifdef AUDIT
2408     vn_lock(fp->vnode, LK_SHARED | LK_RETRY);
2409     AUDIT_ARG_VNODE1(fp->vnode);
2410     VOP_UNLOCK(fp->vnode, 0);
2411 #endif
2412     error = setfflags(td, fp->vnode, uap->flags);
2413     fdrop(fp, td);
2414     return (error);
2415 }
2416
2417 /*
2418  * Common implementation code for chmod(), lchmod() and fchmod().
2419  */
2420 int
2421 setfmode(struct thread *td, struct ucred *cred, struct vnode *vp, int mode)
2422 {
2423     struct mount *mp;
2424     struct vattr vattr;
2425     int error;
2426
2427     if ((error = vn_start_write(vp, &mp, V_WAIT | PCATCH)) != 0)
2428         return (error);
2429     vn_lock(vp, LK_EXCLUSIVE | LK_RETRY);
2430     VATTR_NULL(&vattr);

```

```

2431     vattr.va_mode = mode & ALLPERMS;
2432 #ifdef MAC
2433     error = mac_vnode_check_setmode(cred, vp, vattr.va_mode);
2434     if (error == 0)
2435 #endif
2436         error = VOP_SETATTR(vp, &vattr, cred);
2437     VOP_UNLOCK(vp, 0);
2438     vn_finished_write(mp);
2439     return (error);
2440 }
2441
2442 /*
2443  * Change mode of a file given path name.
2444  */
2445 #ifndef _SYS_SYSPROTO_H_
2446 struct chmod_args {
2447     char    *path;
2448     int     mode;
2449 };
2450 #endif
2451 int
2452 sys_chmod(struct thread *td, struct chmod_args *uap)
2453 {
2454     return (kern_fchmodat(td, AT_FDCWD, uap->path, UIO_USERSPACE,
2455         uap->mode, 0));
2456 }
2457
2458 #ifndef _SYS_SYSPROTO_H_
2459 struct fchmodat_args {
2460     int     dirfd;
2461     char    *path;
2462     mode_t  mode;
2463     int     flag;
2464 }
2465 #endif
2466 int
2467 sys_fchmodat(struct thread *td, struct fchmodat_args *uap)
2468 {
2469     int flag = uap->flag;
2470     int fd = uap->fd;
2471     char *path = uap->path;
2472     mode_t mode = uap->mode;
2473
2474     if (flag & ~AT_SYMLINK_NOFOLLOW)
2475         return (EINVAL);
2476
2477     return (kern_fchmodat(td, fd, path, UIO_USERSPACE, mode, flag));
2478 }
2479
2480 /*
2481  * Change mode of a file given path name (don't follow links.)
2482  */
2483 #ifndef _SYS_SYSPROTO_H_
2484 struct lchmod_args {
2485     char    *path;
2486     int     mode;
2487 };
2488 #endif
2489 int
2490 sys_lchmod(struct thread *td, struct lchmod_args *uap)
2491 {
2492     return (kern_fchmodat(td, AT_FDCWD, uap->path, UIO_USERSPACE,
2493         uap->mode, AT_SYMLINK_NOFOLLOW));
2494 }
2495
2496 int
2497 kern_fchmodat(struct thread *td, int fd, char *path, enum uio_seg pathseg,
2500     mode_t mode, int flag)
2501 {
2502     struct nameidata nd;
2503     cap_rights_t rights;
2504     int error, follow;
2505
2506     AUDIT_ARG_MODE(mode);
2507     follow = (flag & AT_SYMLINK_NOFOLLOW) ? NOFOLLOW : FOLLOW;
2508     NDINIT_ATRIGHTS(&nd, LOOKUP, follow | AUDITVNODE1, pathseg, path, fd,
2509         cap_rights_init(&rights, CAP_FCHMOD), td);
2510     if ((error = namei(&nd)) != 0)
2511         return (error);
2512     NDFREE(&nd, NDF_ONLY_PNBUF);
2513     error = setfmode(td, td->td_ucred, nd.ni_vp, mode);
2514     vrele(nd.ni_vp);
2515     return (error);
2516 }
2517
2518 /*
2519  * Change mode of a file given a file descriptor.
2520  */
2521 #ifndef _SYS_SYSPROTO_H_
2522 struct fchmod_args {
2523     int     fd;
2524     int     mode;
2525 };
2526 #endif
2527 int
2528 sys_fchmod(struct thread *td, struct fchmod_args *uap)

```

```

2529 {
2530     struct file *fp;
2531     cap_rights_t rights;
2532     int error;
2533
2534     AUDIT_ARG_FD(uap->fd);
2535     AUDIT_ARG_MODE(uap->mode);
2536
2537     error = fget(td, uap->fd, cap_rights_init(&rights, CAP_FCHMOD), &fp);
2538     if (error != 0)
2539         return (error);
2540     error = fo_chmod(fp, uap->mode, td->td_ucred, td);
2541     fdrop(fp, td);
2542     return (error);
2543 }
2544
2545 /*
2546  * Common implementation for chown(), lchown(), and fchown()
2547  */
2548 int
2549 setfown(struct thread *td, struct ucred *cred, struct vnode *vp, uid_t uid,
2550         gid_t gid)
2551 {
2552     struct mount *mp;
2553     struct vattr vattr;
2554     int error;
2555
2556     if ((error = vn_start_write(vp, &mp, V_WAIT | PCATCH)) != 0)
2557         return (error);
2558     vn_lock(vp, LK_EXCLUSIVE | LK_RETRY);
2559     VATTR_NULL(&vattr);
2560     vattr.va_uid = uid;
2561     vattr.va_gid = gid;
2562 #ifdef MAC
2563     error = mac_vnode_check_setowner(cred, vp, vattr.va_uid,
2564                                     vattr.va_gid);
2565     if (error == 0)
2566 #endif
2567         error = VOP_SETATTR(vp, &vattr, cred);
2568     VOP_UNLOCK(vp, 0);
2569     vn_finished_write(mp);
2570     return (error);
2571 }
2572
2573 /*
2574  * Set ownership given a path name.
2575  */
2576 #ifndef _SYS_SYSPROTO_H_
2577 struct chown_args {
2578     char *path;
2579     int uid;
2580     int gid;
2581 };
2582 #endif
2583 int
2584 sys_chown(struct thread *td, struct chown_args *uap)
2585 {
2586
2587     return (kern_fchownat(td, AT_FDCWD, uap->path, UIO_USERSPACE, uap->uid,
2588                          uap->gid, 0));
2589 }
2590
2591 #ifndef _SYS_SYSPROTO_H_
2592 struct fchownat_args {
2593     int fd;
2594     const char * path;
2595     uid_t uid;
2596     gid_t gid;
2597     int flag;
2598 };
2599 #endif
2600 int
2601 sys_fchownat(struct thread *td, struct fchownat_args *uap)
2602 {
2603     int flag;
2604
2605     flag = uap->flag;
2606     if (flag & ~AT_SYMLINK_NOFOLLOW)
2607         return (EINVAL);
2608
2609     return (kern_fchownat(td, uap->fd, uap->path, UIO_USERSPACE, uap->uid,
2610                          uap->gid, uap->flag));
2611 }
2612
2613 int
2614 kern_fchownat(struct thread *td, int fd, char *path, enum uio_seg pathseg,
2615               int uid, int gid, int flag)
2616 {
2617     struct nameidata nd;
2618     cap_rights_t rights;
2619     int error, follow;
2620
2621     AUDIT_ARG_OWNER(uid, gid);
2622     follow = (flag & AT_SYMLINK_NOFOLLOW) ? NOFOLLOW : FOLLOW;
2623     NDINIT_ATRIGHTS(&nd, LOOKUP, follow | AUDITVNODE1, pathseg, path, fd,
2624                   cap_rights_init(&rights, CAP_FCHOWN), td);
2625
2626     if ((error = namei(&nd)) != 0)

```

```

2627         return (error);
2628     NDFREE(&nd, NDF_ONLY_PNBUF);
2629     error = setfown(td, td->td_ucred, nd.ni_vp, uid, gid);
2630     vrele(nd.ni_vp);
2631     return (error);
2632 }
2633
2634 /*
2635  * Set ownership given a path name, do not cross symlinks.
2636  */
2637 #ifndef _SYS_SYSPROTO_H_
2638 struct lchown_args {
2639     char    *path;
2640     int     uid;
2641     int     gid;
2642 };
2643 #endif
2644 int
2645 sys_lchown(struct thread *td, struct lchown_args *uap)
2646 {
2647     return (kern_fchownat(td, AT_FDCWD, uap->path, UIO_USERSPACE,
2648         uap->uid, uap->gid, AT_SYMLINK_NOFOLLOW));
2649 }
2650
2651 /*
2652  * Set ownership given a file descriptor.
2653  */
2654 #ifndef _SYS_SYSPROTO_H_
2655 struct fchown_args {
2656     int     fd;
2657     int     uid;
2658     int     gid;
2659 };
2660 #endif
2661 int
2662 sys_fchown(struct thread *td, struct fchown_args *uap)
2663 {
2664     struct file *fp;
2665     cap_rights_t rights;
2666     int error;
2667
2668     AUDIT_ARG_FD(uap->fd);
2669     AUDIT_ARG_OWNER(uap->uid, uap->gid);
2670     error = fget(td, uap->fd, cap_rights_init(&rights, CAP_FCHOWN), &fp);
2671     if (error != 0)
2672         return (error);
2673     error = fo_chown(fp, uap->uid, uap->gid, td->td_ucred, td);
2674     fdrop(fp, td);
2675     return (error);
2676 }
2677
2678 /*
2679  * Common implementation code for utimes(), lutimes(), and futimes().
2680  */
2681 static int
2682 getutimes(const struct timeval *usrtpv, enum uio_seg tvpseg,
2683     struct timespec *tsp)
2684 {
2685     struct timeval tv[2];
2686     const struct timeval *tvp;
2687     int error;
2688
2689     if (usrtpv == NULL) {
2690         vfs_timestamp(&tsp[0]);
2691         tsp[1] = tsp[0];
2692     } else {
2693         if (tvpseg == UIO_SYSSPACE) {
2694             tvp = usrtpv;
2695         } else {
2696             if ((error = copyin(usrtpv, tv, sizeof(tv))) != 0)
2697                 return (error);
2698             tvp = tv;
2699         }
2700
2701         if (tvp[0].tv_usec < 0 || tvp[0].tv_usec >= 1000000 ||
2702             tvp[1].tv_usec < 0 || tvp[1].tv_usec >= 1000000)
2703             return (EINVAL);
2704         TIMEVAL_TO_TIMESPEC(&tvp[0], &tsp[0]);
2705         TIMEVAL_TO_TIMESPEC(&tvp[1], &tsp[1]);
2706     }
2707     return (0);
2708 }
2709
2710 /*
2711  * Common implementation code for futimens(), utimensat().
2712  */
2713 #define UTIMENS_NULL    0x1
2714 #define UTIMENS_EXIT    0x2
2715 static int
2716 getutimens(const struct timespec *usrtspec, enum uio_seg tspspec,
2717     struct timespec *tsp, int *retflags)
2718 {
2719     struct timespec tsnow;
2720     int error;
2721
2722     vfs_timestamp(&tsnow);
2723     *retflags = 0;
2724 }

```

```

2725     if (usrtsmp == NULL) {
2726         tsp[0] = tsnow;
2727         tsp[1] = tsnow;
2728         *retflags |= UTIMENS_NULL;
2729         return (0);
2730     }
2731     if (tspseg == UIO_SYSSPACE) {
2732         tsp[0] = usrtsmp[0];
2733         tsp[1] = usrtsmp[1];
2734     } else if ((error = copyin(usrtsmp, tsp, sizeof(*tsp) * 2)) != 0)
2735         return (error);
2736     if (tsp[0].tv_nsec == UTIME_OMIT && tsp[1].tv_nsec == UTIME_OMIT)
2737         *retflags |= UTIMENS_EXIT;
2738     if (tsp[0].tv_nsec == UTIME_NOW && tsp[1].tv_nsec == UTIME_NOW)
2739         *retflags |= UTIMENS_NULL;
2740     if (tsp[0].tv_nsec == UTIME_OMIT)
2741         tsp[0].tv_sec = VNOVAL;
2742     else if (tsp[0].tv_nsec == UTIME_NOW)
2743         tsp[0] = tsnow;
2744     else if (tsp[0].tv_nsec < 0 || tsp[0].tv_nsec >= 1000000000L)
2745         return (EINVAL);
2746     if (tsp[1].tv_nsec == UTIME_OMIT)
2747         tsp[1].tv_sec = VNOVAL;
2748     else if (tsp[1].tv_nsec == UTIME_NOW)
2749         tsp[1] = tsnow;
2750     else if (tsp[1].tv_nsec < 0 || tsp[1].tv_nsec >= 1000000000L)
2751         return (EINVAL);
2752
2753     return (0);
2754 }
2755
2756 /*
2757  * Common implementation code for utimes(), lutimes(), futimes(), futimens(),
2758  * and utimensat().
2759  */
2760 static int
2761 setutimes(struct thread *td, struct vnode *vp, const struct timespec *ts,
2762           int numtimes, int nullflag)
2763 {
2764     struct mount *mp;
2765     struct vattr vattr;
2766     int error, setbirthtime;
2767
2768     if ((error = vn_start_write(vp, &mp, V_WAIT | PCATCH)) != 0)
2769         return (error);
2770     vn_lock(vp, LK_EXCLUSIVE | LK_RETRY);
2771     setbirthtime = 0;
2772     if (numtimes < 3 && !VOP_GETATTR(vp, &vattr, td->td_ucred) &&
2773         timespeccmp(&ts[1], &vattr.va_birhtime, < ))
2774         setbirthtime = 1;
2775     VATTR_NULL(&vattr);
2776     vattr.va_atime = ts[0];
2777     vattr.va_mtime = ts[1];
2778     if (setbirthtime)
2779         vattr.va_birhtime = ts[1];
2780     if (numtimes > 2)
2781         vattr.va_birhtime = ts[2];
2782     if (nullflag)
2783         vattr.va_vflags |= VA_UTIMES_NULL;
2784 #ifdef MAC
2785     error = mac_vnode_check_setutimes(td->td_ucred, vp, vattr.va_atime,
2786         vattr.va_mtime);
2787 #endif
2788     if (error == 0)
2789         error = VOP_SETATTR(vp, &vattr, td->td_ucred);
2790     VOP_UNLOCK(vp, 0);
2791     vn_finished_write(mp);
2792     return (error);
2793 }
2794
2795 /*
2796  * Set the access and modification times of a file.
2797  */
2798 #ifndef _SYS_SYSPROTO_H_
2799 struct utimes_args {
2800     char *path;
2801     struct timeval *tptr;
2802 };
2803 #endif
2804 int
2805 sys_utimes(struct thread *td, struct utimes_args *uap)
2806 {
2807
2808     return (kern_utimesat(td, AT_FDCWD, uap->path, UIO_USERSPACE,
2809         uap->tptr, UIO_USERSPACE));
2810 }
2811
2812 #ifndef _SYS_SYSPROTO_H_
2813 struct futimesat_args {
2814     int fd;
2815     const char * path;
2816     const struct timeval * times;
2817 };
2818 #endif
2819 int
2820 sys_futimesat(struct thread *td, struct futimesat_args *uap)
2821 {
2822

```

```

2823         return (kern_utimesat(td, uap->fd, uap->path, UIO_USERSPACE,
2824             uap->times, UIO_USERSPACE));
2825     }
2826
2827     int
2828     kern_utimesat(struct thread *td, int fd, char *path, enum uio_seg pathseg,
2829         struct timeval *tptr, enum uio_seg tptrseg)
2830     {
2831         struct nameidata nd;
2832         struct timespec ts[2];
2833         cap_rights_t rights;
2834         int error;
2835
2836         if ((error = getutimes(tptr, tptrseg, ts)) != 0)
2837             return (error);
2838         NDINIT(&nd, LOOKUP, FOLLOW | AUDITVNODE1, pathseg, path, fd,
2839             cap_rights_init(&rights, CAP_FUTIMES), td);
2840
2841         if ((error = namei(&nd)) != 0)
2842             return (error);
2843         NDFREE(&nd, NDF_ONLY_PNBUF);
2844         error = setutimes(td, nd.ni_vp, ts, 2, tptr == NULL);
2845         vrele(nd.ni_vp);
2846         return (error);
2847     }
2848
2849     /*
2850     * Set the access and modification times of a file.
2851     */
2852     #ifndef _SYS_SYSPROTO_H_
2853     struct lutimes_args {
2854         char *path;
2855         struct timeval *tptr;
2856     };
2857     #endif
2858     int
2859     sys_lutimes(struct thread *td, struct lutimes_args *uap)
2860     {
2861
2862         return (kern_lutimes(td, uap->path, UIO_USERSPACE, uap->tptr,
2863             UIO_USERSPACE));
2864     }
2865
2866     int
2867     kern_lutimes(struct thread *td, char *path, enum uio_seg pathseg,
2868         struct timeval *tptr, enum uio_seg tptrseg)
2869     {
2870         struct timespec ts[2];
2871         struct nameidata nd;
2872         int error;
2873
2874         if ((error = getutimes(tptr, tptrseg, ts)) != 0)
2875             return (error);
2876         NDINIT(&nd, LOOKUP, NOFOLLOW | AUDITVNODE1, pathseg, path, td);
2877         if ((error = namei(&nd)) != 0)
2878             return (error);
2879         NDFREE(&nd, NDF_ONLY_PNBUF);
2880         error = setutimes(td, nd.ni_vp, ts, 2, tptr == NULL);
2881         vrele(nd.ni_vp);
2882         return (error);
2883     }
2884
2885     /*
2886     * Set the access and modification times of a file.
2887     */
2888     #ifndef _SYS_SYSPROTO_H_
2889     struct futimes_args {
2890         int fd;
2891         struct timeval *tptr;
2892     };
2893     #endif
2894     int
2895     sys_futimes(struct thread *td, struct futimes_args *uap)
2896     {
2897
2898         return (kern_futimes(td, uap->fd, uap->tptr, UIO_USERSPACE));
2899     }
2900
2901     int
2902     kern_futimes(struct thread *td, int fd, struct timeval *tptr,
2903         enum uio_seg tptrseg)
2904     {
2905         struct timespec ts[2];
2906         struct file *fp;
2907         cap_rights_t rights;
2908         int error;
2909
2910         AUDIT_ARG_FD(fd);
2911         error = getutimes(tptr, tptrseg, ts);
2912         if (error != 0)
2913             return (error);
2914         error = getvnode(td, fd, cap_rights_init(&rights, CAP_FUTIMES), &fp);
2915         if (error != 0)
2916             return (error);
2917     #ifdef AUDIT
2918         vn_lock(fp->f_vnode, LK_SHARED | LK_RETRY);
2919         AUDIT_ARG_VNODE1(fp->f_vnode);
2920         VOP_UNLOCK(fp->f_vnode, 0);

```

```

2921 #endif
2922     error = setutimes(td, fp->f_vnode, ts, 2, tptr == NULL);
2923     fdrop(fp, td);
2924     return (error);
2925 }
2926
2927 int
2928 sys_futimens(struct thread *td, struct futimens_args *uap)
2929 {
2930
2931     return (kern_futimens(td, uap->fd, uap->times, UIO_USERSPACE));
2932 }
2933
2934 int
2935 kern_futimens(struct thread *td, int fd, struct timespec *tptr,
2936               enum uio_seg tptrseg)
2937 {
2938     struct timespec ts[2];
2939     struct file *fp;
2940     cap_rights_t rights;
2941     int error, flags;
2942
2943     AUDIT_ARG_FD(fd);
2944     error = getutimens(tptr, tptrseg, ts, &flags);
2945     if (error != 0)
2946         return (error);
2947     if (flags & UTIMENS_EXIT)
2948         return (0);
2949     error = getvnode(td, fd, cap_rights_init(&rights, CAP_FUTIMES), &fp);
2950     if (error != 0)
2951         return (error);
2952 #ifdef AUDIT
2953     vn_lock(fp->f_vnode, LK_SHARED | LK_RETRY);
2954     AUDIT_ARG_VNODE1(fp->f_vnode);
2955     VOP_UNLOCK(fp->f_vnode, 0);
2956 #endif
2957     error = setutimes(td, fp->f_vnode, ts, 2, flags & UTIMENS_NULL);
2958     fdrop(fp, td);
2959     return (error);
2960 }
2961
2962 int
2963 sys_utimensat(struct thread *td, struct utimensat_args *uap)
2964 {
2965
2966     return (kern_utimensat(td, uap->fd, uap->path, UIO_USERSPACE,
2967                           uap->times, UIO_USERSPACE, uap->flag));
2968 }
2969
2970 int
2971 kern_utimensat(struct thread *td, int fd, char *path, enum uio_seg pathseg,
2972                struct timespec *tptr, enum uio_seg tptrseg, int flag)
2973 {
2974     struct nameidata nd;
2975     struct timespec ts[2];
2976     cap_rights_t rights;
2977     int error, flags;
2978
2979     if (flag & ~AT_SYMLINK_NOFOLLOW)
2980         return (EINVAL);
2981
2982     if ((error = getutimens(tptr, tptrseg, ts, &flags)) != 0)
2983         return (error);
2984     NDINIT_ATRIGHTS(&nd, LOOKUP, ((flag & AT_SYMLINK_NOFOLLOW) ? NOFOLLOW :
2985                                   FOLLOW) | AUDITVNODE1, pathseg, path, fd,
2986                   cap_rights_init(&rights, CAP_FUTIMES), td);
2987     if ((error = namei(&nd)) != 0)
2988         return (error);
2989     /*
2990      * We are allowed to call namei() regardless of 2xUTIME_OMIT.
2991      * POSIX states:
2992      * "If both tv_nsec fields are UTIME_OMIT... EACCESS may be detected."
2993      * "Search permission is denied by a component of the path prefix."
2994      */
2995     NDFREE(&nd, NDF_ONLY_PNBUF);
2996     if ((flags & UTIMENS_EXIT) == 0)
2997         error = setutimes(td, nd.ni_vp, ts, 2, flags & UTIMENS_NULL);
2998     vrele(nd.ni_vp);
2999     return (error);
3000 }
3001
3002 /*
3003  * Truncate a file given its path name.
3004  */
3005 #ifndef _SYS_SYSPROTO_H_
3006 struct truncate_args {
3007     char    *path;
3008     int     pad;
3009     off_t   length;
3010 };
3011 #endif
3012 int
3013 sys_truncate(struct thread *td, struct truncate_args *uap)
3014 {
3015
3016     return (kern_truncate(td, uap->path, UIO_USERSPACE, uap->length));
3017 }
3018

```

```

3019 int
3020 kern_truncate(struct thread *td, char *path, enum uio_seg pathseg, off_t length)
3021 {
3022     struct mount *mp;
3023     struct vnode *vp;
3024     void *rl_cookie;
3025     struct vattr vattr;
3026     struct nameidata nd;
3027     int error;
3028
3029     if (length < 0)
3030         return(EINVAL);
3031     NDINIT(&nd, LOOKUP, FOLLOW | AUDITVNODE1, pathseg, path, td);
3032     if ((error = namei(&nd)) != 0)
3033         return (error);
3034     vp = nd.ni_vp;
3035     rl_cookie = vn_rangelock_wlock(vp, 0, OFF_MAX);
3036     if ((error = vn_start_write(vp, &mp, V_WAIT | PCATCH)) != 0) {
3037         vn_rangelock_unlock(vp, rl_cookie);
3038         vrele(vp);
3039         return (error);
3040     }
3041     NDFREE(&nd, NDF_ONLY_PNBUF);
3042     vn_lock(vp, LK_EXCLUSIVE | LK_RETRY);
3043     if (vp->v_type == VDIR)
3044         error = EISDIR;
3045 #ifdef MAC
3046     else if ((error = mac_vnode_check_write(td->td_ucred, NOCRED, vp))) {
3047     }
3048 #endif
3049     else if ((error = vn_writechk(vp)) == 0 &&
3050             (error = VOP_ACCESS(vp, VWRITE, td->td_ucred, td)) == 0) {
3051         VATTR_NULL(&vattr);
3052         vattr.va_size = length;
3053         error = VOP_SETATTR(vp, &vattr, td->td_ucred);
3054     }
3055     VOP_UNLOCK(vp, 0);
3056     vn_finished_write(mp);
3057     vn_rangelock_unlock(vp, rl_cookie);
3058     vrele(vp);
3059     return (error);
3060 }
3061
3062 #if defined(COMPAT_43)
3063 /*
3064  * Truncate a file given its path name.
3065  */
3066 #ifndef _SYS_SYSPROTO_H_
3067 struct otruncate_args {
3068     char *path;
3069     long length;
3070 };
3071 #endif
3072 int
3073 otruncate(struct thread *td, struct otruncate_args *uap)
3074 {
3075
3076     return (kern_truncate(td, uap->path, UIO_USERSPACE, uap->length));
3077 }
3078 #endif /* COMPAT_43 */
3079
3080 #if defined(COMPAT_FREEBSD6)
3081 /* Versions with the pad argument */
3082 int
3083 freebsd6_truncate(struct thread *td, struct freebsd6_truncate_args *uap)
3084 {
3085
3086     return (kern_truncate(td, uap->path, UIO_USERSPACE, uap->length));
3087 }
3088
3089 int
3090 freebsd6_ftruncate(struct thread *td, struct freebsd6_ftruncate_args *uap)
3091 {
3092
3093     return (kern_ftruncate(td, uap->fd, uap->length));
3094 }
3095 #endif
3096
3097 int
3098 kern_fsync(struct thread *td, int fd, bool fullsync)
3099 {
3100     struct vnode *vp;
3101     struct mount *mp;
3102     struct file *fp;
3103     cap_rights_t rights;
3104     int error, lock_flags;
3105
3106     AUDIT_ARG_FD(fd);
3107     error = getvnode(td, fd, cap_rights_init(&rights, CAP_FSYNC), &fp);
3108     if (error != 0)
3109         return (error);
3110     vp = fp->v_vnode;
3111 #if 0
3112     if (!fullsync)
3113         /* XXXKIB: compete outstanding aio writes */;
3114 #endif
3115     error = vn_start_write(vp, &mp, V_WAIT | PCATCH);
3116     if (error != 0)

```



```

3117         goto drop;
3118     if (MNT_SHARED_WRITES(mp) ||
3119         ((mp == NULL) && MNT_SHARED_WRITES(vp->v_mount))) {
3120         lock_flags = LK_SHARED;
3121     } else {
3122         lock_flags = LK_EXCLUSIVE;
3123     }
3124     vn_lock(vp, lock_flags | LK_RETRY);
3125     AUDIT_ARG_VNODE1(vp);
3126     if (vp->v_object != NULL) {
3127         VM_OBJECT_WLOCK(vp->v_object);
3128         vm_object_page_clean(vp->v_object, 0, 0, 0);
3129         VM_OBJECT_WUNLOCK(vp->v_object);
3130     }
3131     error = fullsync ? VOP_FSYNC(vp, MNT_WAIT, td) : VOP_FDATASYNC(vp, td);
3132     VOP_UNLOCK(vp, 0);
3133     vn_finished_write(mp);
3134 drop:
3135     fdrop(fp, td);
3136     return (error);
3137 }
3138
3139 /*
3140  * Sync an open file.
3141  */
3142 #ifndef _SYS_SYSPROTO_H_
3143 struct fsync_args {
3144     int fd;
3145 };
3146 #endif
3147 int
3148 sys_fsync(struct thread *td, struct fsync_args *uap)
3149 {
3150
3151     return (kern_fsync(td, uap->fd, true));
3152 }
3153
3154 int
3155 sys_fdatasync(struct thread *td, struct fdatasync_args *uap)
3156 {
3157
3158     return (kern_fsync(td, uap->fd, false));
3159 }
3160
3161 /*
3162  * Rename files. Source and destination must either both be directories, or
3163  * both not be directories. If target is a directory, it must be empty.
3164  */
3165 #ifndef _SYS_SYSPROTO_H_
3166 struct rename_args {
3167     char *from;
3168     char *to;
3169 };
3170 #endif
3171 int
3172 sys_rename(struct thread *td, struct rename_args *uap)
3173 {
3174
3175     return (kern_renameat(td, AT_FDCWD, uap->from, AT_FDCWD,
3176         uap->to, UIO_USERSPACE));
3177 }
3178
3179 #ifndef _SYS_SYSPROTO_H_
3180 struct renameat_args {
3181     int oldfd;
3182     char *old;
3183     int newfd;
3184     char *new;
3185 };
3186 #endif
3187 int
3188 sys_renameat(struct thread *td, struct renameat_args *uap)
3189 {
3190
3191     return (kern_renameat(td, uap->oldfd, uap->old, uap->newfd, uap->new,
3192         UIO_USERSPACE));
3193 }
3194
3195 int
3196 kern_renameat(struct thread *td, int oldfd, char *old, int newfd, char *new,
3197     enum uio_seg pathseg)
3198 {
3199     struct mount *mp = NULL;
3200     struct vnode *tvp, *fvp, *tdvp;
3201     struct nameidata fromnd, tond;
3202     cap_rights_t rights;
3203     int error;
3204
3205     again:
3206     bwillwrite();
3207 #ifdef MAC
3208     NDINIT_ATRIGHTS(&fromnd, DELETE, LOCKPARENT | LOCKLEAF | SAVESTART |
3209         AUDITVNODE1, pathseg, old, oldfd,
3210         cap_rights_init(&rights, CAP_RENAMEAT_SOURCE), td);
3211 #else
3212     NDINIT_ATRIGHTS(&fromnd, DELETE, WANTPARENT | SAVESTART | AUDITVNODE1,
3213         pathseg, old, oldfd,
3214         cap_rights_init(&rights, CAP_RENAMEAT_SOURCE), td);

```

```

3215 #endif
3216
3217     if ((error = namei(&fromnd)) != 0)
3218         return (error);
3219 #ifdef MAC
3220     error = mac_vnode_check_rename_from(td->td_ucred, fromnd.ni_dvp,
3221         fromnd.ni_vp, &fromnd.ni_cnd);
3222     VOP_UNLOCK(fromnd.ni_dvp, 0);
3223     if (fromnd.ni_dvp != fromnd.ni_vp)
3224         VOP_UNLOCK(fromnd.ni_vp, 0);
3225 #endif
3226     fvp = fromnd.ni_vp;
3227     NDINIT_ATRIGHTS(&tond, RENAME, LOCKPARENT | LOCKLEAF | NOCACHE |
3228         SAVESTART | AUDITVNODE2, pathseg, new, newfd,
3229         cap_rights_init(&rights, CAP_RENAMEAT_TARGET), td);
3230     if (fromnd.ni_vp->v_type == VDIR)
3231         tond.ni_cnd.cn_flags |= WILLBEDIR;
3232     if ((error = namei(&tond)) != 0) {
3233         /* Translate error code for rename("dir1", "dir2/."). */
3234         if (error == EISDIR && fvp->v_type == VDIR)
3235             error = EINVAL;
3236         NDFREE(&fromnd, NDF_ONLY_PNBUF);
3237         vrele(fromnd.ni_dvp);
3238         vrele(fvp);
3239         goto out1;
3240     }
3241     tdvp = tond.ni_dvp;
3242     tvp = tond.ni_vp;
3243     error = vn_start_write(fvp, &mp, V_NOWAIT);
3244     if (error != 0) {
3245         NDFREE(&fromnd, NDF_ONLY_PNBUF);
3246         NDFREE(&tond, NDF_ONLY_PNBUF);
3247         if (tvp != NULL)
3248             vput(tvp);
3249         if (tdvp == tvp)
3250             vrele(tdvp);
3251         else
3252             vput(tdvp);
3253         vrele(fromnd.ni_dvp);
3254         vrele(fvp);
3255         vrele(tond.ni_startdir);
3256         if (fromnd.ni_startdir != NULL)
3257             vrele(fromnd.ni_startdir);
3258         error = vn_start_write(NULL, &mp, V_XSLEEP | PCATCH);
3259         if (error != 0)
3260             return (error);
3261         goto again;
3262     }
3263     if (tvp != NULL) {
3264         if (fvp->v_type == VDIR && tvp->v_type != VDIR) {
3265             error = ENOTDIR;
3266             goto out;
3267         } else if (fvp->v_type != VDIR && tvp->v_type == VDIR) {
3268             error = EISDIR;
3269             goto out;
3270         }
3271     }
3272 #ifdef CAPABILITIES
3273     if (newfd != AT_FDCWD) {
3274         /*
3275          * If the target already exists we require CAP_UNLINKAT
3276          * from 'newfd'.
3277          */
3278         error = cap_check(&tond.ni_filecaps.fc_rights,
3279             cap_rights_init(&rights, CAP_UNLINKAT));
3280         if (error != 0)
3281             goto out;
3282     }
3283 #endif
3284     if (fvp == tdvp) {
3285         error = EINVAL;
3286         goto out;
3287     }
3288     /*
3289     * If the source is the same as the destination (that is, if they
3290     * are links to the same vnode), then there is nothing to do.
3291     */
3292     if (fvp == tvp)
3293         error = -1;
3294 #ifdef MAC
3295     else
3296         error = mac_vnode_check_rename_to(td->td_ucred, tdvp,
3297             tond.ni_vp, fromnd.ni_dvp == tdvp, &tond.ni_cnd);
3298 #endif
3299 out:
3300     if (error == 0) {
3301         error = VOP_RENAME(fromnd.ni_dvp, fromnd.ni_vp, &fromnd.ni_cnd,
3302             tond.ni_dvp, tond.ni_vp, &tond.ni_cnd);
3303         NDFREE(&fromnd, NDF_ONLY_PNBUF);
3304         NDFREE(&tond, NDF_ONLY_PNBUF);
3305     } else {
3306         NDFREE(&fromnd, NDF_ONLY_PNBUF);
3307         NDFREE(&tond, NDF_ONLY_PNBUF);
3308         if (tvp != NULL)
3309             vput(tvp);
3310         if (tdvp == tvp)
3311             vrele(tdvp);
3312         else

```

```

3313         vput(tdvp);
3314         vrel(fromnd.ni_dvp);
3315         vrel(fvp);
3316     }
3317     vrel(tond.ni_startdir);
3318     vn_finished_write(mp);
3319 out1:
3320     if (fromnd.ni_startdir)
3321         vrel(fromnd.ni_startdir);
3322     if (error == -1)
3323         return (0);
3324     return (error);
3325 }
3326
3327 /*
3328  * Make a directory file.
3329  */
3330 #ifndef _SYS_SYSPROTO_H_
3331 struct mkdir_args {
3332     char *path;
3333     int mode;
3334 };
3335 #endif
3336 int
3337 sys_mkdir(struct thread *td, struct mkdir_args *uap)
3338 {
3339
3340     return (kern_mkdirat(td, AT_FDCWD, uap->path, UIO_USERSPACE,
3341         uap->mode));
3342 }
3343
3344 #ifndef _SYS_SYSPROTO_H_
3345 struct mkdirat_args {
3346     int fd;
3347     char *path;
3348     mode_t mode;
3349 };
3350 #endif
3351 int
3352 sys_mkdirat(struct thread *td, struct mkdirat_args *uap)
3353 {
3354
3355     return (kern_mkdirat(td, uap->fd, uap->path, UIO_USERSPACE, uap->mode));
3356 }
3357
3358 int
3359 kern_mkdirat(struct thread *td, int fd, char *path, enum uio_seg segflg,
3360     int mode)
3361 {
3362     struct mount *mp;
3363     struct vnode *vp;
3364     struct vattr vattr;
3365     struct nameidata nd;
3366     cap_rights_t rights;
3367     int error;
3368
3369     AUDIT_ARG_MODE(mode);
3370 restart:
3371     bwrite();
3372     NDINIT_ATRIGHTS(&nd, CREATE, LOCKPARENT | SAVENAME | AUDITVNODE1 |
3373         NOCACHE, segflg, path, fd, cap_rights_init(&rights, CAP_MKDIRAT),
3374         td);
3375     nd.ni_cnd.cn_flags |= WILLBEDIR;
3376     if ((error = namei(&nd)) != 0)
3377         return (error);
3378     vp = nd.ni_vp;
3379     if (vp != NULL) {
3380         NDFREE(&nd, NDF_ONLY_PNBUF);
3381         /*
3382          * XXX namei called with LOCKPARENT but not LOCKLEAF has
3383          * the strange behaviour of leaving the vnode unlocked
3384          * if the target is the same vnode as the parent.
3385          */
3386         if (vp == nd.ni_dvp)
3387             vrel(nd.ni_dvp);
3388         else
3389             vput(nd.ni_dvp);
3390         vrel(vp);
3391         return (EEXIST);
3392     }
3393     if (vn_start_write(nd.ni_dvp, &mp, V_NOWAIT) != 0) {
3394         NDFREE(&nd, NDF_ONLY_PNBUF);
3395         vput(nd.ni_dvp);
3396         if ((error = vn_start_write(NULL, &mp, V_XSLEEP | PCATCH)) != 0)
3397             return (error);
3398         goto restart;
3399     }
3400     VATTR_NULL(&vattr);
3401     vattr.va_type = VDIR;
3402     vattr.va_mode = (mode & ACCESSPERMS) &~ td->td_proc->p_fd->fd_cmask;
3403 #ifdef MAC
3404     error = mac_vnode_check_create(td->td_ucred, nd.ni_dvp, &nd.ni_cnd,
3405         &vattr);
3406     if (error != 0)
3407         goto out;
3408 #endif
3409     error = VOP_MKDIR(nd.ni_dvp, &nd.ni_vp, &nd.ni_cnd, &vattr);
3410 #ifdef MAC

```

```

3411 out:
3412 #endif
3413     NDFREE(&nd, NDF_ONLY_PNBUF);
3414     vput(nd.ni_dvp);
3415     if (error == 0)
3416         vput(nd.ni_vp);
3417     vn_finished_write(mp);
3418     return (error);
3419 }
3420
3421 /*
3422  * Remove a directory file.
3423  */
3424 #ifndef _SYS_SYSPROTO_H_
3425 struct rmdir_args {
3426     char *path;
3427 };
3428 #endif
3429 int
3430 sys_rmdir(struct thread *td, struct rmdir_args *uap)
3431 {
3432
3433     return (kern_rmdirat(td, AT_FDCWD, uap->path, UIO_USERSPACE));
3434 }
3435
3436 int
3437 kern_rmdirat(struct thread *td, int fd, char *path, enum uio_seg pathseg)
3438 {
3439     struct mount *mp;
3440     struct vnode *vp;
3441     struct nameidata nd;
3442     cap_rights_t rights;
3443     int error;
3444
3445 restart:
3446     bwrite();
3447     NDINIT_ATRIGHTS(&nd, DELETE, LOCKPARENT | LOCKLEAF | AUDITVNODE1,
3448         pathseg, path, fd, cap_rights_init(&rights, CAP_UNLINKAT), td);
3449     if ((error = namei(&nd)) != 0)
3450         return (error);
3451     vp = nd.ni_vp;
3452     if (vp->v_type != VDIR) {
3453         error = ENOTDIR;
3454         goto out;
3455     }
3456     /*
3457      * No rmdir "." please.
3458      */
3459     if (nd.ni_dvp == vp) {
3460         error = EINVAL;
3461         goto out;
3462     }
3463     /*
3464      * The root of a mounted filesystem cannot be deleted.
3465      */
3466     if (vp->v_vflag & VV_ROOT) {
3467         error = EBUSY;
3468         goto out;
3469     }
3470 #ifdef MAC
3471     error = mac_vnode_check_unlink(td->td_ucred, nd.ni_dvp, vp,
3472         &nd.ni_cnd);
3473     if (error != 0)
3474         goto out;
3475 #endif
3476     if (vn_start_write(nd.ni_dvp, &mp, V_NOWAIT) != 0) {
3477         NDFREE(&nd, NDF_ONLY_PNBUF);
3478         vput(vp);
3479         if (nd.ni_dvp == vp)
3480             vrele(nd.ni_dvp);
3481         else
3482             vput(nd.ni_dvp);
3483         if ((error = vn_start_write(NULL, &mp, V_XSLEEP | PCATCH)) != 0)
3484             return (error);
3485         goto restart;
3486     }
3487     vfs_notify_upper(vp, VFS_NOTIFY_UPPER_UNLINK);
3488     error = VOP_RMDIR(nd.ni_dvp, nd.ni_vp, &nd.ni_cnd);
3489     vn_finished_write(mp);
3490 out:
3491     NDFREE(&nd, NDF_ONLY_PNBUF);
3492     vput(vp);
3493     if (nd.ni_dvp == vp)
3494         vrele(nd.ni_dvp);
3495     else
3496         vput(nd.ni_dvp);
3497     return (error);
3498 }
3499
3500 #ifdef COMPAT_43
3501 /*
3502  * Read a block of directory entries in a filesystem independent format.
3503  */
3504 #ifndef _SYS_SYSPROTO_H_
3505 struct ogetdirentargs {
3506     int fd;
3507     char *buf;
3508     u_int count;

```

```

3509         long *basep;
3510     };
3511 #endif
3512 int
3513 ogetdiretries(struct thread *td, struct ogetdiretries_args *uap)
3514 {
3515     long loff;
3516     int error;
3517
3518     error = kern_ogetdiretries(td, uap, &loff);
3519     if (error == 0)
3520         error = copyout(&loff, uap->basep, sizeof(long));
3521     return (error);
3522 }
3523
3524 int
3525 kern_ogetdiretries(struct thread *td, struct ogetdiretries_args *uap,
3526     long *ploff)
3527 {
3528     struct vnode *vp;
3529     struct file *fp;
3530     struct uio auio, kuio;
3531     struct iovec aiov, kio;
3532     struct dirent *dp, *edp;
3533     cap_rights_t rights;
3534     caddr_t dirbuf;
3535     int error, eofflag, readcnt;
3536     long loff;
3537     off_t foffset;
3538
3539     /* XXX arbitrary sanity limit on 'count'. */
3540     if (uap->count > 64 * 1024)
3541         return (EINVAL);
3542     error = getvnode(td, uap->fd, cap_rights_init(&rights, CAP_READ), &fp);
3543     if (error != 0)
3544         return (error);
3545     if ((fp->f_flag & FREAD) == 0) {
3546         fdrop(fp, td);
3547         return (EBADF);
3548     }
3549     vp = fp->v_vnode;
3550     foffset = foffset_lock(fp, 0);
3551
3552     unionread:
3553     if (vp->v_type != VDIR) {
3554         foffset_unlock(fp, foffset, 0);
3555         fdrop(fp, td);
3556         return (EINVAL);
3557     }
3558     aiov.iov_base = uap->buf;
3559     aiov.iov_len = uap->count;
3560     auio.uio_iov = &aio;
3561     auio.uio_iovcnt = 1;
3562     auio.uio_rw = UIO_READ;
3563     auio.uio_segflg = UIO_USERSPACE;
3564     auio.uio_td = td;
3565     auio.uio_resid = uap->count;
3566     vn_lock(vp, LK_SHARED | LK_RETRY);
3567     loff = auio.uio_offset + foffset;
3568 #ifdef MAC
3569     error = mac_vnode_check_readdir(td->td_ucred, vp);
3570     if (error != 0) {
3571         VOP_UNLOCK(vp, 0);
3572         foffset_unlock(fp, foffset, FOF_NOUPDATE);
3573         fdrop(fp, td);
3574         return (error);
3575     }
3576 #endif
3577 # if (BYTE_ORDER != LITTLE_ENDIAN)
3578     if (vp->v_mount->mnt_maxsymlinklen <= 0) {
3579         error = VOP_READDIR(vp, &auio, fp->f_cred, &eofflag,
3580             NULL, NULL);
3581         foffset = auio.uio_offset;
3582     } else
3583 # endif
3584 {
3585     kuio = auio;
3586     kuio.uio_iov = &kio;
3587     kuio.uio_segflg = UIO_SYSSPACE;
3588     kio.iov_len = uap->count;
3589     dirbuf = malloc(uap->count, M_TEMP, M_WAITOK);
3590     kio.iov_base = dirbuf;
3591     error = VOP_READDIR(vp, &kuio, fp->f_cred, &eofflag,
3592         NULL, NULL);
3593     foffset = kuio.uio_offset;
3594     if (error == 0) {
3595         readcnt = uap->count - kuio.uio_resid;
3596         edp = (struct dirent *)&dirbuf[readcnt];
3597         for (dp = (struct dirent *)&dirbuf; dp < edp; ) {
3598             # if (BYTE_ORDER == LITTLE_ENDIAN)
3599                 /*
3600                  * The expected low byte of
3601                  * dp->d_namlen is our dp->d_type.
3602                  * The high MBZ byte of dp->d_namlen
3603                  * is our dp->d_namlen.
3604                  */
3605                 dp->d_type = dp->d_namlen;
3606                 dp->d_namlen = 0;
3607             # else

```

```

3607             /*
3608              * The dp->d_type is the high byte
3609              * of the expected dp->d_namlen,
3610              * so must be zero'ed.
3611              */
3612             dp->d_type = 0;
3613 #endif
3614             if (dp->d_reclen > 0) {
3615                 dp = (struct dirent *)
3616                     ((char *)dp + dp->d_reclen);
3617             } else {
3618                 error = EIO;
3619                 break;
3620             }
3621         }
3622         if (dp >= edp)
3623             error = uiomove(dirbuf, readcnt, &auio);
3624     }
3625     free(dirbuf, M_TEMP);
3626 }
3627 if (error != 0) {
3628     VOP_UNLOCK(vp, 0);
3629     foffset_unlock(fp, foffset, 0);
3630     fdrop(fp, td);
3631     return (error);
3632 }
3633 if (uap->count == auio.uio_resid &&
3634     (vp->v_vflag & VV_ROOT) &&
3635     (vp->v_mount->mnt_flag & MNT_UNION)) {
3636     struct vnode *tvp = vp;
3637     vp = vp->v_mount->mnt_vnodecovered;
3638     VREF(vp);
3639     fp->f_vnode = vp;
3640     fp->f_data = vp;
3641     foffset = 0;
3642     vput(tvp);
3643     goto unionread;
3644 }
3645 VOP_UNLOCK(vp, 0);
3646 foffset_unlock(fp, foffset, 0);
3647 fdrop(fp, td);
3648 td->td_retval[0] = uap->count - auio.uio_resid;
3649 if (error == 0)
3650     *ploff = loff;
3651     return (error);
3652 }
3653 #endif /* COMPAT_43 */
3654
3655 /*
3656  * Read a block of directory entries in a filesystem independent format.
3657  */
3658 #ifndef _SYS_SYSPROTO_H_
3659 struct getdirentries_args {
3660     int fd;
3661     char *buf;
3662     u_int count;
3663     long *basep;
3664 };
3665 #endif
3666 int
3667 sys_getdirentries(struct thread *td, struct getdirentries_args *uap)
3668 {
3669     long base;
3670     int error;
3671
3672     error = kern_getdirentries(td, uap->fd, uap->buf, uap->count, &base,
3673         NULL, UIO_USERSPACE);
3674     if (error != 0)
3675         return (error);
3676     if (uap->basep != NULL)
3677         error = copyout(&base, uap->basep, sizeof(long));
3678     return (error);
3679 }
3680
3681 int
3682 kern_getdirentries(struct thread *td, int fd, char *buf, u_int count,
3683     long *basep, ssize_t *residp, enum uio_seg bufseg)
3684 {
3685     struct vnode *vp;
3686     struct file *fp;
3687     struct uio auio;
3688     struct iovec aio;
3689     cap_rights_t rights;
3690     long loff;
3691     int error, eofflag;
3692     off_t foffset;
3693
3694     AUDIT_ARG_FD(fd);
3695     if (count > IOSIZE_MAX)
3696         return (EINVAL);
3697     auio.uio_resid = count;
3698     error = getvnode(td, fd, cap_rights_init(&rights, CAP_READ), &fp);
3699     if (error != 0)
3700         return (error);
3701     if ((fp->f_flag & FREAD) == 0) {
3702         fdrop(fp, td);
3703         return (EBADF);
3704     }

```

```

3705     vp = fp->f_vnode;
3706     foffset = foffset_lock(fp, 0);
3707 unionread:
3708     if (vp->v_type != VDIR) {
3709         error = EINVAL;
3710         goto fail;
3711     }
3712     aiov.iiov_base = buf;
3713     aiov.iiov_len = count;
3714     aiov.uio_iiov = &aiov;
3715     aiov.uio_iovcnt = 1;
3716     aiov.uio_rw = UIO_READ;
3717     aiov.uio_segflg = bufseg;
3718     aiov.uio_td = td;
3719     vn_lock(vp, LK_SHARED | LK_RETRY);
3720     AUDIT_ARG_VNODE1(vp);
3721     loff = aiov.uio_offset = foffset;
3722 #ifdef MAC
3723     error = mac_vnode_check_readdir(td->td_ucred, vp);
3724     if (error == 0)
3725 #endif
3726         error = VOP_READDIR(vp, &aiov, fp->f_cred, &eofflag, NULL,
3727             NULL);
3728     foffset = aiov.uio_offset;
3729     if (error != 0) {
3730         VOP_UNLOCK(vp, 0);
3731         goto fail;
3732     }
3733     if (count == aiov.uio_resid &&
3734         (vp->v_vflag & VV_ROOT) &&
3735         (vp->v_mount->mnt_flag & MNT_UNION)) {
3736         struct vnode *tvp = vp;
3737
3738         vp = vp->v_mount->mnt_vnodecovered;
3739         VREF(vp);
3740         fp->f_vnode = vp;
3741         fp->f_data = vp;
3742         foffset = 0;
3743         vput(tvp);
3744         goto unionread;
3745     }
3746     VOP_UNLOCK(vp, 0);
3747     *basep = loff;
3748     if (residp != NULL)
3749         *residp = aiov.uio_resid;
3750     td->td_retval[0] = count - aiov.uio_resid;
3751 fail:
3752     foffset_unlock(fp, foffset, 0);
3753     fdrop(fp, td);
3754     return (error);
3755 }
3756
3757 #ifndef _SYS_SYSPROTO_H_
3758 struct getdents_args {
3759     int fd;
3760     char *buf;
3761     size_t count;
3762 };
3763 #endif
3764 int
3765 sys_getdents(struct thread *td, struct getdents_args *uap)
3766 {
3767     struct getdirentargs ap;
3768
3769     ap.fd = uap->fd;
3770     ap.buf = uap->buf;
3771     ap.count = uap->count;
3772     ap.basep = NULL;
3773     return (sys_getdirentargs(td, &ap));
3774 }
3775
3776 /*
3777  * Set the mode mask for creation of filesystem nodes.
3778  */
3779 #ifndef _SYS_SYSPROTO_H_
3780 struct umask_args {
3781     int newmask;
3782 };
3783 #endif
3784 int
3785 sys_umask(struct thread *td, struct umask_args *uap)
3786 {
3787     struct filedesc *fdp;
3788
3789     fdp = td->td_proc->p_fdp;
3790     FILEDESC_XLOCK(fdp);
3791     td->td_retval[0] = fdp->fd_cmask;
3792     fdp->fd_cmask = uap->newmask & ALLPERMS;
3793     FILEDESC_XUNLOCK(fdp);
3794     return (0);
3795 }
3796
3797 /*
3798  * Void all references to file by ripping underlying filesystem away from
3799  * vnode.
3800  */
3801 #ifndef _SYS_SYSPROTO_H_
3802 struct revoke_args {

```

```

3803     char    *path;
3804 };
3805 #endif
3806 int
3807 sys_revoke(struct thread *td, struct revoke_args *uap)
3808 {
3809     struct vnode *vp;
3810     struct vattr vattr;
3811     struct nameidata nd;
3812     int error;
3813
3814     NDINIT(&nd, LOOKUP, FOLLOW | LOCKLEAF | AUDITVNODE1, UIO_USERSPACE,
3815         uap->path, td);
3816     if ((error = namei(&nd)) != 0)
3817         return (error);
3818     vp = nd.ni_vp;
3819     NDFREE(&nd, NDF_ONLY_PNBUF);
3820     if (vp->v_type != VCHR || vp->v_rdev == NULL) {
3821         error = EINVAL;
3822         goto out;
3823     }
3824 #ifdef MAC
3825     error = mac_vnode_check_revoke(td->td_ucred, vp);
3826     if (error != 0)
3827         goto out;
3828 #endif
3829     error = VOP_GETATTR(vp, &vattr, td->td_ucred);
3830     if (error != 0)
3831         goto out;
3832     if (td->td_ucred->cr_uid != vattr.va_uid) {
3833         error = priv_check(td, PRIV_VFS_ADMIN);
3834         if (error != 0)
3835             goto out;
3836     }
3837     if (vcount(vp) > 1)
3838         VOP_REVOKE(vp, REVOKEALL);
3839 out:
3840     vput(vp);
3841     return (error);
3842 }
3843
3844 /*
3845  * Convert a user file descriptor to a kernel file entry and check that, if it
3846  * is a capability, the correct rights are present. A reference on the file
3847  * entry is held upon returning.
3848  */
3849 int
3850 getvnode(struct thread *td, int fd, cap_rights_t *rightsp, struct file **fpp)
3851 {
3852     struct file *fp;
3853     int error;
3854
3855     error = fget_unlocked(td->td_proc->p_fd, fd, rightsp, &fp, NULL);
3856     if (error != 0)
3857         return (error);
3858
3859     /*
3860      * The file could be not of the vnode type, or it may be not
3861      * yet fully initialized, in which case the f_vnode pointer
3862      * may be set, but f_ops is still badfileops. E.g.,
3863      * devfs_open() transiently create such situation to
3864      * facilitate csw_d_fopen().
3865      *
3866      * Dupfdopen() handling in kern_openat() installs the
3867      * half-baked file into the process descriptor table, allowing
3868      * other thread to dereference it. Guard against the race by
3869      * checking f_ops.
3870      */
3871     if (fp->f_vnode == NULL || fp->f_ops == &badfileops) {
3872         fdrop(fp, td);
3873         return (EINVAL);
3874     }
3875     *fpp = fp;
3876     return (0);
3877 }
3878
3879 /*
3880  * Get an (NFS) file handle.
3881  */
3882 #ifdef _SYS_SYSPROTO_H_
3883 struct lgetfh_args {
3884     char    *fname;
3885     fhandle_t *fhp;
3886 };
3887 #endif
3888 int
3889 sys_lgetfh(struct thread *td, struct lgetfh_args *uap)
3890 {
3891     struct nameidata nd;
3892     fhandle_t fh;
3893     struct vnode *vp;
3894     int error;
3895
3896     error = priv_check(td, PRIV_VFS_GETFH);
3897     if (error != 0)
3898         return (error);
3899     NDINIT(&nd, LOOKUP, NOFOLLOW | LOCKLEAF | AUDITVNODE1, UIO_USERSPACE,
3900

```



```

3901     uap->fname, td);
3902     error = namei(&nd);
3903     if (error != 0)
3904         return (error);
3905     NDFREE(&nd, NDF_ONLY_PNBUF);
3906     vp = nd.ni_vp;
3907     bzero(&fh, sizeof(fh));
3908     fh.fh_fsid = vp->v_mount->mnt_stat.f_fsid;
3909     error = VOP_VPTOFH(vp, &fh.fh_fid);
3910     vput(vp);
3911     if (error == 0)
3912         error = copyout(&fh, uap->fhp, sizeof (fh));
3913     return (error);
3914 }
3915
3916 #ifndef _SYS_SYSPROTO_H_
3917 struct getfh_args {
3918     char *fname;
3919     fhandle_t *fhp;
3920 };
3921 #endif
3922 int
3923 sys_getfh(struct thread *td, struct getfh_args *uap)
3924 {
3925     struct nameidata nd;
3926     fhandle_t fh;
3927     struct vnode *vp;
3928     int error;
3929
3930     error = priv_check(td, PRIV_VFS_GETFH);
3931     if (error != 0)
3932         return (error);
3933     NDINIT(&nd, LOOKUP, FOLLOW | LOCKLEAF | AUDITVNODE1, UIO_USERSPACE,
3934     uap->fname, td);
3935     error = namei(&nd);
3936     if (error != 0)
3937         return (error);
3938     NDFREE(&nd, NDF_ONLY_PNBUF);
3939     vp = nd.ni_vp;
3940     bzero(&fh, sizeof(fh));
3941     fh.fh_fsid = vp->v_mount->mnt_stat.f_fsid;
3942     error = VOP_VPTOFH(vp, &fh.fh_fid);
3943     vput(vp);
3944     if (error == 0)
3945         error = copyout(&fh, uap->fhp, sizeof (fh));
3946     return (error);
3947 }
3948
3949 /*
3950  * syscall for the rpc.lockd to use to translate a NFS file handle into an
3951  * open descriptor.
3952  *
3953  * warning: do not remove the priv_check() call or this becomes one giant
3954  * security hole.
3955  */
3956 #ifndef _SYS_SYSPROTO_H_
3957 struct fhopen_args {
3958     const struct fhandle *u_fhp;
3959     int flags;
3960 };
3961 #endif
3962 int
3963 sys_fhopen(struct thread *td, struct fhopen_args *uap)
3964 {
3965     struct mount *mp;
3966     struct vnode *vp;
3967     struct fhandle fhp;
3968     struct file *fp;
3969     int fmode, error;
3970     int indx;
3971
3972     error = priv_check(td, PRIV_VFS_FHOPEN);
3973     if (error != 0)
3974         return (error);
3975     indx = -1;
3976     fmode = FFLAGS(uap->flags);
3977     /* why not allow a non-read/write open for our lockd? */
3978     if (((fmode & (FREAD | FWRITE)) == 0) || (fmode & O_CREAT))
3979         return (EINVAL);
3980     error = copyin(uap->u_fhp, &fhp, sizeof(fhp));
3981     if (error != 0)
3982         return(error);
3983     /* find the mount point */
3984     mp = vfs_busyfs(&fhp.fh_fsid);
3985     if (mp == NULL)
3986         return (ESTALE);
3987     /* now give me my vnode, it gets returned to me locked */
3988     error = VFS_FHTOVP(mp, &fhp.fh_fid, LK_EXCLUSIVE, &vp);
3989     vfs_unbusy(mp);
3990     if (error != 0)
3991         return (error);
3992
3993     error = falloc_noinstall(td, &fp);
3994     if (error != 0) {
3995         vput(vp);
3996         return (error);
3997     }
3998     /*

```

```

3999     * An extra reference on `fp' has been held for us by
4000     * falloc_noinstall().
4001     */
4002
4003 #ifdef INVARIANTS
4004     td->td_dupfd = -1;
4005 #endif
4006     error = vn_open_vnode(vp, fmode, td->td_ucred, td, fp);
4007     if (error != 0) {
4008         KASSERT(fp->f_ops == &badfileops,
4009             ("VOP_OPEN in fhopen() set f_ops"));
4010         KASSERT(td->td_dupfd < 0,
4011             ("fhopen() encountered fdopen()"));
4012
4013         vput(vp);
4014         goto bad;
4015     }
4016 #ifdef INVARIANTS
4017     td->td_dupfd = 0;
4018 #endif
4019     fp->f_vnode = vp;
4020     fp->f_seqcount = 1;
4021     finit(fp, (fmode & FMASK) | (fp->f_flag & FHASLOCK), DTYPE_VNODE, vp,
4022         &vnops);
4023     VOP_UNLOCK(vp, 0);
4024     if ((fmode & O_TRUNC) != 0) {
4025         error = fo_truncate(fp, 0, td->td_ucred, td);
4026         if (error != 0)
4027             goto bad;
4028     }
4029
4030     error = finstall(td, fp, &indx, fmode, NULL);
4031 bad:
4032     fdrop(fp, td);
4033     td->td_retval[0] = indx;
4034     return (error);
4035 }
4036
4037 /*
4038  * Stat an (NFS) file handle.
4039  */
4040 #ifndef _SYS_SYSPROTO_H_
4041 struct fhstat_args {
4042     struct fhandle *u_fhp;
4043     struct stat *sb;
4044 };
4045 #endif
4046 int
4047 sys_fhstat(struct thread *td, struct fhstat_args *uap)
4048 {
4049     struct stat sb;
4050     struct fhandle fh;
4051     int error;
4052
4053     error = copyin(uap->u_fhp, &fh, sizeof(fh));
4054     if (error != 0)
4055         return (error);
4056     error = kern_fhstat(td, fh, &sb);
4057     if (error == 0)
4058         error = copyout(&sb, uap->sb, sizeof(sb));
4059     return (error);
4060 }
4061
4062 int
4063 kern_fhstat(struct thread *td, struct fhandle fh, struct stat *sb)
4064 {
4065     struct mount *mp;
4066     struct vnode *vp;
4067     int error;
4068
4069     error = priv_check(td, PRIV_VFS_FHSTAT);
4070     if (error != 0)
4071         return (error);
4072     if ((mp = vfs_busyfs(&fh.fh_fsid)) == NULL)
4073         return (ESTALE);
4074     error = VFS_FHTOVP(mp, &fh.fh_fid, LK_EXCLUSIVE, &vp);
4075     vfs_unbusy(mp);
4076     if (error != 0)
4077         return (error);
4078     error = vn_stat(vp, sb, td->td_ucred, NOCRED, td);
4079     vput(vp);
4080     return (error);
4081 }
4082
4083 /*
4084  * Implement fstatfs() for (NFS) file handles.
4085  */
4086 #ifndef _SYS_SYSPROTO_H_
4087 struct fhstatfs_args {
4088     struct fhandle *u_fhp;
4089     struct statfs *buf;
4090 };
4091 #endif
4092 int
4093 sys_fhstatfs(struct thread *td, struct fhstatfs_args *uap)
4094 {
4095     struct statfs *sfp;
4096     fhandle_t fh;

```

```

4097     int error;
4098
4099     error = copyin(uap->u_fhp, &fh, sizeof(fhandle_t));
4100     if (error != 0)
4101         return (error);
4102     sfp = malloc(sizeof(struct statfs), M_STATFS, M_WAITOK);
4103     error = kern_fhstatfs(td, fh, sfp);
4104     if (error == 0)
4105         error = copyout(sfp, uap->buf, sizeof(*sfp));
4106     free(sfp, M_STATFS);
4107     return (error);
4108 }
4109
4110 int
4111 kern_fhstatfs(struct thread *td, fhandle_t fh, struct statfs *buf)
4112 {
4113     struct statfs *sp;
4114     struct mount *mp;
4115     struct vnode *vp;
4116     int error;
4117
4118     error = priv_check(td, PRIV_VFS_FHSTATFS);
4119     if (error != 0)
4120         return (error);
4121     if ((mp = vfs_bsyfs(&fh.fh_fsid)) == NULL)
4122         return (ESTALE);
4123     error = VFS_FHTOVP(mp, &fh.fh_fid, LK_EXCLUSIVE, &vp);
4124     if (error != 0) {
4125         vfs_unbusy(mp);
4126         return (error);
4127     }
4128     vput(vp);
4129     error = prison_canseemount(td->td_ucred, mp);
4130     if (error != 0)
4131         goto out;
4132 #ifdef MAC
4133     error = mac_mount_check_stat(td->td_ucred, mp);
4134     if (error != 0)
4135         goto out;
4136 #endif
4137     /*
4138      * Set these in case the underlying filesystem fails to do so.
4139      */
4140     sp = &mp->mnt_stat;
4141     sp->f_version = STATFS_VERSION;
4142     sp->f_namemax = NAME_MAX;
4143     sp->f_flags = mp->mnt_flag & MNT_VISFLAGMASK;
4144     error = VFS_STATFS(mp, sp);
4145     if (error == 0)
4146         *buf = *sp;
4147 out:
4148     vfs_unbusy(mp);
4149     return (error);
4150 }
4151
4152 int
4153 kern_posix_fallocate(struct thread *td, int fd, off_t offset, off_t len)
4154 {
4155     struct file *fp;
4156     struct mount *mp;
4157     struct vnode *vp;
4158     cap_rights_t rights;
4159     off_t olen, ooffset;
4160     int error;
4161
4162     if (offset < 0 || len <= 0)
4163         return (EINVAL);
4164     /* Check for wrap. */
4165     if (offset > OFF_MAX - len)
4166         return (EFBIG);
4167     error = fget(td, fd, cap_rights_init(&rights, CAP_PWRITE), &fp);
4168     if (error != 0)
4169         return (error);
4170     if ((fp->f_ops->fo_flags & DFLAG_SEEKABLE) == 0) {
4171         error = EPIPE;
4172         goto out;
4173     }
4174     if ((fp->f_flag & FWRITE) == 0) {
4175         error = EBADF;
4176         goto out;
4177     }
4178     if (fp->f_type != DTYPE_VNODE) {
4179         error = ENODEV;
4180         goto out;
4181     }
4182     vp = fp->f_vnode;
4183     if (vp->v_type != VREG) {
4184         error = ENODEV;
4185         goto out;
4186     }
4187
4188     /* Allocating blocks may take a long time, so iterate. */
4189     for (;;) {
4190         olen = len;
4191         ooffset = offset;
4192
4193         bwrite();
4194         mp = NULL;

```

```

4195         error = vn_start_write(vp, &mp, V_WAIT | PCATCH);
4196         if (error != 0)
4197             break;
4198         error = vn_lock(vp, LK_EXCLUSIVE);
4199         if (error != 0) {
4200             vn_finished_write(mp);
4201             break;
4202         }
4203 #ifdef MAC
4204         error = mac_vnode_check_write(td->td_ucred, fp->f_cred, vp);
4205         if (error == 0)
4206 #endif
4207             error = VOP_ALLOCATE(vp, &offset, &len);
4208         VOP_UNLOCK(vp, 0);
4209         vn_finished_write(mp);
4210
4211         if (olen + offset != offset + len) {
4212             panic("offset + len changed from %jx/%jx to %jx/%jx",
4213                 offset, olen, offset, len);
4214         }
4215         if (error != 0 || len == 0)
4216             break;
4217         KASSERT(olen > len, ("Iteration did not make progress?"));
4218         maybe_yield();
4219     }
4220 out:
4221     fdrop(fp, td);
4222     return (error);
4223 }
4224
4225 int
4226 sys_posix_fallocate(struct thread *td, struct posix_fallocate_args *uap)
4227 {
4228     int error;
4229
4230     error = kern_posix_fallocate(td, uap->fd, uap->offset, uap->len);
4231     return (kern_posix_error(td, error));
4232 }
4233
4234 /*
4235  * Unlike madvise(2), we do not make a best effort to remember every
4236  * possible caching hint. Instead, we remember the last setting with
4237  * the exception that we will allow POSIX_FADV_NORMAL to adjust the
4238  * region of any current setting.
4239  */
4240 int
4241 kern_posix_fadvise(struct thread *td, int fd, off_t offset, off_t len,
4242     int advice)
4243 {
4244     struct fadvise_info *fa, *new;
4245     struct file *fp;
4246     struct vnode *vp;
4247     cap_rights_t rights;
4248     off_t end;
4249     int error;
4250
4251     if (offset < 0 || len < 0 || offset > OFF_MAX - len)
4252         return (EINVAL);
4253     switch (advice) {
4254     case POSIX_FADV_SEQUENTIAL:
4255     case POSIX_FADV_RANDOM:
4256     case POSIX_FADV_NOREUSE:
4257         new = malloc(sizeof(*fa), M_FADVISE, M_WAITOK);
4258         break;
4259     case POSIX_FADV_NORMAL:
4260     case POSIX_FADV_WILLNEED:
4261     case POSIX_FADV_DONTNEED:
4262         new = NULL;
4263         break;
4264     default:
4265         return (EINVAL);
4266     }
4267     /* XXX: CAP_POSIX_FADVISE? */
4268     error = fget(td, fd, cap_rights_init(&rights), &fp);
4269     if (error != 0)
4270         goto out;
4271     if ((fp->f_ops->fo_flags & DFLAG_SEEKABLE) == 0) {
4272         error = ESPIPE;
4273         goto out;
4274     }
4275     if (fp->f_type != DTYPE_VNODE) {
4276         error = ENODEV;
4277         goto out;
4278     }
4279     vp = fp->f_vnode;
4280     if (vp->v_type != VREG) {
4281         error = ENODEV;
4282         goto out;
4283     }
4284     if (len == 0)
4285         end = OFF_MAX;
4286     else
4287         end = offset + len - 1;
4288     switch (advice) {
4289     case POSIX_FADV_SEQUENTIAL:
4290     case POSIX_FADV_RANDOM:
4291     case POSIX_FADV_NOREUSE:
4292         /*

```

```

4293     * Try to merge any existing non-standard region with
4294     * this new region if possible, otherwise create a new
4295     * non-standard region for this request.
4296     */
4297     mtx_pool_lock(mtxpool_sleep, fp);
4298     fa = fp->f_advice;
4299     if (fa != NULL && fa->fa_advice == advice &&
4300         ((fa->fa_start <= end && fa->fa_end >= offset) ||
4301          (end != OFF_MAX && fa->fa_start == end + 1) ||
4302          (fa->fa_end != OFF_MAX && fa->fa_end + 1 == offset))) {
4303         if (offset < fa->fa_start)
4304             fa->fa_start = offset;
4305         if (end > fa->fa_end)
4306             fa->fa_end = end;
4307     } else {
4308         new->fa_advice = advice;
4309         new->fa_start = offset;
4310         new->fa_end = end;
4311         fp->f_advice = new;
4312         new = fa;
4313     }
4314     mtx_pool_unlock(mtxpool_sleep, fp);
4315     break;
4316 case POSIX_FADV_NORMAL:
4317     /*
4318     * If a the "normal" region overlaps with an existing
4319     * non-standard region, trim or remove the
4320     * non-standard region.
4321     */
4322     mtx_pool_lock(mtxpool_sleep, fp);
4323     fa = fp->f_advice;
4324     if (fa != NULL) {
4325         if (offset <= fa->fa_start && end >= fa->fa_end) {
4326             new = fa;
4327             fp->f_advice = NULL;
4328         } else if (offset <= fa->fa_start &&
4329                    end >= fa->fa_start)
4330             fa->fa_start = end + 1;
4331         else if (offset <= fa->fa_end && end >= fa->fa_end)
4332             fa->fa_end = offset - 1;
4333         else if (offset >= fa->fa_start && end <= fa->fa_end) {
4334             /*
4335             * If the "normal" region is a middle
4336             * portion of the existing
4337             * non-standard region, just remove
4338             * the whole thing rather than picking
4339             * one side or the other to
4340             * preserve.
4341             */
4342             new = fa;
4343             fp->f_advice = NULL;
4344         }
4345     }
4346     mtx_pool_unlock(mtxpool_sleep, fp);
4347     break;
4348 case POSIX_FADV_WILLNEED:
4349 case POSIX_FADV_DONTNEED:
4350     error = VOP_ADVISE(vp, offset, end, advice);
4351     break;
4352 }
4353 out:
4354 if (fp != NULL)
4355     fdrop(fp, td);
4356 free(new, M_FADVISE);
4357 return (error);
4358 }
4359
4360 int
4361 sys_posix_fadvise(struct thread *td, struct posix_fadvise_args *uap)
4362 {
4363     int error;
4364
4365     error = kern_posix_fadvise(td, uap->fd, uap->offset, uap->len,
4366                               uap->advice);
4367     return (kern_posix_error(td, error));
4368 }

```