MidnightBSD / src Public

Code Issues 60 Pull requests Actions Projects Wiki

1691c07ff4

src / sys / kern / vfs_syscalls.c

laffer1 Sync with freebsd History

1 contributor

4368 lines (4034 sloc) 94 KB

```c
/*-
 * Copyright (c) 1989, 1993
 *	The Regents of the University of California.  All rights reserved.
 * (c) UNIX System Laboratories, Inc.
 * All or some portions of this file are derived from material licensed
 * to the University of California by American Telephone and Telegraph
 * Co. or Unix System Laboratories, Inc. and are reproduced herein with
 * the permission of UNIX System Laboratories, Inc.
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions
 * are met:
 * 1. Redistributions of source code must retain the above copyright
 *    notice, this list of conditions and the following disclaimer.
 * 2. Redistributions in binary form must reproduce the above copyright
 *    notice, this list of conditions and the following disclaimer in the
 *    documentation and/or other materials provided with the distribution.
 * 4. Neither the name of the University nor the names of its contributors
 *    may be used to endorse or promote products derived from this software
 *    without specific prior written permission.
 *
 * THIS SOFTWARE IS PROVIDED BY THE REGENTS AND CONTRIBUTORS ``AS IS'' AND
 * ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
 * IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
 * ARE DISCLAIMED.  IN NO EVENT SHALL THE REGENTS OR CONTRIBUTORS BE LIABLE
 * FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
 * DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
 * OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
 * HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
 * LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
 * OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
 * SUCH DAMAGE.
 *
 *	@(#)vfs_syscalls.c	8.13 (Berkeley) 4/15/94
 */

#include <sys/cdefs.h>
__FBSDID("$FreeBSD: stable/11/sys/kern/vfs_syscalls.c 338987 2018-09-27 18:54:41Z gordon $");

#include "opt_capsicum.h"
#include "opt_compat.h"
#include "opt_ktrace.h"

#include <sys/param.h>
#include <sys/systm.h>
#include <sys/bio.h>
#include <sys/buf.h>
#include <sys/capsicum.h>
#include <sys/disk.h>
#include <sys/sysent.h>
#include <sys/malloc.h>
#include <sys/mount.h>
#include <sys/mutex.h>
#include <sys/sysproto.h>
#include <sys/namei.h>
#include <sys/filedesc.h>
#include <sys/kernel.h>
#include <sys/fcntl.h>
#include <sys/file.h>
#include <sys/filio.h>
#include <sys/limits.h>
#include <sys/linker.h>
#include <sys/rwlock.h>
#include <sys/sdt.h>
#include <sys/stat.h>
#include <sys/sx.h>
#include <sys/unistd.h>
#include <sys/vnode.h>
#include <sys/priv.h>
#include <sys/proc.h>
#include <sys/dirent.h>
#include <sys/jail.h>
#include <sys/syscallsubr.h>
#include <sys/sysctl.h>
#ifdef KTRACE
#include <sys/ktrace.h>
#endif

```

```c
#include <machine/stdarg.h>

#include <security/audit/audit.h>
#include <security/mac/mac_framework.h>

#include <vm/vm.h>
#include <vm/vm_object.h>
#include <vm/vm_page.h>
#include <vm/uma.h>

#include <ufs/ufs/quota.h>

MALLOC_DEFINE(M_FADVISE, "fadvise", "posix_fadvise(2) information");

SDT_PROVIDER_DEFINE(vfs);
SDT_PROBE_DEFINE2(vfs, , stat, mode, "char *", "int");
SDT_PROBE_DEFINE2(vfs, , stat, reg, "char *", "int");

static int kern_chflagsat(struct thread *td, int fd, const char *path,
    enum uio_seg pathseg, u_long flags, int atflag);
static int setfflags(struct thread *td, struct vnode *, u_long);
static int getutimes(const struct timeval *, enum uio_seg, struct timespec *);
static int getutimens(const struct timespec *, enum uio_seg,
    struct timespec *, int *);
static int setutimes(struct thread *td, struct vnode *,
    const struct timespec *, int, int);
static int vn_access(struct vnode *vp, int user_flags, struct ucred *cred,
    struct thread *td);

/*
 * Sync each mounted filesystem.
 */
#ifndef _SYS_SYSPROTO_H_
struct sync_args {
	int     dummy;
};
#endif
/* ARGSUSED */
int
sys_sync(struct thread *td, struct sync_args *uap)
{
	struct mount *mp, *nmp;
	int save;

	mtx_lock(&mountlist_mtx);
	for (mp = TAILQ_FIRST(&mountlist); mp != NULL; mp = nmp) {
		if (vfs_busy(mp, MBF_NOWAIT | MBF_MNTLSTLOCK)) {
			nmp = TAILQ_NEXT(mp, mnt_list);
			continue;
		}
		if ((mp->mnt_flag & MNT_RDONLY) == 0 &&
		    vn_start_write(NULL, &mp, V_NOWAIT) == 0) {
			save = curthread_pflags_set(TDP_SYNCIO);
			vfs_msync(mp, MNT_NOWAIT);
			VFS_SYNC(mp, MNT_NOWAIT);
			curthread_pflags_restore(save);
			vn_finished_write(mp);
		}
		mtx_lock(&mountlist_mtx);
		nmp = TAILQ_NEXT(mp, mnt_list);
		vfs_unbusy(mp);
	}
	mtx_unlock(&mountlist_mtx);
	return (0);
}

/*
 * Change filesystem quotas.
 */
#ifndef _SYS_SYSPROTO_H_
struct quotactl_args {
	char *path;
	int cmd;
	int uid;
	caddr_t arg;
};
#endif
int
sys_quotactl(struct thread *td, struct quotactl_args *uap)
{
	struct mount *mp;
	struct nameidata nd;
	int error;

	AUDIT_ARG_CMD(uap->cmd);
	AUDIT_ARG_UID(uap->uid);
	if (!prison_allow(td->td_ucred, PR_ALLOW_QUOTAS))
		return (EPERM);
	NDINIT(&nd, LOOKUP, FOLLOW | LOCKLEAF | AUDITVNODE1, UIO_USERSPACE,
	    uap->path, td);
	if ((error = namei(&nd)) != 0)
		return (error);
	NDFREE(&nd, NDF_ONLY_PNBUF);
	mp = nd.ni_vp->v_mount;
	vfs_ref(mp);
	vput(nd.ni_vp);
	error = vfs_busy(mp, 0);
	vfs_rel(mp);
	if (error != 0)
		return (error);
	error = VFS_QUOTACTL(mp, uap->cmd, uap->uid, uap->arg);

	/*
	 * Since quota on operation typically needs to open quota
	 * file, the Q_QUOTAON handler needs to unbusy the mount point
	 * before calling into namei.  Otherwise, unmount might be
	 * started between two vfs_busy() invocations (first is our,
	 * second is from mount point cross-walk code in lookup()),
	 * causing deadlock.
	 *
	 * Require that Q_QUOTAON handles the vfs_busy() reference on
	 * its own, always returning with ubusied mount point.
	 */
	if ((uap->cmd >> SUBCMDSHIFT) != Q_QUOTAON &&
	    (uap->cmd >> SUBCMDSHIFT) != Q_QUOTAOFF)
		vfs_unbusy(mp);
	return (error);
}

/*
 * Used by statfs conversion routines to scale the block size up if
 * necessary so that all of the block counts are <= 'max_size'.  Note
 * that 'max_size' should be a bitmask, i.e. 2^n - 1 for some non-zero
 * value of 'n'.
 */
void
statfs_scale_blocks(struct statfs *sf, long max_size)
{
	uint64_t count;
	int shift;

	KASSERT(powerof2(max_size + 1), ("%s: invalid max_size", __func__));

	/*
	 * Attempt to scale the block counts to give a more accurate
	 * overview to userland of the ratio of free space to used
	 * space.  To do this, find the largest block count and compute
	 * a divisor that lets it fit into a signed integer <= max_size.
	 */
	if (sf->f_bavail < 0)
		count = -sf->f_bavail;
	else
		count = sf->f_bavail;
	count = MAX(sf->f_blocks, MAX(sf->f_bfree, count));
	if (count <= max_size)
		return;

	count >>= flsl(max_size);
	shift = 0;
	while (count > 0) {
		shift++;
		count >>=1;
	}

	sf->f_bsize <<= shift;
	sf->f_blocks >>= shift;
	sf->f_bfree >>= shift;
	sf->f_bavail >>= shift;
}

static int
kern_do_statfs(struct thread *td, struct mount *mp, struct statfs *buf)
{
	struct statfs *sp;
	int error;

	if (mp == NULL)
		return (EBADF);
	error = vfs_busy(mp, 0);
	vfs_rel(mp);
	if (error != 0)
		return (error);
#ifdef MAC
	error = mac_mount_check_stat(td->td_ucred, mp);
	if (error != 0)
		goto out;
#endif
	/*
	 * Set these in case the underlying filesystem fails to do so.
	 */
	sp = &mp->mnt_stat;
	sp->f_version = STATFS_VERSION;
	sp->f_namemax = NAME_MAX;
	sp->f_flags = mp->mnt_flag & MNT_VISFLAGMASK;
	error = VFS_STATFS(mp, sp);
	if (error != 0)
		goto out;
	*buf = *sp;
	if (priv_check(td, PRIV_VFS_GENERATION)) {
		buf->f_fsid.val[0] = buf->f_fsid.val[1] = 0;
		prison_enforce_statfs(td->td_ucred, mp, buf);
	}
out:
	vfs_unbusy(mp);
	return (error);
}
```

```c

/*
 * Get filesystem statistics.
 */
#ifndef _SYS_SYSPROTO_H_
struct statfs_args {
        char *path;
        struct statfs *buf;
};
#endif
int
sys_statfs(struct thread *td, struct statfs_args *uap)
{
        struct statfs *sfp;
        int error;

        sfp = malloc(sizeof(struct statfs), M_STATFS, M_WAITOK);
        error = kern_statfs(td, uap->path, UIO_USERSPACE, sfp);
        if (error == 0)
                error = copyout(sfp, uap->buf, sizeof(struct statfs));
        free(sfp, M_STATFS);
        return (error);
}

int
kern_statfs(struct thread *td, char *path, enum uio_seg pathseg,
    struct statfs *buf)
{
        struct mount *mp;
        struct nameidata nd;
        int error;

        NDINIT(&nd, LOOKUP, FOLLOW | LOCKSHARED | LOCKLEAF | AUDITVNODE1,
            pathseg, path, td);
        error = namei(&nd);
        if (error != 0)
                return (error);
        mp = nd.ni_vp->v_mount;
        vfs_ref(mp);
        NDFREE(&nd, NDF_ONLY_PNBUF);
        vput(nd.ni_vp);
        return (kern_do_statfs(td, mp, buf));
}

/*
 * Get filesystem statistics.
 */
#ifndef _SYS_SYSPROTO_H_
struct fstatfs_args {
        int fd;
        struct statfs *buf;
};
#endif
int
sys_fstatfs(struct thread *td, struct fstatfs_args *uap)
{
        struct statfs *sfp;
        int error;

        sfp = malloc(sizeof(struct statfs), M_STATFS, M_WAITOK);
        error = kern_fstatfs(td, uap->fd, sfp);
        if (error == 0)
                error = copyout(sfp, uap->buf, sizeof(struct statfs));
        free(sfp, M_STATFS);
        return (error);
}

int
kern_fstatfs(struct thread *td, int fd, struct statfs *buf)
{
        struct file *fp;
        struct mount *mp;
        struct vnode *vp;
        cap_rights_t rights;
        int error;

        AUDIT_ARG_FD(fd);
        error = getvnode(td, fd, cap_rights_init(&rights, CAP_FSTATFS), &fp);
        if (error != 0)
                return (error);
        vp = fp->f_vnode;
        vn_lock(vp, LK_SHARED | LK_RETRY);
#ifdef AUDIT
        AUDIT_ARG_VNODE1(vp);
#endif
        mp = vp->v_mount;
        if (mp != NULL)
                vfs_ref(mp);
        VOP_UNLOCK(vp, 0);
        fdrop(fp, td);
        return (kern_do_statfs(td, mp, buf));
}

/*
 * Get statistics on all filesystems.
 */
#ifndef _SYS_SYSPROTO_H_
struct getfsstat_args {
	struct statfs *buf;
	long bufsize;
	int mode;
};
#endif
int
sys_getfsstat(struct thread *td, struct getfsstat_args *uap)
{
	size_t count;
	int error;

	if (uap->bufsize < 0 || uap->bufsize > SIZE_MAX)
		return (EINVAL);
	error = kern_getfsstat(td, &uap->buf, uap->bufsize, &count,
	    UIO_USERSPACE, uap->mode);
	if (error == 0)
		td->td_retval[0] = count;
	return (error);
}

/*
 * If (bufsize > 0 && bufseg == UIO_SYSSPACE)
 *	The caller is responsible for freeing memory which will be allocated
 *	in '*buf'.
 */
int
kern_getfsstat(struct thread *td, struct statfs **buf, size_t bufsize,
    size_t *countp, enum uio_seg bufseg, int mode)
{
	struct mount *mp, *nmp;
	struct statfs *sfsp, *sp, *sptmp, *tofree;
	size_t count, maxcount;
	int error;

	switch (mode) {
	case MNT_WAIT:
	case MNT_NOWAIT:
		break;
	default:
		return (EINVAL);
	}
restart:
	maxcount = bufsize / sizeof(struct statfs);
	if (bufsize == 0) {
		sfsp = NULL;
		tofree = NULL;
	} else if (bufseg == UIO_USERSPACE) {
		sfsp = *buf;
		tofree = NULL;
	} else /* if (bufseg == UIO_SYSSPACE) */ {
		count = 0;
		mtx_lock(&mountlist_mtx);
		TAILQ_FOREACH(mp, &mountlist, mnt_list) {
			count++;
		}
		mtx_unlock(&mountlist_mtx);
		if (maxcount > count)
			maxcount = count;
		tofree = sfsp = *buf = malloc(maxcount * sizeof(struct statfs),
		    M_STATFS, M_WAITOK);
	}
	count = 0;
	mtx_lock(&mountlist_mtx);
	for (mp = TAILQ_FIRST(&mountlist); mp != NULL; mp = nmp) {
		if (prison_canseemount(td->td_ucred, mp) != 0) {
			nmp = TAILQ_NEXT(mp, mnt_list);
			continue;
		}
#ifdef MAC
		if (mac_mount_check_stat(td->td_ucred, mp) != 0) {
			nmp = TAILQ_NEXT(mp, mnt_list);
			continue;
		}
#endif
		if (mode == MNT_WAIT) {
			if (vfs_busy(mp, MBF_MNTLSTLOCK) != 0) {
				/*
				 * If vfs_busy() failed, and MBF_NOWAIT
				 * wasn't passed, then the mp is gone.
				 * Furthermore, because of MBF_MNTLSTLOCK,
				 * the mountlist_mtx was dropped.  We have
				 * no other choice than to start over.
				 */
				mtx_unlock(&mountlist_mtx);
				free(tofree, M_STATFS);
				goto restart;
			}
		} else {
			if (vfs_busy(mp, MBF_NOWAIT | MBF_MNTLSTLOCK) != 0) {
				nmp = TAILQ_NEXT(mp, mnt_list);
				continue;
			}
		}
		if (sfsp != NULL && count < maxcount) {
			sp = &mp->mnt_stat;
			/*
			 * Set these in case the underlying filesystem
			 * fails to do so.
			 */
			sp->f_version = STATFS_VERSION;
			sp->f_namemax = NAME_MAX;
			sp->f_flags = mp->mnt_flag & MNT_VISFLAGMASK;
			/*
			 * If MNT_NOWAIT is specified, do not refresh
			 * the fsstat cache.
			 */
			if (mode != MNT_NOWAIT) {
				error = VFS_STATFS(mp, sp);
				if (error != 0) {
					mtx_lock(&mountlist_mtx);
					nmp = TAILQ_NEXT(mp, mnt_list);
					vfs_unbusy(mp);
					continue;
				}
			}
			if (priv_check(td, PRIV_VFS_GENERATION)) {
				sptmp = malloc(sizeof(struct statfs), M_STATFS,
				    M_WAITOK);
				*sptmp = *sp;
				sptmp->f_fsid.val[0] = sptmp->f_fsid.val[1] = 0;
				prison_enforce_statfs(td->td_ucred, mp, sptmp);
				sp = sptmp;
			} else
				sptmp = NULL;
			if (bufseg == UIO_SYSSPACE) {
				bcopy(sp, sfsp, sizeof(*sp));
				free(sptmp, M_STATFS);
			} else /* if (bufseg == UIO_USERSPACE) */ {
				error = copyout(sp, sfsp, sizeof(*sp));
				free(sptmp, M_STATFS);
				if (error != 0) {
					vfs_unbusy(mp);
					return (error);
				}
			}
			sfsp++;
		}
		count++;
		mtx_lock(&mountlist_mtx);
		nmp = TAILQ_NEXT(mp, mnt_list);
		vfs_unbusy(mp);
	}
	mtx_unlock(&mountlist_mtx);
	if (sfsp != NULL && count > maxcount)
		*countp = maxcount;
	else
		*countp = count;
	return (0);
}

#ifdef COMPAT_FREEBSD4
/*
 * Get old format filesystem statistics.
 */
static void cvtstatfs(struct statfs *, struct ostatfs *);

#ifndef _SYS_SYSPROTO_H_
struct freebsd4_statfs_args {
	char *path;
	struct ostatfs *buf;
};
#endif
int
freebsd4_statfs(struct thread *td, struct freebsd4_statfs_args *uap)
{
	struct ostatfs osb;
	struct statfs *sfp;
	int error;

	sfp = malloc(sizeof(struct statfs), M_STATFS, M_WAITOK);
	error = kern_statfs(td, uap->path, UIO_USERSPACE, sfp);
	if (error == 0) {
		cvtstatfs(sfp, &osb);
		error = copyout(&osb, uap->buf, sizeof(osb));
	}
	free(sfp, M_STATFS);
	return (error);
}

/*
 * Get filesystem statistics.
 */
#ifndef _SYS_SYSPROTO_H_
struct freebsd4_fstatfs_args {
	int fd;
	struct ostatfs *buf;
};
#endif
int
freebsd4_fstatfs(struct thread *td, struct freebsd4_fstatfs_args *uap)
{
	struct ostatfs osb;
	struct statfs *sfp;
	int error;

	sfp = malloc(sizeof(struct statfs), M_STATFS, M_WAITOK);
```

```c
	error = kern_fstatfs(td, uap->fd, sfp);
	if (error == 0) {
		cvtstatfs(sfp, &osb);
		error = copyout(&osb, uap->buf, sizeof(osb));
	}
	free(sfp, M_STATFS);
	return (error);
}

/*
 * Get statistics on all filesystems.
 */
#ifndef _SYS_SYSPROTO_H_
struct freebsd4_getfsstat_args {
	struct ostatfs *buf;
	long bufsize;
	int mode;
};
#endif
int
freebsd4_getfsstat(struct thread *td, struct freebsd4_getfsstat_args *uap)
{
	struct statfs *buf, *sp;
	struct ostatfs osb;
	size_t count, size;
	int error;

	if (uap->bufsize < 0)
		return (EINVAL);
	count = uap->bufsize / sizeof(struct ostatfs);
	if (count > SIZE_MAX / sizeof(struct statfs))
		return (EINVAL);
	size = count * sizeof(struct statfs);
	error = kern_getfsstat(td, &buf, size, &count, UIO_SYSSPACE,
	    uap->mode);
	if (buf == NULL)
		return (EINVAL);
	td->td_retval[0] = count;
	if (size != 0) {
		sp = buf;
		while (count != 0 && error == 0) {
			cvtstatfs(sp, &osb);
			error = copyout(&osb, uap->buf, sizeof(osb));
			sp++;
			uap->buf++;
			count--;
		}
		free(buf, M_STATFS);
	}
	return (error);
}

/*
 * Implement fstatfs() for (NFS) file handles.
 */
#ifndef _SYS_SYSPROTO_H_
struct freebsd4_fhstatfs_args {
	struct fhandle *u_fhp;
	struct ostatfs *buf;
};
#endif
int
freebsd4_fhstatfs(struct thread *td, struct freebsd4_fhstatfs_args *uap)
{
	struct ostatfs osb;
	struct statfs *sfp;
	fhandle_t fh;
	int error;

	error = copyin(uap->u_fhp, &fh, sizeof(fhandle_t));
	if (error != 0)
		return (error);
	sfp = malloc(sizeof(struct statfs), M_STATFS, M_WAITOK);
	error = kern_fhstatfs(td, fh, sfp);
	if (error == 0) {
		cvtstatfs(sfp, &osb);
		error = copyout(&osb, uap->buf, sizeof(osb));
	}
	free(sfp, M_STATFS);
	return (error);
}

/*
 * Convert a new format statfs structure to an old format statfs structure.
 */
static void
cvtstatfs(struct statfs *nsp, struct ostatfs *osp)
{

	statfs_scale_blocks(nsp, LONG_MAX);
	bzero(osp, sizeof(*osp));
	osp->f_bsize = nsp->f_bsize;
	osp->f_iosize = MIN(nsp->f_iosize, LONG_MAX);
	osp->f_blocks = nsp->f_blocks;
	osp->f_bfree = nsp->f_bfree;
	osp->f_bavail = nsp->f_bavail;
	osp->f_files = MIN(nsp->f_files, LONG_MAX);
	osp->f_ffree = MIN(nsp->f_ffree, LONG_MAX);
	osp->f_owner = nsp->f_owner;
	osp->f_type = nsp->f_type;
	osp->f_flags = nsp->f_flags;
	osp->f_syncwrites = MIN(nsp->f_syncwrites, LONG_MAX);
	osp->f_asyncwrites = MIN(nsp->f_asyncwrites, LONG_MAX);
	osp->f_syncreads = MIN(nsp->f_syncreads, LONG_MAX);
	osp->f_asyncreads = MIN(nsp->f_asyncreads, LONG_MAX);
	strlcpy(osp->f_fstypename, nsp->f_fstypename,
	    MIN(MFSNAMELEN, OMFSNAMELEN));
	strlcpy(osp->f_mntonname, nsp->f_mntonname,
	    MIN(MNAMELEN, OMNAMELEN));
	strlcpy(osp->f_mntfromname, nsp->f_mntfromname,
	    MIN(MNAMELEN, OMNAMELEN));
	osp->f_fsid = nsp->f_fsid;
}
#endif /* COMPAT_FREEBSD4 */

/*
 * Change current working directory to a given file descriptor.
 */
#ifndef _SYS_SYSPROTO_H_
struct fchdir_args {
	int	fd;
};
#endif
int
sys_fchdir(struct thread *td, struct fchdir_args *uap)
{
	struct vnode *vp, *tdp;
	struct mount *mp;
	struct file *fp;
	cap_rights_t rights;
	int error;

	AUDIT_ARG_FD(uap->fd);
	error = getvnode(td, uap->fd, cap_rights_init(&rights, CAP_FCHDIR),
	    &fp);
	if (error != 0)
		return (error);
	vp = fp->f_vnode;
	vrefact(vp);
	fdrop(fp, td);
	vn_lock(vp, LK_SHARED | LK_RETRY);
	AUDIT_ARG_VNODE1(vp);
	error = change_dir(vp, td);
	while (!error && (mp = vp->v_mountedhere) != NULL) {
		if (vfs_busy(mp, 0))
			continue;
		error = VFS_ROOT(mp, LK_SHARED, &tdp);
		vfs_unbusy(mp);
		if (error != 0)
			break;
		vput(vp);
		vp = tdp;
	}
	if (error != 0) {
		vput(vp);
		return (error);
	}
	VOP_UNLOCK(vp, 0);
	pwd_chdir(td, vp);
	return (0);
}

/*
 * Change current working directory (``.'').
 */
#ifndef _SYS_SYSPROTO_H_
struct chdir_args {
	char	*path;
};
#endif
int
sys_chdir(struct thread *td, struct chdir_args *uap)
{

	return (kern_chdir(td, uap->path, UIO_USERSPACE));
}

int
kern_chdir(struct thread *td, char *path, enum uio_seg pathseg)
{
	struct nameidata nd;
	int error;

	NDINIT(&nd, LOOKUP, FOLLOW | LOCKSHARED | LOCKLEAF | AUDITVNODE1,
	    pathseg, path, td);
	if ((error = namei(&nd)) != 0)
		return (error);
	if ((error = change_dir(nd.ni_vp, td)) != 0) {
		vput(nd.ni_vp);
		NDFREE(&nd, NDF_ONLY_PNBUF);
		return (error);
	}
	VOP_UNLOCK(nd.ni_vp, 0);
	NDFREE(&nd, NDF_ONLY_PNBUF);
	pwd_chdir(td, nd.ni_vp);
	return (0);
```

```c
}

/*
 * Change notion of root (``/'') directory.
 */
#ifndef _SYS_SYSPROTO_H_
struct chroot_args {
	char	*path;
};
#endif
int
sys_chroot(struct thread *td, struct chroot_args *uap)
{
	struct nameidata nd;
	int error;

	error = priv_check(td, PRIV_VFS_CHROOT);
	if (error != 0)
		return (error);
	NDINIT(&nd, LOOKUP, FOLLOW | LOCKSHARED | LOCKLEAF | AUDITVNODE1,
	    UIO_USERSPACE, uap->path, td);
	error = namei(&nd);
	if (error != 0)
		goto error;
	error = change_dir(nd.ni_vp, td);
	if (error != 0)
		goto e_vunlock;
#ifdef MAC
	error = mac_vnode_check_chroot(td->td_ucred, nd.ni_vp);
	if (error != 0)
		goto e_vunlock;
#endif
	VOP_UNLOCK(nd.ni_vp, 0);
	error = pwd_chroot(td, nd.ni_vp);
	vrele(nd.ni_vp);
	NDFREE(&nd, NDF_ONLY_PNBUF);
	return (error);
e_vunlock:
	vput(nd.ni_vp);
error:
	NDFREE(&nd, NDF_ONLY_PNBUF);
	return (error);
}

/*
 * Common routine for chroot and chdir.  Callers must provide a locked vnode
 * instance.
 */
int
change_dir(struct vnode *vp, struct thread *td)
{
#ifdef MAC
	int error;
#endif

	ASSERT_VOP_LOCKED(vp, "change_dir(): vp not locked");
	if (vp->v_type != VDIR)
		return (ENOTDIR);
#ifdef MAC
	error = mac_vnode_check_chdir(td->td_ucred, vp);
	if (error != 0)
		return (error);
#endif
	return (VOP_ACCESS(vp, VEXEC, td->td_ucred, td));
}

static __inline void
flags_to_rights(int flags, cap_rights_t *rightsp)
{

	if (flags & O_EXEC) {
		cap_rights_set(rightsp, CAP_FEXECVE);
	} else {
		switch ((flags & O_ACCMODE)) {
		case O_RDONLY:
			cap_rights_set(rightsp, CAP_READ);
			break;
		case O_RDWR:
			cap_rights_set(rightsp, CAP_READ);
			/* FALLTHROUGH */
		case O_WRONLY:
			cap_rights_set(rightsp, CAP_WRITE);
			if (!(flags & (O_APPEND | O_TRUNC)))
				cap_rights_set(rightsp, CAP_SEEK);
			break;
		}
	}

	if (flags & O_CREAT)
		cap_rights_set(rightsp, CAP_CREATE);

	if (flags & O_TRUNC)
		cap_rights_set(rightsp, CAP_FTRUNCATE);

	if (flags & (O_SYNC | O_FSYNC))
		cap_rights_set(rightsp, CAP_FSYNC);

	if (flags & (O_EXLOCK | O_SHLOCK))
```

```
		cap_rights_set(rightsp, CAP_FLOCK);
}

/*
 * Check permissions, allocate an open file structure, and call the device
 * open routine if any.
 */
#ifndef _SYS_SYSPROTO_H_
struct open_args {
	char	*path;
	int	flags;
	int	mode;
};
#endif
int
sys_open(struct thread *td, struct open_args *uap)
{

	return (kern_openat(td, AT_FDCWD, uap->path, UIO_USERSPACE,
	    uap->flags, uap->mode));
}

#ifndef _SYS_SYSPROTO_H_
struct openat_args {
	int	fd;
	char	*path;
	int	flag;
	int	mode;
};
#endif
int
sys_openat(struct thread *td, struct openat_args *uap)
{

	AUDIT_ARG_FD(uap->fd);
	return (kern_openat(td, uap->fd, uap->path, UIO_USERSPACE, uap->flag,
	    uap->mode));
}

int
kern_openat(struct thread *td, int fd, char *path, enum uio_seg pathseg,
    int flags, int mode)
{
	struct proc *p = td->td_proc;
	struct filedesc *fdp = p->p_fd;
	struct file *fp;
	struct vnode *vp;
	struct nameidata nd;
	cap_rights_t rights;
	int cmode, error, indx;

	indx = -1;

	AUDIT_ARG_FFLAGS(flags);
	AUDIT_ARG_MODE(mode);
	cap_rights_init(&rights, CAP_LOOKUP);
	flags_to_rights(flags, &rights);
	/*
	 * Only one of the O_EXEC, O_RDONLY, O_WRONLY and O_RDWR flags
	 * may be specified.
	 */
	if (flags & O_EXEC) {
		if (flags & O_ACCMODE)
			return (EINVAL);
	} else if ((flags & O_ACCMODE) == O_ACCMODE) {
		return (EINVAL);
	} else {
		flags = FFLAGS(flags);
	}

	/*
	 * Allocate a file structure. The descriptor to reference it
	 * is allocated and set by finstall() below.
	 */
	error = falloc_noinstall(td, &fp);
	if (error != 0)
		return (error);
	/*
	 * An extra reference on `fp' has been held for us by
	 * falloc_noinstall().
	 */
	/* Set the flags early so the finit in devfs can pick them up. */
	fp->f_flag = flags & FMASK;
	cmode = ((mode & ~fdp->fd_cmask) & ALLPERMS) & ~S_ISTXT;
	NDINIT_ATRIGHTS(&nd, LOOKUP, FOLLOW | AUDITVNODE1, pathseg, path, fd,
	    &rights, td);
	td->td_dupfd = -1;		/* XXX check for fdopen */
	error = vn_open(&nd, &flags, cmode, fp);
	if (error != 0) {
		/*
		 * If the vn_open replaced the method vector, something
		 * wonderous happened deep below and we just pass it up
		 * pretending we know what we do.
		 */
		if (error == ENXIO && fp->f_ops != &badfileops)
			goto success;

		/*
```

```
		 * Handle special fdopen() case. bleh.
		 *
		 * Don't do this for relative (capability) lookups; we don't
		 * understand exactly what would happen, and we don't think
		 * that it ever should.
		 */
		if ((nd.ni_lcf & NI_LCF_STRICTRELATIVE) == 0 &&
		    (error == ENODEV || error == ENXIO) &&
		    td->td_dupfd >= 0) {
			error = dupfdopen(td, fdp, td->td_dupfd, flags, error,
			    &indx);
			if (error == 0)
				goto success;
		}

		goto bad;
	}
	td->td_dupfd = 0;
	NDFREE(&nd, NDF_ONLY_PNBUF);
	vp = nd.ni_vp;

	/*
	 * Store the vnode, for any f_type. Typically, the vnode use
	 * count is decremented by direct call to vn_closefile() for
	 * files that switched type in the cdevsw fdopen() method.
	 */
	fp->f_vnode = vp;
	/*
	 * If the file wasn't claimed by devfs bind it to the normal
	 * vnode operations here.
	 */
	if (fp->f_ops == &badfileops) {
		KASSERT(vp->v_type != VFIFO, ("Unexpected fifo."));
		fp->f_seqcount = 1;
		finit(fp, (flags & FMASK) | (fp->f_flag & FHASLOCK),
		    DTYPE_VNODE, vp, &vnops);
	}

	VOP_UNLOCK(vp, 0);
	if (flags & O_TRUNC) {
		error = fo_truncate(fp, 0, td->td_ucred, td);
		if (error != 0)
			goto bad;
	}
success:
	/*
	 * If we haven't already installed the FD (for dupfdopen), do so now.
	 */
	if (indx == -1) {
		struct filecaps *fcaps;

#ifdef CAPABILITIES
		if ((nd.ni_lcf & NI_LCF_STRICTRELATIVE) != 0)
			fcaps = &nd.ni_filecaps;
		else
#endif
			fcaps = NULL;
		error = finstall(td, fp, &indx, flags, fcaps);
		/* On success finstall() consumes fcaps. */
		if (error != 0) {
			filecaps_free(&nd.ni_filecaps);
			goto bad;
		}
	} else {
		filecaps_free(&nd.ni_filecaps);
	}

	/*
	 * Release our private reference, leaving the one associated with
	 * the descriptor table intact.
	 */
	fdrop(fp, td);
	td->td_retval[0] = indx;
	return (0);
bad:
	KASSERT(indx == -1, ("indx=%d, should be -1", indx));
	fdrop(fp, td);
	return (error);
}

#ifdef COMPAT_43
/*
 * Create a file.
 */
#ifndef _SYS_SYSPROTO_H_
struct ocreat_args {
	char	*path;
	int	mode;
};
#endif
int
ocreat(struct thread *td, struct ocreat_args *uap)
{

	return (kern_openat(td, AT_FDCWD, uap->path, UIO_USERSPACE,
	    O_WRONLY | O_CREAT | O_TRUNC, uap->mode));
}
#endif /* COMPAT_43 */
```

```

/*
 * Create a special file.
 */
#ifndef _SYS_SYSPROTO_H_
struct mknod_args {
	char	*path;
	int	mode;
	int	dev;
};
#endif
int
sys_mknod(struct thread *td, struct mknod_args *uap)
{

	return (kern_mknodat(td, AT_FDCWD, uap->path, UIO_USERSPACE,
	    uap->mode, uap->dev));
}

#ifndef _SYS_SYSPROTO_H_
struct mknodat_args {
	int	fd;
	char	*path;
	mode_t	mode;
	dev_t	dev;
};
#endif
int
sys_mknodat(struct thread *td, struct mknodat_args *uap)
{

	return (kern_mknodat(td, uap->fd, uap->path, UIO_USERSPACE, uap->mode,
	    uap->dev));
}

int
kern_mknodat(struct thread *td, int fd, char *path, enum uio_seg pathseg,
    int mode, int dev)
{
	struct vnode *vp;
	struct mount *mp;
	struct vattr vattr;
	struct nameidata nd;
	cap_rights_t rights;
	int error, whiteout = 0;

	AUDIT_ARG_MODE(mode);
	AUDIT_ARG_DEV(dev);
	switch (mode & S_IFMT) {
	case S_IFCHR:
	case S_IFBLK:
		error = priv_check(td, PRIV_VFS_MKNOD_DEV);
		if (error == 0 && dev == VNOVAL)
			error = EINVAL;
		break;
	case S_IFWHT:
		error = priv_check(td, PRIV_VFS_MKNOD_WHT);
		break;
	case S_IFIFO:
		if (dev == 0)
			return (kern_mkfifoat(td, fd, path, pathseg, mode));
		/* FALLTHROUGH */
	default:
		error = EINVAL;
		break;
	}
	if (error != 0)
		return (error);
restart:
	bwillwrite();
	NDINIT_ATRIGHTS(&nd, CREATE, LOCKPARENT | SAVENAME | AUDITVNODE1 |
	    NOCACHE, pathseg, path, fd, cap_rights_init(&rights, CAP_MKNODAT),
	    td);
	if ((error = namei(&nd)) != 0)
		return (error);
	vp = nd.ni_vp;
	if (vp != NULL) {
		NDFREE(&nd, NDF_ONLY_PNBUF);
		if (vp == nd.ni_dvp)
			vrele(nd.ni_dvp);
		else
			vput(nd.ni_dvp);
		vrele(vp);
		return (EEXIST);
	} else {
		VATTR_NULL(&vattr);
		vattr.va_mode = (mode & ALLPERMS) &
		    ~td->td_proc->p_fd->fd_cmask;
		vattr.va_rdev = dev;
		whiteout = 0;

		switch (mode & S_IFMT) {
		case S_IFCHR:
			vattr.va_type = VCHR;
			break;
		case S_IFBLK:
			vattr.va_type = VBLK;
			break;
```

```c
		case S_IFWHT:
			whiteout = 1;
			break;
		default:
			panic("kern_mknod: invalid mode");
		}
	}
	if (vn_start_write(nd.ni_dvp, &mp, V_NOWAIT) != 0) {
		NDFREE(&nd, NDF_ONLY_PNBUF);
		vput(nd.ni_dvp);
		if ((error = vn_start_write(NULL, &mp, V_XSLEEP | PCATCH)) != 0)
			return (error);
		goto restart;
	}
#ifdef MAC
	if (error == 0 && !whiteout)
		error = mac_vnode_check_create(td->td_ucred, nd.ni_dvp,
		    &nd.ni_cnd, &vattr);
#endif
	if (error == 0) {
		if (whiteout)
			error = VOP_WHITEOUT(nd.ni_dvp, &nd.ni_cnd, CREATE);
		else {
			error = VOP_MKNOD(nd.ni_dvp, &nd.ni_vp,
						&nd.ni_cnd, &vattr);
			if (error == 0)
				vput(nd.ni_vp);
		}
	}
	NDFREE(&nd, NDF_ONLY_PNBUF);
	vput(nd.ni_dvp);
	vn_finished_write(mp);
	return (error);
}

/*
 * Create a named pipe.
 */
#ifndef _SYS_SYSPROTO_H_
struct mkfifo_args {
	char	*path;
	int	mode;
};
#endif
int
sys_mkfifo(struct thread *td, struct mkfifo_args *uap)
{

	return (kern_mkfifoat(td, AT_FDCWD, uap->path, UIO_USERSPACE,
	    uap->mode));
}

#ifndef _SYS_SYSPROTO_H_
struct mkfifoat_args {
	int	fd;
	char	*path;
	mode_t	mode;
};
#endif
int
sys_mkfifoat(struct thread *td, struct mkfifoat_args *uap)
{

	return (kern_mkfifoat(td, uap->fd, uap->path, UIO_USERSPACE,
	    uap->mode));
}

int
kern_mkfifoat(struct thread *td, int fd, char *path, enum uio_seg pathseg,
    int mode)
{
	struct mount *mp;
	struct vattr vattr;
	struct nameidata nd;
	cap_rights_t rights;
	int error;

	AUDIT_ARG_MODE(mode);
restart:
	bwillwrite();
	NDINIT_ATRIGHTS(&nd, CREATE, LOCKPARENT | SAVENAME | AUDITVNODE1 |
	    NOCACHE, pathseg, path, fd, cap_rights_init(&rights, CAP_MKFIFOAT),
	    td);
	if ((error = namei(&nd)) != 0)
		return (error);
	if (nd.ni_vp != NULL) {
		NDFREE(&nd, NDF_ONLY_PNBUF);
		if (nd.ni_vp == nd.ni_dvp)
			vrele(nd.ni_dvp);
		else
			vput(nd.ni_dvp);
		vrele(nd.ni_vp);
		return (EEXIST);
	}
	if (vn_start_write(nd.ni_dvp, &mp, V_NOWAIT) != 0) {
		NDFREE(&nd, NDF_ONLY_PNBUF);
		vput(nd.ni_dvp);
		if ((error = vn_start_write(NULL, &mp, V_XSLEEP | PCATCH)) != 0)
```

```
				return (error);
			goto restart;
		}
		VATTR_NULL(&vattr);
		vattr.va_type = VFIFO;
		vattr.va_mode = (mode & ALLPERMS) & ~td->td_proc->p_fd->fd_cmask;
#ifdef MAC
		error = mac_vnode_check_create(td->td_ucred, nd.ni_dvp, &nd.ni_cnd,
		    &vattr);
		if (error != 0)
			goto out;
#endif
		error = VOP_MKNOD(nd.ni_dvp, &nd.ni_vp, &nd.ni_cnd, &vattr);
		if (error == 0)
			vput(nd.ni_vp);
#ifdef MAC
out:
#endif
		vput(nd.ni_dvp);
		vn_finished_write(mp);
		NDFREE(&nd, NDF_ONLY_PNBUF);
		return (error);
}

/*
 * Make a hard file link.
 */
#ifndef _SYS_SYSPROTO_H_
struct link_args {
		char	*path;
		char	*link;
};
#endif
int
sys_link(struct thread *td, struct link_args *uap)
{

		return (kern_linkat(td, AT_FDCWD, AT_FDCWD, uap->path, uap->link,
		    UIO_USERSPACE, FOLLOW));
}

#ifndef _SYS_SYSPROTO_H_
struct linkat_args {
		int	fd1;
		char	*path1;
		int	fd2;
		char	*path2;
		int	flag;
};
#endif
int
sys_linkat(struct thread *td, struct linkat_args *uap)
{
		int flag;

		flag = uap->flag;
		if (flag & ~AT_SYMLINK_FOLLOW)
			return (EINVAL);

		return (kern_linkat(td, uap->fd1, uap->fd2, uap->path1, uap->path2,
		    UIO_USERSPACE, (flag & AT_SYMLINK_FOLLOW) ? FOLLOW : NOFOLLOW));
}

int hardlink_check_uid = 0;
SYSCTL_INT(_security_bsd, OID_AUTO, hardlink_check_uid, CTLFLAG_RW,
    &hardlink_check_uid, 0,
    "Unprivileged processes cannot create hard links to files owned by other "
    "users");
static int hardlink_check_gid = 0;
SYSCTL_INT(_security_bsd, OID_AUTO, hardlink_check_gid, CTLFLAG_RW,
    &hardlink_check_gid, 0,
    "Unprivileged processes cannot create hard links to files owned by other "
    "groups");

static int
can_hardlink(struct vnode *vp, struct ucred *cred)
{
		struct vattr va;
		int error;

		if (!hardlink_check_uid && !hardlink_check_gid)
			return (0);

		error = VOP_GETATTR(vp, &va, cred);
		if (error != 0)
			return (error);

		if (hardlink_check_uid && cred->cr_uid != va.va_uid) {
			error = priv_check_cred(cred, PRIV_VFS_LINK, 0);
			if (error != 0)
				return (error);
		}

		if (hardlink_check_gid && !groupmember(va.va_gid, cred)) {
			error = priv_check_cred(cred, PRIV_VFS_LINK, 0);
			if (error != 0)
				return (error);
		}

	return (0);
}

int
kern_linkat(struct thread *td, int fd1, int fd2, char *path1, char *path2,
    enum uio_seg segflg, int follow)
{
	struct vnode *vp;
	struct mount *mp;
	struct nameidata nd;
	cap_rights_t rights;
	int error;

again:
	bwillwrite();
	NDINIT_ATRIGHTS(&nd, LOOKUP, follow | AUDITVNODE1, segflg, path1, fd1,
	    cap_rights_init(&rights, CAP_LINKAT_SOURCE), td);

	if ((error = namei(&nd)) != 0)
		return (error);
	NDFREE(&nd, NDF_ONLY_PNBUF);
	vp = nd.ni_vp;
	if (vp->v_type == VDIR) {
		vrele(vp);
		return (EPERM);		/* POSIX */
	}
	NDINIT_ATRIGHTS(&nd, CREATE,
	    LOCKPARENT | SAVENAME | AUDITVNODE2 | NOCACHE, segflg, path2, fd2,
	    cap_rights_init(&rights, CAP_LINKAT_TARGET), td);
	if ((error = namei(&nd)) == 0) {
		if (nd.ni_vp != NULL) {
			NDFREE(&nd, NDF_ONLY_PNBUF);
			if (nd.ni_dvp == nd.ni_vp)
				vrele(nd.ni_dvp);
			else
				vput(nd.ni_dvp);
			vrele(nd.ni_vp);
			vrele(vp);
			return (EEXIST);
		} else if (nd.ni_dvp->v_mount != vp->v_mount) {
			/*
			 * Cross-device link.  No need to recheck
			 * vp->v_type, since it cannot change, except
			 * to VBAD.
			 */
			NDFREE(&nd, NDF_ONLY_PNBUF);
			vput(nd.ni_dvp);
			vrele(vp);
			return (EXDEV);
		} else if ((error = vn_lock(vp, LK_EXCLUSIVE)) == 0) {
			error = can_hardlink(vp, td->td_ucred);
#ifdef MAC
			if (error == 0)
				error = mac_vnode_check_link(td->td_ucred,
				    nd.ni_dvp, vp, &nd.ni_cnd);
#endif
			if (error != 0) {
				vput(vp);
				vput(nd.ni_dvp);
				NDFREE(&nd, NDF_ONLY_PNBUF);
				return (error);
			}
			error = vn_start_write(vp, &mp, V_NOWAIT);
			if (error != 0) {
				vput(vp);
				vput(nd.ni_dvp);
				NDFREE(&nd, NDF_ONLY_PNBUF);
				error = vn_start_write(NULL, &mp,
				    V_XSLEEP | PCATCH);
				if (error != 0)
					return (error);
				goto again;
			}
			error = VOP_LINK(nd.ni_dvp, vp, &nd.ni_cnd);
			VOP_UNLOCK(vp, 0);
			vput(nd.ni_dvp);
			vn_finished_write(mp);
			NDFREE(&nd, NDF_ONLY_PNBUF);
		} else {
			vput(nd.ni_dvp);
			NDFREE(&nd, NDF_ONLY_PNBUF);
			vrele(vp);
			goto again;
		}
	}
	vrele(vp);
	return (error);
}

/*
 * Make a symbolic link.
 */
#ifndef _SYS_SYSPROTO_H_
struct symlink_args {
	char	*path;
	char	*link;
};
```

```
#endif
int
sys_symlink(struct thread *td, struct symlink_args *uap)
{

	return (kern_symlinkat(td, uap->path, AT_FDCWD, uap->link,
	    UIO_USERSPACE));
}

#ifndef _SYS_SYSPROTO_H_
struct symlinkat_args {
	char    *path;
	int     fd;
	char    *path2;
};
#endif
int
sys_symlinkat(struct thread *td, struct symlinkat_args *uap)
{

	return (kern_symlinkat(td, uap->path1, uap->fd, uap->path2,
	    UIO_USERSPACE));
}

int
kern_symlinkat(struct thread *td, char *path1, int fd, char *path2,
    enum uio_seg segflg)
{
	struct mount *mp;
	struct vattr vattr;
	char *syspath;
	struct nameidata nd;
	int error;
	cap_rights_t rights;

	if (segflg == UIO_SYSSPACE) {
		syspath = path1;
	} else {
		syspath = uma_zalloc(namei_zone, M_WAITOK);
		if ((error = copyinstr(path1, syspath, MAXPATHLEN, NULL)) != 0)
			goto out;
	}
	AUDIT_ARG_TEXT(syspath);
restart:
	bwillwrite();
	NDINIT_ATRIGHTS(&nd, CREATE, LOCKPARENT | SAVENAME | AUDITVNODE1 |
	    NOCACHE, segflg, path2, fd, cap_rights_init(&rights, CAP_SYMLINKAT),
	    td);
	if ((error = namei(&nd)) != 0)
		goto out;
	if (nd.ni_vp) {
		NDFREE(&nd, NDF_ONLY_PNBUF);
		if (nd.ni_vp == nd.ni_dvp)
			vrele(nd.ni_dvp);
		else
			vput(nd.ni_dvp);
		vrele(nd.ni_vp);
		error = EEXIST;
		goto out;
	}
	if (vn_start_write(nd.ni_dvp, &mp, V_NOWAIT) != 0) {
		NDFREE(&nd, NDF_ONLY_PNBUF);
		vput(nd.ni_dvp);
		if ((error = vn_start_write(NULL, &mp, V_XSLEEP | PCATCH)) != 0)
			goto out;
		goto restart;
	}
	VATTR_NULL(&vattr);
	vattr.va_mode = ACCESSPERMS &~ td->td_proc->p_fd->fd_cmask;
#ifdef MAC
	vattr.va_type = VLNK;
	error = mac_vnode_check_create(td->td_ucred, nd.ni_dvp, &nd.ni_cnd,
	    &vattr);
	if (error != 0)
		goto out2;
#endif
	error = VOP_SYMLINK(nd.ni_dvp, &nd.ni_vp, &nd.ni_cnd, &vattr, syspath);
	if (error == 0)
		vput(nd.ni_vp);
#ifdef MAC
out2:
#endif
	NDFREE(&nd, NDF_ONLY_PNBUF);
	vput(nd.ni_dvp);
	vn_finished_write(mp);
out:
	if (segflg != UIO_SYSSPACE)
		uma_zfree(namei_zone, syspath);
	return (error);
}

/*
 * Delete a whiteout from the filesystem.
 */
#ifndef _SYS_SYSPROTO_H_
struct undelete_args {
	char *path;
};
```

```c
#endif
int
sys_undelete(struct thread *td, struct undelete_args *uap)
{
	struct mount *mp;
	struct nameidata nd;
	int error;

restart:
	bwillwrite();
	NDINIT(&nd, DELETE, LOCKPARENT | DOWHITEOUT | AUDITVNODE1,
	    UIO_USERSPACE, uap->path, td);
	error = namei(&nd);
	if (error != 0)
		return (error);

	if (nd.ni_vp != NULLVP || !(nd.ni_cnd.cn_flags & ISWHITEOUT)) {
		NDFREE(&nd, NDF_ONLY_PNBUF);
		if (nd.ni_vp == nd.ni_dvp)
			vrele(nd.ni_dvp);
		else
			vput(nd.ni_dvp);
		if (nd.ni_vp)
			vrele(nd.ni_vp);
		return (EEXIST);
	}
	if (vn_start_write(nd.ni_dvp, &mp, V_NOWAIT) != 0) {
		NDFREE(&nd, NDF_ONLY_PNBUF);
		vput(nd.ni_dvp);
		if ((error = vn_start_write(NULL, &mp, V_XSLEEP | PCATCH)) != 0)
			return (error);
		goto restart;
	}
	error = VOP_WHITEOUT(nd.ni_dvp, &nd.ni_cnd, DELETE);
	NDFREE(&nd, NDF_ONLY_PNBUF);
	vput(nd.ni_dvp);
	vn_finished_write(mp);
	return (error);
}

/*
 * Delete a name from the filesystem.
 */
#ifndef _SYS_SYSPROTO_H_
struct unlink_args {
	char	*path;
};
#endif
int
sys_unlink(struct thread *td, struct unlink_args *uap)
{

	return (kern_unlinkat(td, AT_FDCWD, uap->path, UIO_USERSPACE, 0));
}

#ifndef _SYS_SYSPROTO_H_
struct unlinkat_args {
	int	fd;
	char	*path;
	int	flag;
};
#endif
int
sys_unlinkat(struct thread *td, struct unlinkat_args *uap)
{
	int flag = uap->flag;
	int fd = uap->fd;
	char *path = uap->path;

	if (flag & ~AT_REMOVEDIR)
		return (EINVAL);

	if (flag & AT_REMOVEDIR)
		return (kern_rmdirat(td, fd, path, UIO_USERSPACE));
	else
		return (kern_unlinkat(td, fd, path, UIO_USERSPACE, 0));
}

int
kern_unlinkat(struct thread *td, int fd, char *path, enum uio_seg pathseg,
    ino_t oldinum)
{
	struct mount *mp;
	struct vnode *vp;
	struct nameidata nd;
	struct stat sb;
	cap_rights_t rights;
	int error;

restart:
	bwillwrite();
	NDINIT_ATRIGHTS(&nd, DELETE, LOCKPARENT | LOCKLEAF | AUDITVNODE1,
	    pathseg, path, fd, cap_rights_init(&rights, CAP_UNLINKAT), td);
	if ((error = namei(&nd)) != 0)
		return (error == EINVAL ? EPERM : error);
	vp = nd.ni_vp;
	if (vp->v_type == VDIR && oldinum == 0) {
		error = EPERM;		/* POSIX */
	} else if (oldinum != 0 &&
		  ((error = vn_stat(vp, &sb, td->td_ucred, NOCRED, td)) == 0) &&
		  sb.st_ino != oldinum) {
			error = EIDRM;	/* Identifier removed */
	} else {
		/*
		 * The root of a mounted filesystem cannot be deleted.
		 *
		 * XXX: can this only be a VDIR case?
		 */
		if (vp->v_vflag & VV_ROOT)
			error = EBUSY;
	}
	if (error == 0) {
		if (vn_start_write(nd.ni_dvp, &mp, V_NOWAIT) != 0) {
			NDFREE(&nd, NDF_ONLY_PNBUF);
			vput(nd.ni_dvp);
			if (vp == nd.ni_dvp)
				vrele(vp);
			else
				vput(vp);
			if ((error = vn_start_write(NULL, &mp,
			    V_XSLEEP | PCATCH)) != 0)
				return (error);
			goto restart;
		}
#ifdef MAC
		error = mac_vnode_check_unlink(td->td_ucred, nd.ni_dvp, vp,
		    &nd.ni_cnd);
		if (error != 0)
			goto out;
#endif
		vfs_notify_upper(vp, VFS_NOTIFY_UPPER_UNLINK);
		error = VOP_REMOVE(nd.ni_dvp, vp, &nd.ni_cnd);
#ifdef MAC
out:
#endif
		vn_finished_write(mp);
	}
	NDFREE(&nd, NDF_ONLY_PNBUF);
	vput(nd.ni_dvp);
	if (vp == nd.ni_dvp)
		vrele(vp);
	else
		vput(vp);
	return (error);
}

/*
 * Reposition read/write file offset.
 */
#ifndef _SYS_SYSPROTO_H_
struct lseek_args {
	int	fd;
	int	pad;
	off_t	offset;
	int	whence;
};
#endif
int
sys_lseek(struct thread *td, struct lseek_args *uap)
{

	return (kern_lseek(td, uap->fd, uap->offset, uap->whence));
}

int
kern_lseek(struct thread *td, int fd, off_t offset, int whence)
{
	struct file *fp;
	cap_rights_t rights;
	int error;

	AUDIT_ARG_FD(fd);
	error = fget(td, fd, cap_rights_init(&rights, CAP_SEEK), &fp);
	if (error != 0)
		return (error);
	error = (fp->f_ops->fo_flags & DFLAG_SEEKABLE) != 0 ?
	    fo_seek(fp, offset, whence, td) : ESPIPE;
	fdrop(fp, td);
	return (error);
}

#if defined(COMPAT_43)
/*
 * Reposition read/write file offset.
 */
#ifndef _SYS_SYSPROTO_H_
struct olseek_args {
	int	fd;
	long	offset;
	int	whence;
};
#endif
int
olseek(struct thread *td, struct olseek_args *uap)
{

```

```c
	return (kern_lseek(td, uap->fd, uap->offset, uap->whence));
}
#endif /* COMPAT_43 */

#if defined(COMPAT_FREEBSD6)
/* Version with the 'pad' argument */
int
freebsd6_lseek(struct thread *td, struct freebsd6_lseek_args *uap)
{

	return (kern_lseek(td, uap->fd, uap->offset, uap->whence));
}
#endif

/*
 * Check access permissions using passed credentials.
 */
static int
vn_access(struct vnode *vp, int user_flags, struct ucred *cred,
     struct thread *td)
{
	accmode_t accmode;
	int error;

	/* Flags == 0 means only check for existence. */
	if (user_flags == 0)
		return (0);

	accmode = 0;
	if (user_flags & R_OK)
		accmode |= VREAD;
	if (user_flags & W_OK)
		accmode |= VWRITE;
	if (user_flags & X_OK)
		accmode |= VEXEC;
#ifdef MAC
	error = mac_vnode_check_access(cred, vp, accmode);
	if (error != 0)
		return (error);
#endif
	if ((accmode & VWRITE) == 0 || (error = vn_writechk(vp)) == 0)
		error = VOP_ACCESS(vp, accmode, cred, td);
	return (error);
}

/*
 * Check access permissions using "real" credentials.
 */
#ifndef _SYS_SYSPROTO_H_
struct access_args {
	char	*path;
	int	amode;
};
#endif
int
sys_access(struct thread *td, struct access_args *uap)
{

	return (kern_accessat(td, AT_FDCWD, uap->path, UIO_USERSPACE,
	    0, uap->amode));
}

#ifndef _SYS_SYSPROTO_H_
struct faccessat_args {
	int	dirfd;
	char	*path;
	int	amode;
	int	flag;
}
#endif
int
sys_faccessat(struct thread *td, struct faccessat_args *uap)
{

	return (kern_accessat(td, uap->fd, uap->path, UIO_USERSPACE, uap->flag,
	    uap->amode));
}

int
kern_accessat(struct thread *td, int fd, char *path, enum uio_seg pathseg,
    int flag, int amode)
{
	struct ucred *cred, *usecred;
	struct vnode *vp;
	struct nameidata nd;
	cap_rights_t rights;
	int error;

	if (flag & ~AT_EACCESS)
		return (EINVAL);
	if (amode != F_OK && (amode & ~(R_OK | W_OK | X_OK)) != 0)
		return (EINVAL);

	/*
	 * Create and modify a temporary credential instead of one that
	 * is potentially shared (if we need one).
	 */
	cred = td->td_ucred;
	if ((flag & AT_EACCESS) == 0 &&
	    ((cred->cr_uid != cred->cr_ruid ||
	    cred->cr_rgid != cred->cr_groups[0]))) {
		usecred = crdup(cred);
		usecred->cr_uid = cred->cr_ruid;
		usecred->cr_groups[0] = cred->cr_rgid;
		td->td_ucred = usecred;
	} else
		usecred = cred;
	AUDIT_ARG_VALUE(amode);
	NDINIT_ATRIGHTS(&nd, LOOKUP, FOLLOW | LOCKSHARED | LOCKLEAF |
	    AUDITVNODE1, pathseg, path, fd, cap_rights_init(&rights, CAP_FSTAT),
	    td);
	if ((error = namei(&nd)) != 0)
		goto out;
	vp = nd.ni_vp;

	error = vn_access(vp, amode, usecred, td);
	NDFREE(&nd, NDF_ONLY_PNBUF);
	vput(vp);
out:
	if (usecred != cred) {
		td->td_ucred = cred;
		crfree(usecred);
	}
	return (error);
}

/*
 * Check access permissions using "effective" credentials.
 */
#ifndef _SYS_SYSPROTO_H_
struct eaccess_args {
	char	*path;
	int	amode;
};
#endif
int
sys_eaccess(struct thread *td, struct eaccess_args *uap)
{

	return (kern_accessat(td, AT_FDCWD, uap->path, UIO_USERSPACE,
	    AT_EACCESS, uap->amode));
}

#if defined(COMPAT_43)
/*
 * Get file status; this version follows links.
 */
#ifndef _SYS_SYSPROTO_H_
struct ostat_args {
	char	*path;
	struct ostat *ub;
};
#endif
int
ostat(struct thread *td, struct ostat_args *uap)
{
	struct stat sb;
	struct ostat osb;
	int error;

	error = kern_statat(td, 0, AT_FDCWD, uap->path, UIO_USERSPACE,
	    &sb, NULL);
	if (error != 0)
		return (error);
	cvtstat(&sb, &osb);
	return (copyout(&osb, uap->ub, sizeof (osb)));
}

/*
 * Get file status; this version does not follow links.
 */
#ifndef _SYS_SYSPROTO_H_
struct olstat_args {
	char	*path;
	struct ostat *ub;
};
#endif
int
olstat(struct thread *td, struct olstat_args *uap)
{
	struct stat sb;
	struct ostat osb;
	int error;

	error = kern_statat(td, AT_SYMLINK_NOFOLLOW, AT_FDCWD, uap->path,
	    UIO_USERSPACE, &sb, NULL);
	if (error != 0)
		return (error);
	cvtstat(&sb, &osb);
	return (copyout(&osb, uap->ub, sizeof (osb)));
}

/*
 * Convert from an old to a new stat structure.
 */
void
```

```
cvtstat(struct stat *st, struct ostat *ost)
{

	bzero(ost, sizeof(*ost));
	ost->st_dev = st->st_dev;
	ost->st_ino = st->st_ino;
	ost->st_mode = st->st_mode;
	ost->st_nlink = st->st_nlink;
	ost->st_uid = st->st_uid;
	ost->st_gid = st->st_gid;
	ost->st_rdev = st->st_rdev;
	if (st->st_size < (quad_t)1 << 32)
		ost->st_size = st->st_size;
	else
		ost->st_size = -2;
	ost->st_atim = st->st_atim;
	ost->st_mtim = st->st_mtim;
	ost->st_ctim = st->st_ctim;
	ost->st_blksize = st->st_blksize;
	ost->st_blocks = st->st_blocks;
	ost->st_flags = st->st_flags;
	ost->st_gen = st->st_gen;
}
#endif /* COMPAT_43 */

/*
 * Get file status; this version follows links.
 */
#ifndef _SYS_SYSPROTO_H_
struct stat_args {
	char	*path;
	struct stat *ub;
};
#endif
int
sys_stat(struct thread *td, struct stat_args *uap)
{
	struct stat sb;
	int error;

	error = kern_statat(td, 0, AT_FDCWD, uap->path, UIO_USERSPACE,
	    &sb, NULL);
	if (error == 0)
		error = copyout(&sb, uap->ub, sizeof (sb));
	return (error);
}

#ifndef _SYS_SYSPROTO_H_
struct fstatat_args {
	int	fd;
	char	*path;
	struct stat	*buf;
	int	flag;
}
#endif
int
sys_fstatat(struct thread *td, struct fstatat_args *uap)
{
	struct stat sb;
	int error;

	error = kern_statat(td, uap->flag, uap->fd, uap->path,
	    UIO_USERSPACE, &sb, NULL);
	if (error == 0)
		error = copyout(&sb, uap->buf, sizeof (sb));
	return (error);
}

int
kern_statat(struct thread *td, int flag, int fd, char *path,
    enum uio_seg pathseg, struct stat *sbp,
    void (*hook)(struct vnode *vp, struct stat *sbp))
{
	struct nameidata nd;
	struct stat sb;
	cap_rights_t rights;
	int error;

	if (flag & ~AT_SYMLINK_NOFOLLOW)
		return (EINVAL);

	NDINIT_ATRIGHTS(&nd, LOOKUP, ((flag & AT_SYMLINK_NOFOLLOW) ? NOFOLLOW :
	    FOLLOW) | LOCKSHARED | LOCKLEAF | AUDITVNODE1, pathseg, path, fd,
	    cap_rights_init(&rights, CAP_FSTAT), td);

	if ((error = namei(&nd)) != 0)
		return (error);
	error = vn_stat(nd.ni_vp, &sb, td->td_ucred, NOCRED, td);
	if (error == 0) {
		SDT_PROBE2(vfs, , stat, mode, path, sb.st_mode);
		if (S_ISREG(sb.st_mode))
			SDT_PROBE2(vfs, , stat, reg, path, pathseg);
		if (__predict_false(hook != NULL))
			hook(nd.ni_vp, &sb);
	}
	NDFREE(&nd, NDF_ONLY_PNBUF);
	vput(nd.ni_vp);
	if (error != 0)
```

```
		return (error);
	*sbp = sb;
#ifdef KTRACE
	if (KTRPOINT(td, KTR_STRUCT))
		ktrstat(&sb);
#endif
	return (0);
}

/*
 * Get file status; this version does not follow links.
 */
#ifndef _SYS_SYSPROTO_H_
struct lstat_args {
	char	*path;
	struct stat *ub;
};
#endif
int
sys_lstat(struct thread *td, struct lstat_args *uap)
{
	struct stat sb;
	int error;

	error = kern_statat(td, AT_SYMLINK_NOFOLLOW, AT_FDCWD, uap->path,
	    UIO_USERSPACE, &sb, NULL);
	if (error == 0)
		error = copyout(&sb, uap->ub, sizeof (sb));
	return (error);
}

/*
 * Implementation of the NetBSD [l]stat() functions.
 */
void
cvtnstat( struct stat *sb, struct nstat *nsb)
{

	bzero(nsb, sizeof *nsb);
	nsb->st_dev = sb->st_dev;
	nsb->st_ino = sb->st_ino;
	nsb->st_mode = sb->st_mode;
	nsb->st_nlink = sb->st_nlink;
	nsb->st_uid = sb->st_uid;
	nsb->st_gid = sb->st_gid;
	nsb->st_rdev = sb->st_rdev;
	nsb->st_atim = sb->st_atim;
	nsb->st_mtim = sb->st_mtim;
	nsb->st_ctim = sb->st_ctim;
	nsb->st_size = sb->st_size;
	nsb->st_blocks = sb->st_blocks;
	nsb->st_blksize = sb->st_blksize;
	nsb->st_flags = sb->st_flags;
	nsb->st_gen = sb->st_gen;
	nsb->st_birthtim = sb->st_birthtim;
}

#ifndef _SYS_SYSPROTO_H_
struct nstat_args {
	char	*path;
	struct nstat *ub;
};
#endif
int
sys_nstat(struct thread *td, struct nstat_args *uap)
{
	struct stat sb;
	struct nstat nsb;
	int error;

	error = kern_statat(td, 0, AT_FDCWD, uap->path, UIO_USERSPACE,
	    &sb, NULL);
	if (error != 0)
		return (error);
	cvtnstat(&sb, &nsb);
	return (copyout(&nsb, uap->ub, sizeof (nsb)));
}

/*
 * NetBSD lstat.  Get file status; this version does not follow links.
 */
#ifndef _SYS_SYSPROTO_H_
struct lstat_args {
	char	*path;
	struct stat *ub;
};
#endif
int
sys_nlstat(struct thread *td, struct nlstat_args *uap)
{
	struct stat sb;
	struct nstat nsb;
	int error;

	error = kern_statat(td, AT_SYMLINK_NOFOLLOW, AT_FDCWD, uap->path,
	    UIO_USERSPACE, &sb, NULL);
	if (error != 0)
		return (error);
	cvtnstat(&sb, &nsb);
	return (copyout(&nsb, uap->ub, sizeof (nsb)));
}

/*
 * Get configurable pathname variables.
 */
#ifndef _SYS_SYSPROTO_H_
struct pathconf_args {
	char	*path;
	int	name;
};
#endif
int
sys_pathconf(struct thread *td, struct pathconf_args *uap)
{

	return (kern_pathconf(td, uap->path, UIO_USERSPACE, uap->name, FOLLOW));
}

#ifndef _SYS_SYSPROTO_H_
struct lpathconf_args {
	char	*path;
	int	name;
};
#endif
int
sys_lpathconf(struct thread *td, struct lpathconf_args *uap)
{

	return (kern_pathconf(td, uap->path, UIO_USERSPACE, uap->name,
	    NOFOLLOW));
}

int
kern_pathconf(struct thread *td, char *path, enum uio_seg pathseg, int name,
    u_long flags)
{
	struct nameidata nd;
	int error;

	NDINIT(&nd, LOOKUP, LOCKSHARED | LOCKLEAF | AUDITVNODE1 | flags,
	    pathseg, path, td);
	if ((error = namei(&nd)) != 0)
		return (error);
	NDFREE(&nd, NDF_ONLY_PNBUF);

	error = VOP_PATHCONF(nd.ni_vp, name, td->td_retval);
	vput(nd.ni_vp);
	return (error);
}

/*
 * Return target name of a symbolic link.
 */
#ifndef _SYS_SYSPROTO_H_
struct readlink_args {
	char	*path;
	char	*buf;
	size_t	count;
};
#endif
int
sys_readlink(struct thread *td, struct readlink_args *uap)
{

	return (kern_readlinkat(td, AT_FDCWD, uap->path, UIO_USERSPACE,
	    uap->buf, UIO_USERSPACE, uap->count));
}
#ifndef _SYS_SYSPROTO_H_
struct readlinkat_args {
	int	fd;
	char	*path;
	char	*buf;
	size_t	bufsize;
};
#endif
int
sys_readlinkat(struct thread *td, struct readlinkat_args *uap)
{

	return (kern_readlinkat(td, uap->fd, uap->path, UIO_USERSPACE,
	    uap->buf, UIO_USERSPACE, uap->bufsize));
}

int
kern_readlinkat(struct thread *td, int fd, char *path, enum uio_seg pathseg,
    char *buf, enum uio_seg bufseg, size_t count)
{
	struct vnode *vp;
	struct iovec aiov;
	struct uio auio;
	struct nameidata nd;
	int error;

	if (count > IOSIZE_MAX)
		return (EINVAL);

	NDINIT_AT(&nd, LOOKUP, NOFOLLOW | LOCKSHARED | LOCKLEAF | AUDITVNODE1,
	    pathseg, path, fd, td);

	if ((error = namei(&nd)) != 0)
		return (error);
	NDFREE(&nd, NDF_ONLY_PNBUF);
	vp = nd.ni_vp;
#ifdef MAC
	error = mac_vnode_check_readlink(td->td_ucred, vp);
	if (error != 0) {
		vput(vp);
		return (error);
	}
#endif
	if (vp->v_type != VLNK && (vp->v_vflag & VV_READLINK) == 0)
		error = EINVAL;
	else {
		aiov.iov_base = buf;
		aiov.iov_len = count;
		auio.uio_iov = &aiov;
		auio.uio_iovcnt = 1;
		auio.uio_offset = 0;
		auio.uio_rw = UIO_READ;
		auio.uio_segflg = bufseg;
		auio.uio_td = td;
		auio.uio_resid = count;
		error = VOP_READLINK(vp, &auio, td->td_ucred);
		td->td_retval[0] = count - auio.uio_resid;
	}
	vput(vp);
	return (error);
}

/*
 * Common implementation code for chflags() and fchflags().
 */
static int
setfflags(struct thread *td, struct vnode *vp, u_long flags)
{
	struct mount *mp;
	struct vattr vattr;
	int error;

	/* We can't support the value matching VNOVAL. */
	if (flags == VNOVAL)
		return (EOPNOTSUPP);

	/*
	 * Prevent non-root users from setting flags on devices.  When
	 * a device is reused, users can retain ownership of the device
	 * if they are allowed to set flags and programs assume that
	 * chown can't fail when done as root.
	 */
	if (vp->v_type == VCHR || vp->v_type == VBLK) {
		error = priv_check(td, PRIV_VFS_CHFLAGS_DEV);
		if (error != 0)
			return (error);
	}

	if ((error = vn_start_write(vp, &mp, V_WAIT | PCATCH)) != 0)
		return (error);
	VATTR_NULL(&vattr);
	vattr.va_flags = flags;
	vn_lock(vp, LK_EXCLUSIVE | LK_RETRY);
#ifdef MAC
	error = mac_vnode_check_setflags(td->td_ucred, vp, vattr.va_flags);
	if (error == 0)
#endif
		error = VOP_SETATTR(vp, &vattr, td->td_ucred);
	VOP_UNLOCK(vp, 0);
	vn_finished_write(mp);
	return (error);
}

/*
 * Change flags of a file given a path name.
 */
#ifndef _SYS_SYSPROTO_H_
struct chflags_args {
	const char *path;
	u_long	flags;
};
#endif
int
sys_chflags(struct thread *td, struct chflags_args *uap)
{

	return (kern_chflagsat(td, AT_FDCWD, uap->path, UIO_USERSPACE,
	    uap->flags, 0));
}

#ifndef _SYS_SYSPROTO_H_
struct chflagsat_args {
	int	fd;
	const char *path;
	u_long	flags;
	int	atflag;
}
```

```
#endif
int
sys_chflagsat(struct thread *td, struct chflagsat_args *uap)
{
        int fd = uap->fd;
        const char *path = uap->path;
        u_long flags = uap->flags;
        int atflag = uap->atflag;

        if (atflag & ~AT_SYMLINK_NOFOLLOW)
                return (EINVAL);

        return (kern_chflagsat(td, fd, path, UIO_USERSPACE, flags, atflag));
}

/*
 * Same as chflags() but doesn't follow symlinks.
 */
#ifndef _SYS_SYSPROTO_H_
struct lchflags_args {
        const char *path;
        u_long flags;
};
#endif
int
sys_lchflags(struct thread *td, struct lchflags_args *uap)
{

        return (kern_chflagsat(td, AT_FDCWD, uap->path, UIO_USERSPACE,
            uap->flags, AT_SYMLINK_NOFOLLOW));
}

static int
kern_chflagsat(struct thread *td, int fd, const char *path,
    enum uio_seg pathseg, u_long flags, int atflag)
{
        struct nameidata nd;
        cap_rights_t rights;
        int error, follow;

        AUDIT_ARG_FFLAGS(flags);
        follow = (atflag & AT_SYMLINK_NOFOLLOW) ? NOFOLLOW : FOLLOW;
        NDINIT_ATRIGHTS(&nd, LOOKUP, follow | AUDITVNODE1, pathseg, path, fd,
            cap_rights_init(&rights, CAP_FCHFLAGS), td);
        if ((error = namei(&nd)) != 0)
                return (error);
        NDFREE(&nd, NDF_ONLY_PNBUF);
        error = setfflags(td, nd.ni_vp, flags);
        vrele(nd.ni_vp);
        return (error);
}

/*
 * Change flags of a file given a file descriptor.
 */
#ifndef _SYS_SYSPROTO_H_
struct fchflags_args {
        int     fd;
        u_long  flags;
};
#endif
int
sys_fchflags(struct thread *td, struct fchflags_args *uap)
{
        struct file *fp;
        cap_rights_t rights;
        int error;

        AUDIT_ARG_FD(uap->fd);
        AUDIT_ARG_FFLAGS(uap->flags);
        error = getvnode(td, uap->fd, cap_rights_init(&rights, CAP_FCHFLAGS),
            &fp);
        if (error != 0)
                return (error);
#ifdef AUDIT
        vn_lock(fp->f_vnode, LK_SHARED | LK_RETRY);
        AUDIT_ARG_VNODE1(fp->f_vnode);
        VOP_UNLOCK(fp->f_vnode, 0);
#endif
        error = setfflags(td, fp->f_vnode, uap->flags);
        fdrop(fp, td);
        return (error);
}

/*
 * Common implementation code for chmod(), lchmod() and fchmod().
 */
int
setfmode(struct thread *td, struct ucred *cred, struct vnode *vp, int mode)
{
        struct mount *mp;
        struct vattr vattr;
        int error;

        if ((error = vn_start_write(vp, &mp, V_WAIT | PCATCH)) != 0)
                return (error);
        vn_lock(vp, LK_EXCLUSIVE | LK_RETRY);
        VATTR_NULL(&vattr);
	vattr.va_mode = mode & ALLPERMS;
#ifdef MAC
	error = mac_vnode_check_setmode(cred, vp, vattr.va_mode);
	if (error == 0)
#endif
		error = VOP_SETATTR(vp, &vattr, cred);
	VOP_UNLOCK(vp, 0);
	vn_finished_write(mp);
	return (error);
}

/*
 * Change mode of a file given path name.
 */
#ifndef _SYS_SYSPROTO_H_
struct chmod_args {
	char	*path;
	int	mode;
};
#endif
int
sys_chmod(struct thread *td, struct chmod_args *uap)
{

	return (kern_fchmodat(td, AT_FDCWD, uap->path, UIO_USERSPACE,
	    uap->mode, 0));
}

#ifndef _SYS_SYSPROTO_H_
struct fchmodat_args {
	int	dirfd;
	char	*path;
	mode_t	mode;
	int	flag;
}
#endif
int
sys_fchmodat(struct thread *td, struct fchmodat_args *uap)
{
	int flag = uap->flag;
	int fd = uap->fd;
	char *path = uap->path;
	mode_t mode = uap->mode;

	if (flag & ~AT_SYMLINK_NOFOLLOW)
		return (EINVAL);

	return (kern_fchmodat(td, fd, path, UIO_USERSPACE, mode, flag));
}

/*
 * Change mode of a file given path name (don't follow links.)
 */
#ifndef _SYS_SYSPROTO_H_
struct lchmod_args {
	char	*path;
	int	mode;
};
#endif
int
sys_lchmod(struct thread *td, struct lchmod_args *uap)
{

	return (kern_fchmodat(td, AT_FDCWD, uap->path, UIO_USERSPACE,
	    uap->mode, AT_SYMLINK_NOFOLLOW));
}

int
kern_fchmodat(struct thread *td, int fd, char *path, enum uio_seg pathseg,
    mode_t mode, int flag)
{
	struct nameidata nd;
	cap_rights_t rights;
	int error, follow;

	AUDIT_ARG_MODE(mode);
	follow = (flag & AT_SYMLINK_NOFOLLOW) ? NOFOLLOW : FOLLOW;
	NDINIT_ATRIGHTS(&nd, LOOKUP, follow | AUDITVNODE1, pathseg, path, fd,
	    cap_rights_init(&rights, CAP_FCHMOD), td);
	if ((error = namei(&nd)) != 0)
		return (error);
	NDFREE(&nd, NDF_ONLY_PNBUF);
	error = setfmode(td, td->td_ucred, nd.ni_vp, mode);
	vrele(nd.ni_vp);
	return (error);
}

/*
 * Change mode of a file given a file descriptor.
 */
#ifndef _SYS_SYSPROTO_H_
struct fchmod_args {
	int	fd;
	int	mode;
};
#endif
int
sys_fchmod(struct thread *td, struct fchmod_args *uap)
```

```
{
	struct file *fp;
	cap_rights_t rights;
	int error;

	AUDIT_ARG_FD(uap->fd);
	AUDIT_ARG_MODE(uap->mode);

	error = fget(td, uap->fd, cap_rights_init(&rights, CAP_FCHMOD), &fp);
	if (error != 0)
		return (error);
	error = fo_chmod(fp, uap->mode, td->td_ucred, td);
	fdrop(fp, td);
	return (error);
}

/*
 * Common implementation for chown(), lchown(), and fchown()
 */
int
setfown(struct thread *td, struct ucred *cred, struct vnode *vp, uid_t uid,
    gid_t gid)
{
	struct mount *mp;
	struct vattr vattr;
	int error;

	if ((error = vn_start_write(vp, &mp, V_WAIT | PCATCH)) != 0)
		return (error);
	vn_lock(vp, LK_EXCLUSIVE | LK_RETRY);
	VATTR_NULL(&vattr);
	vattr.va_uid = uid;
	vattr.va_gid = gid;
#ifdef MAC
	error = mac_vnode_check_setowner(cred, vp, vattr.va_uid,
	    vattr.va_gid);
	if (error == 0)
#endif
		error = VOP_SETATTR(vp, &vattr, cred);
	VOP_UNLOCK(vp, 0);
	vn_finished_write(mp);
	return (error);
}

/*
 * Set ownership given a path name.
 */
#ifndef _SYS_SYSPROTO_H_
struct chown_args {
	char	*path;
	int	uid;
	int	gid;
};
#endif
int
sys_chown(struct thread *td, struct chown_args *uap)
{

	return (kern_fchownat(td, AT_FDCWD, uap->path, UIO_USERSPACE, uap->uid,
	    uap->gid, 0));
}

#ifndef _SYS_SYSPROTO_H_
struct fchownat_args {
	int fd;
	const char * path;
	uid_t uid;
	gid_t gid;
	int flag;
};
#endif
int
sys_fchownat(struct thread *td, struct fchownat_args *uap)
{
	int flag;

	flag = uap->flag;
	if (flag & ~AT_SYMLINK_NOFOLLOW)
		return (EINVAL);

	return (kern_fchownat(td, uap->fd, uap->path, UIO_USERSPACE, uap->uid,
	    uap->gid, uap->flag));
}

int
kern_fchownat(struct thread *td, int fd, char *path, enum uio_seg pathseg,
    int uid, int gid, int flag)
{
	struct nameidata nd;
	cap_rights_t rights;
	int error, follow;

	AUDIT_ARG_OWNER(uid, gid);
	follow = (flag & AT_SYMLINK_NOFOLLOW) ? NOFOLLOW : FOLLOW;
	NDINIT_ATRIGHTS(&nd, LOOKUP, follow | AUDITVNODE1, pathseg, path, fd,
	    cap_rights_init(&rights, CAP_FCHOWN), td);

	if ((error = namei(&nd)) != 0)
```

```
                return (error);
        NDFREE(&nd, NDF_ONLY_PNBUF);
        error = setfown(td, td->td_ucred, nd.ni_vp, uid, gid);
        vrele(nd.ni_vp);
        return (error);
}

/*
 * Set ownership given a path name, do not cross symlinks.
 */
#ifndef _SYS_SYSPROTO_H_
struct lchown_args {
        char    *path;
        int     uid;
        int     gid;
};
#endif
int
sys_lchown(struct thread *td, struct lchown_args *uap)
{

        return (kern_fchownat(td, AT_FDCWD, uap->path, UIO_USERSPACE,
            uap->uid, uap->gid, AT_SYMLINK_NOFOLLOW));
}

/*
 * Set ownership given a file descriptor.
 */
#ifndef _SYS_SYSPROTO_H_
struct fchown_args {
        int     fd;
        int     uid;
        int     gid;
};
#endif
int
sys_fchown(struct thread *td, struct fchown_args *uap)
{
        struct file *fp;
        cap_rights_t rights;
        int error;

        AUDIT_ARG_FD(uap->fd);
        AUDIT_ARG_OWNER(uap->uid, uap->gid);
        error = fget(td, uap->fd, cap_rights_init(&rights, CAP_FCHOWN), &fp);
        if (error != 0)
                return (error);
        error = fo_chown(fp, uap->uid, uap->gid, td->td_ucred, td);
        fdrop(fp, td);
        return (error);
}

/*
 * Common implementation code for utimes(), lutimes(), and futimes().
 */
static int
getutimes(const struct timeval *usrtvp, enum uio_seg tvpseg,
    struct timespec *tsp)
{
        struct timeval tv[2];
        const struct timeval *tvp;
        int error;

        if (usrtvp == NULL) {
                vfs_timestamp(&tsp[0]);
                tsp[1] = tsp[0];
        } else {
                if (tvpseg == UIO_SYSSPACE) {
                        tvp = usrtvp;
                } else {
                        if ((error = copyin(usrtvp, tv, sizeof(tv))) != 0)
                                return (error);
                        tvp = tv;
                }

                if (tvp[0].tv_usec < 0 || tvp[0].tv_usec >= 1000000 ||
                    tvp[1].tv_usec < 0 || tvp[1].tv_usec >= 1000000)
                        return (EINVAL);
                TIMEVAL_TO_TIMESPEC(&tvp[0], &tsp[0]);
                TIMEVAL_TO_TIMESPEC(&tvp[1], &tsp[1]);
        }
        return (0);
}

/*
 * Common implementation code for futimens(), utimensat().
 */
#define UTIMENS_NULL    0x1
#define UTIMENS_EXIT    0x2
static int
getutimens(const struct timespec *usrtsp, enum uio_seg tspseg,
    struct timespec *tsp, int *retflags)
{
        struct timespec tsnow;
        int error;

        vfs_timestamp(&tsnow);
        *retflags = 0;
	if (usrtsp == NULL) {
		tsp[0] = tsnow;
		tsp[1] = tsnow;
		*retflags |= UTIMENS_NULL;
		return (0);
	}
	if (tspseg == UIO_SYSSPACE) {
		tsp[0] = usrtsp[0];
		tsp[1] = usrtsp[1];
	} else if ((error = copyin(usrtsp, tsp, sizeof(*tsp) * 2)) != 0)
		return (error);
	if (tsp[0].tv_nsec == UTIME_OMIT && tsp[1].tv_nsec == UTIME_OMIT)
		*retflags |= UTIMENS_EXIT;
	if (tsp[0].tv_nsec == UTIME_NOW && tsp[1].tv_nsec == UTIME_NOW)
		*retflags |= UTIMENS_NULL;
	if (tsp[0].tv_nsec == UTIME_OMIT)
		tsp[0].tv_sec = VNOVAL;
	else if (tsp[0].tv_nsec == UTIME_NOW)
		tsp[0] = tsnow;
	else if (tsp[0].tv_nsec < 0 || tsp[0].tv_nsec >= 1000000000L)
		return (EINVAL);
	if (tsp[1].tv_nsec == UTIME_OMIT)
		tsp[1].tv_sec = VNOVAL;
	else if (tsp[1].tv_nsec == UTIME_NOW)
		tsp[1] = tsnow;
	else if (tsp[1].tv_nsec < 0 || tsp[1].tv_nsec >= 1000000000L)
		return (EINVAL);

	return (0);
}

/*
 * Common implementation code for utimes(), lutimes(), futimes(), futimens(),
 * and utimensat().
 */
static int
setutimes(struct thread *td, struct vnode *vp, const struct timespec *ts,
    int numtimes, int nullflag)
{
	struct mount *mp;
	struct vattr vattr;
	int error, setbirthtime;

	if ((error = vn_start_write(vp, &mp, V_WAIT | PCATCH)) != 0)
		return (error);
	vn_lock(vp, LK_EXCLUSIVE | LK_RETRY);
	setbirthtime = 0;
	if (numtimes < 3 && !VOP_GETATTR(vp, &vattr, td->td_ucred) &&
	    timespeccmp(&ts[1], &vattr.va_birthtime, < ))
		setbirthtime = 1;
	VATTR_NULL(&vattr);
	vattr.va_atime = ts[0];
	vattr.va_mtime = ts[1];
	if (setbirthtime)
		vattr.va_birthtime = ts[1];
	if (numtimes > 2)
		vattr.va_birthtime = ts[2];
	if (nullflag)
		vattr.va_vaflags |= VA_UTIMES_NULL;
#ifdef MAC
	error = mac_vnode_check_setutimes(td->td_ucred, vp, vattr.va_atime,
	    vattr.va_mtime);
#endif
	if (error == 0)
		error = VOP_SETATTR(vp, &vattr, td->td_ucred);
	VOP_UNLOCK(vp, 0);
	vn_finished_write(mp);
	return (error);
}

/*
 * Set the access and modification times of a file.
 */
#ifndef _SYS_SYSPROTO_H_
struct utimes_args {
	char	*path;
	struct	timeval *tptr;
};
#endif
int
sys_utimes(struct thread *td, struct utimes_args *uap)
{

	return (kern_utimesat(td, AT_FDCWD, uap->path, UIO_USERSPACE,
	    uap->tptr, UIO_USERSPACE));
}

#ifndef _SYS_SYSPROTO_H_
struct futimesat_args {
	int fd;
	const char * path;
	const struct timeval * times;
};
#endif
int
sys_futimesat(struct thread *td, struct futimesat_args *uap)
{

	return (kern_utimesat(td, uap->fd, uap->path, UIO_USERSPACE,
	    uap->times, UIO_USERSPACE));
}

int
kern_utimesat(struct thread *td, int fd, char *path, enum uio_seg pathseg,
    struct timeval *tptr, enum uio_seg tptrseg)
{
	struct nameidata nd;
	struct timespec ts[2];
	cap_rights_t rights;
	int error;

	if ((error = getutimes(tptr, tptrseg, ts)) != 0)
		return (error);
	NDINIT_ATRIGHTS(&nd, LOOKUP, FOLLOW | AUDITVNODE1, pathseg, path, fd,
	    cap_rights_init(&rights, CAP_FUTIMES), td);

	if ((error = namei(&nd)) != 0)
		return (error);
	NDFREE(&nd, NDF_ONLY_PNBUF);
	error = setutimes(td, nd.ni_vp, ts, 2, tptr == NULL);
	vrele(nd.ni_vp);
	return (error);
}

/*
 * Set the access and modification times of a file.
 */
#ifndef _SYS_SYSPROTO_H_
struct lutimes_args {
	char	*path;
	struct	timeval *tptr;
};
#endif
int
sys_lutimes(struct thread *td, struct lutimes_args *uap)
{

	return (kern_lutimes(td, uap->path, UIO_USERSPACE, uap->tptr,
	    UIO_USERSPACE));
}

int
kern_lutimes(struct thread *td, char *path, enum uio_seg pathseg,
    struct timeval *tptr, enum uio_seg tptrseg)
{
	struct timespec ts[2];
	struct nameidata nd;
	int error;

	if ((error = getutimes(tptr, tptrseg, ts)) != 0)
		return (error);
	NDINIT(&nd, LOOKUP, NOFOLLOW | AUDITVNODE1, pathseg, path, td);
	if ((error = namei(&nd)) != 0)
		return (error);
	NDFREE(&nd, NDF_ONLY_PNBUF);
	error = setutimes(td, nd.ni_vp, ts, 2, tptr == NULL);
	vrele(nd.ni_vp);
	return (error);
}

/*
 * Set the access and modification times of a file.
 */
#ifndef _SYS_SYSPROTO_H_
struct futimes_args {
	int	fd;
	struct	timeval *tptr;
};
#endif
int
sys_futimes(struct thread *td, struct futimes_args *uap)
{

	return (kern_futimes(td, uap->fd, uap->tptr, UIO_USERSPACE));
}

int
kern_futimes(struct thread *td, int fd, struct timeval *tptr,
    enum uio_seg tptrseg)
{
	struct timespec ts[2];
	struct file *fp;
	cap_rights_t rights;
	int error;

	AUDIT_ARG_FD(fd);
	error = getutimes(tptr, tptrseg, ts);
	if (error != 0)
		return (error);
	error = getvnode(td, fd, cap_rights_init(&rights, CAP_FUTIMES), &fp);
	if (error != 0)
		return (error);
#ifdef AUDIT
	vn_lock(fp->f_vnode, LK_SHARED | LK_RETRY);
	AUDIT_ARG_VNODE1(fp->f_vnode);
	VOP_UNLOCK(fp->f_vnode, 0);
```

```c
#endif
	error = setutimes(td, fp->f_vnode, ts, 2, tptr == NULL);
	fdrop(fp, td);
	return (error);
}

int
sys_futimens(struct thread *td, struct futimens_args *uap)
{

	return (kern_futimens(td, uap->fd, uap->times, UIO_USERSPACE));
}

int
kern_futimens(struct thread *td, int fd, struct timespec *tptr,
    enum uio_seg tptrseg)
{
	struct timespec ts[2];
	struct file *fp;
	cap_rights_t rights;
	int error, flags;

	AUDIT_ARG_FD(fd);
	error = getutimens(tptr, tptrseg, ts, &flags);
	if (error != 0)
		return (error);
	if (flags & UTIMENS_EXIT)
		return (0);
	error = getvnode(td, fd, cap_rights_init(&rights, CAP_FUTIMES), &fp);
	if (error != 0)
		return (error);
#ifdef AUDIT
	vn_lock(fp->f_vnode, LK_SHARED | LK_RETRY);
	AUDIT_ARG_VNODE1(fp->f_vnode);
	VOP_UNLOCK(fp->f_vnode, 0);
#endif
	error = setutimes(td, fp->f_vnode, ts, 2, flags & UTIMENS_NULL);
	fdrop(fp, td);
	return (error);
}

int
sys_utimensat(struct thread *td, struct utimensat_args *uap)
{

	return (kern_utimensat(td, uap->fd, uap->path, UIO_USERSPACE,
	    uap->times, UIO_USERSPACE, uap->flag));
}

int
kern_utimensat(struct thread *td, int fd, char *path, enum uio_seg pathseg,
    struct timespec *tptr, enum uio_seg tptrseg, int flag)
{
	struct nameidata nd;
	struct timespec ts[2];
	cap_rights_t rights;
	int error, flags;

	if (flag & ~AT_SYMLINK_NOFOLLOW)
		return (EINVAL);

	if ((error = getutimens(tptr, tptrseg, ts, &flags)) != 0)
		return (error);
	NDINIT_ATRIGHTS(&nd, LOOKUP, ((flag & AT_SYMLINK_NOFOLLOW) ? NOFOLLOW :
	    FOLLOW) | AUDITVNODE1, pathseg, path, fd,
	    cap_rights_init(&rights, CAP_FUTIMES), td);
	if ((error = namei(&nd)) != 0)
		return (error);
	/*
	 * We are allowed to call namei() regardless of 2xUTIME_OMIT.
	 * POSIX states:
	 * "If both tv_nsec fields are UTIME_OMIT... EACCESS may be detected."
	 * "Search permission is denied by a component of the path prefix."
	 */
	NDFREE(&nd, NDF_ONLY_PNBUF);
	if ((flags & UTIMENS_EXIT) == 0)
		error = setutimes(td, nd.ni_vp, ts, 2, flags & UTIMENS_NULL);
	vrele(nd.ni_vp);
	return (error);
}

/*
 * Truncate a file given its path name.
 */
#ifndef _SYS_SYSPROTO_H_
struct truncate_args {
	char	*path;
	int	pad;
	off_t	length;
};
#endif
int
sys_truncate(struct thread *td, struct truncate_args *uap)
{

	return (kern_truncate(td, uap->path, UIO_USERSPACE, uap->length));
}

```

```c
int
kern_truncate(struct thread *td, char *path, enum uio_seg pathseg, off_t length)
{
	struct mount *mp;
	struct vnode *vp;
	void *rl_cookie;
	struct vattr vattr;
	struct nameidata nd;
	int error;

	if (length < 0)
		return(EINVAL);
	NDINIT(&nd, LOOKUP, FOLLOW | AUDITVNODE1, pathseg, path, td);
	if ((error = namei(&nd)) != 0)
		return (error);
	vp = nd.ni_vp;
	rl_cookie = vn_rangelock_wlock(vp, 0, OFF_MAX);
	if ((error = vn_start_write(vp, &mp, V_WAIT | PCATCH)) != 0) {
		vn_rangelock_unlock(vp, rl_cookie);
		vrele(vp);
		return (error);
	}
	NDFREE(&nd, NDF_ONLY_PNBUF);
	vn_lock(vp, LK_EXCLUSIVE | LK_RETRY);
	if (vp->v_type == VDIR)
		error = EISDIR;
#ifdef MAC
	else if ((error = mac_vnode_check_write(td->td_ucred, NOCRED, vp))) {
	}
#endif
	else if ((error = vn_writechk(vp)) == 0 &&
	    (error = VOP_ACCESS(vp, VWRITE, td->td_ucred, td)) == 0) {
		VATTR_NULL(&vattr);
		vattr.va_size = length;
		error = VOP_SETATTR(vp, &vattr, td->td_ucred);
	}
	VOP_UNLOCK(vp, 0);
	vn_finished_write(mp);
	vn_rangelock_unlock(vp, rl_cookie);
	vrele(vp);
	return (error);
}

#if defined(COMPAT_43)
/*
 * Truncate a file given its path name.
 */
#ifndef _SYS_SYSPROTO_H_
struct otruncate_args {
	char	*path;
	long	length;
};
#endif
int
otruncate(struct thread *td, struct otruncate_args *uap)
{

	return (kern_truncate(td, uap->path, UIO_USERSPACE, uap->length));
}
#endif /* COMPAT_43 */

#if defined(COMPAT_FREEBSD6)
/* Versions with the pad argument */
int
freebsd6_truncate(struct thread *td, struct freebsd6_truncate_args *uap)
{

	return (kern_truncate(td, uap->path, UIO_USERSPACE, uap->length));
}

int
freebsd6_ftruncate(struct thread *td, struct freebsd6_ftruncate_args *uap)
{

	return (kern_ftruncate(td, uap->fd, uap->length));
}
#endif

int
kern_fsync(struct thread *td, int fd, bool fullsync)
{
	struct vnode *vp;
	struct mount *mp;
	struct file *fp;
	cap_rights_t rights;
	int error, lock_flags;

	AUDIT_ARG_FD(fd);
	error = getvnode(td, fd, cap_rights_init(&rights, CAP_FSYNC), &fp);
	if (error != 0)
		return (error);
	vp = fp->f_vnode;
#if 0
	if (!fullsync)
		/* XXXKIB: compete outstanding aio writes */;
#endif
	error = vn_start_write(vp, &mp, V_WAIT | PCATCH);
	if (error != 0)
```

```c
		goto drop;
	if (MNT_SHARED_WRITES(mp) ||
	    ((mp == NULL) && MNT_SHARED_WRITES(vp->v_mount))) {
		lock_flags = LK_SHARED;
	} else {
		lock_flags = LK_EXCLUSIVE;
	}
	vn_lock(vp, lock_flags | LK_RETRY);
	AUDIT_ARG_VNODE1(vp);
	if (vp->v_object != NULL) {
		VM_OBJECT_WLOCK(vp->v_object);
		vm_object_page_clean(vp->v_object, 0, 0, 0);
		VM_OBJECT_WUNLOCK(vp->v_object);
	}
	error = fullsync ? VOP_FSYNC(vp, MNT_WAIT, td) : VOP_FDATASYNC(vp, td);
	VOP_UNLOCK(vp, 0);
	vn_finished_write(mp);
drop:
	fdrop(fp, td);
	return (error);
}

/*
 * Sync an open file.
 */
#ifndef _SYS_SYSPROTO_H_
struct fsync_args {
	int	fd;
};
#endif
int
sys_fsync(struct thread *td, struct fsync_args *uap)
{

	return (kern_fsync(td, uap->fd, true));
}

int
sys_fdatasync(struct thread *td, struct fdatasync_args *uap)
{

	return (kern_fsync(td, uap->fd, false));
}

/*
 * Rename files.  Source and destination must either both be directories, or
 * both not be directories.  If target is a directory, it must be empty.
 */
#ifndef _SYS_SYSPROTO_H_
struct rename_args {
	char	*from;
	char	*to;
};
#endif
int
sys_rename(struct thread *td, struct rename_args *uap)
{

	return (kern_renameat(td, AT_FDCWD, uap->from, AT_FDCWD,
	    uap->to, UIO_USERSPACE));
}

#ifndef _SYS_SYSPROTO_H_
struct renameat_args {
	int	oldfd;
	char	*old;
	int	newfd;
	char	*new;
};
#endif
int
sys_renameat(struct thread *td, struct renameat_args *uap)
{

	return (kern_renameat(td, uap->oldfd, uap->old, uap->newfd, uap->new,
	    UIO_USERSPACE));
}

int
kern_renameat(struct thread *td, int oldfd, char *old, int newfd, char *new,
    enum uio_seg pathseg)
{
	struct mount *mp = NULL;
	struct vnode *tvp, *fvp, *tdvp;
	struct nameidata fromnd, tond;
	cap_rights_t rights;
	int error;

again:
	bwillwrite();
#ifdef MAC
	NDINIT_ATRIGHTS(&fromnd, DELETE, LOCKPARENT | LOCKLEAF | SAVESTART |
	    AUDITVNODE1, pathseg, old, oldfd,
	    cap_rights_init(&rights, CAP_RENAMEAT_SOURCE), td);
#else
	NDINIT_ATRIGHTS(&fromnd, DELETE, WANTPARENT | SAVESTART | AUDITVNODE1,
	    pathseg, old, oldfd,
	    cap_rights_init(&rights, CAP_RENAMEAT_SOURCE), td);
```

```c
#endif

	if ((error = namei(&fromnd)) != 0)
		return (error);
#ifdef MAC
	error = mac_vnode_check_rename_from(td->td_ucred, fromnd.ni_dvp,
	    fromnd.ni_vp, &fromnd.ni_cnd);
	VOP_UNLOCK(fromnd.ni_dvp, 0);
	if (fromnd.ni_dvp != fromnd.ni_vp)
		VOP_UNLOCK(fromnd.ni_vp, 0);
#endif
	fvp = fromnd.ni_vp;
	NDINIT_ATRIGHTS(&tond, RENAME, LOCKPARENT | LOCKLEAF | NOCACHE |
	    SAVESTART | AUDITVNODE2, pathseg, new, newfd,
	    cap_rights_init(&rights, CAP_RENAMEAT_TARGET), td);
	if (fromnd.ni_vp->v_type == VDIR)
		tond.ni_cnd.cn_flags |= WILLBEDIR;
	if ((error = namei(&tond)) != 0) {
		/* Translate error code for rename("dir1", "dir2/."). */
		if (error == EISDIR && fvp->v_type == VDIR)
			error = EINVAL;
		NDFREE(&fromnd, NDF_ONLY_PNBUF);
		vrele(fromnd.ni_dvp);
		vrele(fvp);
		goto out1;
	}
	tdvp = tond.ni_dvp;
	tvp = tond.ni_vp;
	error = vn_start_write(fvp, &mp, V_NOWAIT);
	if (error != 0) {
		NDFREE(&fromnd, NDF_ONLY_PNBUF);
		NDFREE(&tond, NDF_ONLY_PNBUF);
		if (tvp != NULL)
			vput(tvp);
		if (tdvp == tvp)
			vrele(tdvp);
		else
			vput(tdvp);
		vrele(fromnd.ni_dvp);
		vrele(fvp);
		vrele(tond.ni_startdir);
		if (fromnd.ni_startdir != NULL)
			vrele(fromnd.ni_startdir);
		error = vn_start_write(NULL, &mp, V_XSLEEP | PCATCH);
		if (error != 0)
			return (error);
		goto again;
	}
	if (tvp != NULL) {
		if (fvp->v_type == VDIR && tvp->v_type != VDIR) {
			error = ENOTDIR;
			goto out;
		} else if (fvp->v_type != VDIR && tvp->v_type == VDIR) {
			error = EISDIR;
			goto out;
		}
#ifdef CAPABILITIES
		if (newfd != AT_FDCWD) {
			/*
			 * If the target already exists we require CAP_UNLINKAT
			 * from 'newfd'.
			 */
			error = cap_check(&tond.ni_filecaps.fc_rights,
			    cap_rights_init(&rights, CAP_UNLINKAT));
			if (error != 0)
				goto out;
		}
#endif
	}
	if (fvp == tdvp) {
		error = EINVAL;
		goto out;
	}
	/*
	 * If the source is the same as the destination (that is, if they
	 * are links to the same vnode), then there is nothing to do.
	 */
	if (fvp == tvp)
		error = -1;
#ifdef MAC
	else
		error = mac_vnode_check_rename_to(td->td_ucred, tdvp,
		    tond.ni_vp, fromnd.ni_dvp == tdvp, &tond.ni_cnd);
#endif
out:
	if (error == 0) {
		error = VOP_RENAME(fromnd.ni_dvp, fromnd.ni_vp, &fromnd.ni_cnd,
		    tond.ni_dvp, tond.ni_vp, &tond.ni_cnd);
		NDFREE(&fromnd, NDF_ONLY_PNBUF);
		NDFREE(&tond, NDF_ONLY_PNBUF);
	} else {
		NDFREE(&fromnd, NDF_ONLY_PNBUF);
		NDFREE(&tond, NDF_ONLY_PNBUF);
		if (tvp != NULL)
			vput(tvp);
		if (tdvp == tvp)
			vrele(tdvp);
		else
```

```c
				vput(tdvp);
			vrele(fromnd.ni_dvp);
			vrele(fvp);
	}
	vrele(tond.ni_startdir);
	vn_finished_write(mp);
out1:
	if (fromnd.ni_startdir)
		vrele(fromnd.ni_startdir);
	if (error == -1)
		return (0);
	return (error);
}

/*
 * Make a directory file.
 */
#ifndef _SYS_SYSPROTO_H_
struct mkdir_args {
	char	*path;
	int	mode;
};
#endif
int
sys_mkdir(struct thread *td, struct mkdir_args *uap)
{

	return (kern_mkdirat(td, AT_FDCWD, uap->path, UIO_USERSPACE,
	    uap->mode));
}

#ifndef _SYS_SYSPROTO_H_
struct mkdirat_args {
	int	fd;
	char	*path;
	mode_t	mode;
};
#endif
int
sys_mkdirat(struct thread *td, struct mkdirat_args *uap)
{

	return (kern_mkdirat(td, uap->fd, uap->path, UIO_USERSPACE, uap->mode));
}

int
kern_mkdirat(struct thread *td, int fd, char *path, enum uio_seg segflg,
    int mode)
{
	struct mount *mp;
	struct vnode *vp;
	struct vattr vattr;
	struct nameidata nd;
	cap_rights_t rights;
	int error;

	AUDIT_ARG_MODE(mode);
restart:
	bwillwrite();
	NDINIT_ATRIGHTS(&nd, CREATE, LOCKPARENT | SAVENAME | AUDITVNODE1 |
	    NOCACHE, segflg, path, fd, cap_rights_init(&rights, CAP_MKDIRAT),
	    td);
	nd.ni_cnd.cn_flags |= WILLBEDIR;
	if ((error = namei(&nd)) != 0)
		return (error);
	vp = nd.ni_vp;
	if (vp != NULL) {
		NDFREE(&nd, NDF_ONLY_PNBUF);
		/*
		 * XXX namei called with LOCKPARENT but not LOCKLEAF has
		 * the strange behaviour of leaving the vnode unlocked
		 * if the target is the same vnode as the parent.
		 */
		if (vp == nd.ni_dvp)
			vrele(nd.ni_dvp);
		else
			vput(nd.ni_dvp);
		vrele(vp);
		return (EEXIST);
	}
	if (vn_start_write(nd.ni_dvp, &mp, V_NOWAIT) != 0) {
		NDFREE(&nd, NDF_ONLY_PNBUF);
		vput(nd.ni_dvp);
		if ((error = vn_start_write(NULL, &mp, V_XSLEEP | PCATCH)) != 0)
			return (error);
		goto restart;
	}
	VATTR_NULL(&vattr);
	vattr.va_type = VDIR;
	vattr.va_mode = (mode & ACCESSPERMS) &~ td->td_proc->p_fd->fd_cmask;
#ifdef MAC
	error = mac_vnode_check_create(td->td_ucred, nd.ni_dvp, &nd.ni_cnd,
	    &vattr);
	if (error != 0)
		goto out;
#endif
	error = VOP_MKDIR(nd.ni_dvp, &nd.ni_vp, &nd.ni_cnd, &vattr);
#ifdef MAC
```

```
out:
#endif
	NDFREE(&nd, NDF_ONLY_PNBUF);
	vput(nd.ni_dvp);
	if (error == 0)
		vput(nd.ni_vp);
	vn_finished_write(mp);
	return (error);
}

/*
 * Remove a directory file.
 */
#ifndef _SYS_SYSPROTO_H_
struct rmdir_args {
	char	*path;
};
#endif
int
sys_rmdir(struct thread *td, struct rmdir_args *uap)
{

	return (kern_rmdirat(td, AT_FDCWD, uap->path, UIO_USERSPACE));
}

int
kern_rmdirat(struct thread *td, int fd, char *path, enum uio_seg pathseg)
{
	struct mount *mp;
	struct vnode *vp;
	struct nameidata nd;
	cap_rights_t rights;
	int error;

restart:
	bwillwrite();
	NDINIT_ATRIGHTS(&nd, DELETE, LOCKPARENT | LOCKLEAF | AUDITVNODE1,
	    pathseg, path, fd, cap_rights_init(&rights, CAP_UNLINKAT), td);
	if ((error = namei(&nd)) != 0)
		return (error);
	vp = nd.ni_vp;
	if (vp->v_type != VDIR) {
		error = ENOTDIR;
		goto out;
	}
	/*
	 * No rmdir "." please.
	 */
	if (nd.ni_dvp == vp) {
		error = EINVAL;
		goto out;
	}
	/*
	 * The root of a mounted filesystem cannot be deleted.
	 */
	if (vp->v_vflag & VV_ROOT) {
		error = EBUSY;
		goto out;
	}
#ifdef MAC
	error = mac_vnode_check_unlink(td->td_ucred, nd.ni_dvp, vp,
	    &nd.ni_cnd);
	if (error != 0)
		goto out;
#endif
	if (vn_start_write(nd.ni_dvp, &mp, V_NOWAIT) != 0) {
		NDFREE(&nd, NDF_ONLY_PNBUF);
		vput(vp);
		if (nd.ni_dvp == vp)
			vrele(nd.ni_dvp);
		else
			vput(nd.ni_dvp);
		if ((error = vn_start_write(NULL, &mp, V_XSLEEP | PCATCH)) != 0)
			return (error);
		goto restart;
	}
	vfs_notify_upper(vp, VFS_NOTIFY_UPPER_UNLINK);
	error = VOP_RMDIR(nd.ni_dvp, nd.ni_vp, &nd.ni_cnd);
	vn_finished_write(mp);
out:
	NDFREE(&nd, NDF_ONLY_PNBUF);
	vput(vp);
	if (nd.ni_dvp == vp)
		vrele(nd.ni_dvp);
	else
		vput(nd.ni_dvp);
	return (error);
}

#ifdef COMPAT_43
/*
 * Read a block of directory entries in a filesystem independent format.
 */
#ifndef _SYS_SYSPROTO_H_
struct ogetdirentries_args {
	int	fd;
	char	*buf;
	u_int	count;
	long	*basep;
};
#endif
int
ogetdirentries(struct thread *td, struct ogetdirentries_args *uap)
{
	long loff;
	int error;

	error = kern_ogetdirentries(td, uap, &loff);
	if (error == 0)
		error = copyout(&loff, uap->basep, sizeof(long));
	return (error);
}

int
kern_ogetdirentries(struct thread *td, struct ogetdirentries_args *uap,
    long *ploff)
{
	struct vnode *vp;
	struct file *fp;
	struct uio auio, kuio;
	struct iovec aiov, kiov;
	struct dirent *dp, *edp;
	cap_rights_t rights;
	caddr_t dirbuf;
	int error, eofflag, readcnt;
	long loff;
	off_t foffset;

	/* XXX arbitrary sanity limit on `count'. */
	if (uap->count > 64 * 1024)
		return (EINVAL);
	error = getvnode(td, uap->fd, cap_rights_init(&rights, CAP_READ), &fp);
	if (error != 0)
		return (error);
	if ((fp->f_flag & FREAD) == 0) {
		fdrop(fp, td);
		return (EBADF);
	}
	vp = fp->f_vnode;
	foffset = foffset_lock(fp, 0);
unionread:
	if (vp->v_type != VDIR) {
		foffset_unlock(fp, foffset, 0);
		fdrop(fp, td);
		return (EINVAL);
	}
	aiov.iov_base = uap->buf;
	aiov.iov_len = uap->count;
	auio.uio_iov = &aiov;
	auio.uio_iovcnt = 1;
	auio.uio_rw = UIO_READ;
	auio.uio_segflg = UIO_USERSPACE;
	auio.uio_td = td;
	auio.uio_resid = uap->count;
	vn_lock(vp, LK_SHARED | LK_RETRY);
	loff = auio.uio_offset = foffset;
#ifdef MAC
	error = mac_vnode_check_readdir(td->td_ucred, vp);
	if (error != 0) {
		VOP_UNLOCK(vp, 0);
		foffset_unlock(fp, foffset, FOF_NOUPDATE);
		fdrop(fp, td);
		return (error);
	}
#endif
#	if (BYTE_ORDER != LITTLE_ENDIAN)
		if (vp->v_mount->mnt_maxsymlinklen <= 0) {
			error = VOP_READDIR(vp, &auio, fp->f_cred, &eofflag,
			    NULL, NULL);
			foffset = auio.uio_offset;
		} else
#	endif
	{
		kuio = auio;
		kuio.uio_iov = &kiov;
		kuio.uio_segflg = UIO_SYSSPACE;
		kiov.iov_len = uap->count;
		dirbuf = malloc(uap->count, M_TEMP, M_WAITOK);
		kiov.iov_base = dirbuf;
		error = VOP_READDIR(vp, &kuio, fp->f_cred, &eofflag,
			    NULL, NULL);
		foffset = kuio.uio_offset;
		if (error == 0) {
			readcnt = uap->count - kuio.uio_resid;
			edp = (struct dirent *)&dirbuf[readcnt];
			for (dp = (struct dirent *)dirbuf; dp < edp; ) {
#				if (BYTE_ORDER == LITTLE_ENDIAN)
					/*
					 * The expected low byte of
					 * dp->d_namlen is our dp->d_type.
					 * The high MBZ byte of dp->d_namlen
					 * is our dp->d_namlen.
					 */
					dp->d_type = dp->d_namlen;
					dp->d_namlen = 0;
#				else
```

```
					/*
					 * The dp->d_type is the high byte
					 * of the expected dp->d_namlen,
					 * so must be zero'ed.
					 */
					dp->d_type = 0;
#				endif
				if (dp->d_reclen > 0) {
					dp = (struct dirent *)
					    ((char *)dp + dp->d_reclen);
				} else {
					error = EIO;
					break;
				}
			}
			if (dp >= edp)
				error = uiomove(dirbuf, readcnt, &auio);
		}
		free(dirbuf, M_TEMP);
	}
	if (error != 0) {
		VOP_UNLOCK(vp, 0);
		foffset_unlock(fp, foffset, 0);
		fdrop(fp, td);
		return (error);
	}
	if (uap->count == auio.uio_resid &&
	    (vp->v_vflag & VV_ROOT) &&
	    (vp->v_mount->mnt_flag & MNT_UNION)) {
		struct vnode *tvp = vp;
		vp = vp->v_mount->mnt_vnodecovered;
		VREF(vp);
		fp->f_vnode = vp;
		fp->f_data = vp;
		foffset = 0;
		vput(tvp);
		goto unionread;
	}
	VOP_UNLOCK(vp, 0);
	foffset_unlock(fp, foffset, 0);
	fdrop(fp, td);
	td->td_retval[0] = uap->count - auio.uio_resid;
	if (error == 0)
		*ploff = loff;
	return (error);
}
#endif /* COMPAT_43 */

/*
 * Read a block of directory entries in a filesystem independent format.
 */
#ifndef _SYS_SYSPROTO_H_
struct getdirentries_args {
	int	fd;
	char	*buf;
	u_int	count;
	long	*basep;
};
#endif
int
sys_getdirentries(struct thread *td, struct getdirentries_args *uap)
{
	long base;
	int error;

	error = kern_getdirentries(td, uap->fd, uap->buf, uap->count, &base,
	    NULL, UIO_USERSPACE);
	if (error != 0)
		return (error);
	if (uap->basep != NULL)
		error = copyout(&base, uap->basep, sizeof(long));
	return (error);
}

int
kern_getdirentries(struct thread *td, int fd, char *buf, u_int count,
    long *basep, ssize_t *residp, enum uio_seg bufseg)
{
	struct vnode *vp;
	struct file *fp;
	struct uio auio;
	struct iovec aiov;
	cap_rights_t rights;
	long loff;
	int error, eofflag;
	off_t foffset;

	AUDIT_ARG_FD(fd);
	if (count > IOSIZE_MAX)
		return (EINVAL);
	auio.uio_resid = count;
	error = getvnode(td, fd, cap_rights_init(&rights, CAP_READ), &fp);
	if (error != 0)
		return (error);
	if ((fp->f_flag & FREAD) == 0) {
		fdrop(fp, td);
		return (EBADF);
	}
```

```c
	vp = fp->f_vnode;
	foffset = foffset_lock(fp, 0);
unionread:
	if (vp->v_type != VDIR) {
		error = EINVAL;
		goto fail;
	}
	aiov.iov_base = buf;
	aiov.iov_len = count;
	auio.uio_iov = &aiov;
	auio.uio_iovcnt = 1;
	auio.uio_rw = UIO_READ;
	auio.uio_segflg = bufseg;
	auio.uio_td = td;
	vn_lock(vp, LK_SHARED | LK_RETRY);
	AUDIT_ARG_VNODE1(vp);
	loff = auio.uio_offset = foffset;
#ifdef MAC
	error = mac_vnode_check_readdir(td->td_ucred, vp);
	if (error == 0)
#endif
		error = VOP_READDIR(vp, &auio, fp->f_cred, &eofflag, NULL,
		    NULL);
	foffset = auio.uio_offset;
	if (error != 0) {
		VOP_UNLOCK(vp, 0);
		goto fail;
	}
	if (count == auio.uio_resid &&
	    (vp->v_vflag & VV_ROOT) &&
	    (vp->v_mount->mnt_flag & MNT_UNION)) {
		struct vnode *tvp = vp;

		vp = vp->v_mount->mnt_vnodecovered;
		VREF(vp);
		fp->f_vnode = vp;
		fp->f_data = vp;
		foffset = 0;
		vput(tvp);
		goto unionread;
	}
	VOP_UNLOCK(vp, 0);
	*basep = loff;
	if (residp != NULL)
		*residp = auio.uio_resid;
	td->td_retval[0] = count - auio.uio_resid;
fail:
	foffset_unlock(fp, foffset, 0);
	fdrop(fp, td);
	return (error);
}

#ifndef _SYS_SYSPROTO_H_
struct getdents_args {
	int fd;
	char *buf;
	size_t count;
};
#endif
int
sys_getdents(struct thread *td, struct getdents_args *uap)
{
	struct getdirentries_args ap;

	ap.fd = uap->fd;
	ap.buf = uap->buf;
	ap.count = uap->count;
	ap.basep = NULL;
	return (sys_getdirentries(td, &ap));
}

/*
 * Set the mode mask for creation of filesystem nodes.
 */
#ifndef _SYS_SYSPROTO_H_
struct umask_args {
	int	newmask;
};
#endif
int
sys_umask(struct thread *td, struct umask_args *uap)
{
	struct filedesc *fdp;

	fdp = td->td_proc->p_fd;
	FILEDESC_XLOCK(fdp);
	td->td_retval[0] = fdp->fd_cmask;
	fdp->fd_cmask = uap->newmask & ALLPERMS;
	FILEDESC_XUNLOCK(fdp);
	return (0);
}

/*
 * Void all references to file by ripping underlying filesystem away from
 * vnode.
 */
#ifndef _SYS_SYSPROTO_H_
struct revoke_args {
        char    *path;
};
#endif
int
sys_revoke(struct thread *td, struct revoke_args *uap)
{
        struct vnode *vp;
        struct vattr vattr;
        struct nameidata nd;
        int error;

        NDINIT(&nd, LOOKUP, FOLLOW | LOCKLEAF | AUDITVNODE1, UIO_USERSPACE,
            uap->path, td);
        if ((error = namei(&nd)) != 0)
                return (error);
        vp = nd.ni_vp;
        NDFREE(&nd, NDF_ONLY_PNBUF);
        if (vp->v_type != VCHR || vp->v_rdev == NULL) {
                error = EINVAL;
                goto out;
        }
#ifdef MAC
        error = mac_vnode_check_revoke(td->td_ucred, vp);
        if (error != 0)
                goto out;
#endif
        error = VOP_GETATTR(vp, &vattr, td->td_ucred);
        if (error != 0)
                goto out;
        if (td->td_ucred->cr_uid != vattr.va_uid) {
                error = priv_check(td, PRIV_VFS_ADMIN);
                if (error != 0)
                        goto out;
        }
        if (vcount(vp) > 1)
                VOP_REVOKE(vp, REVOKEALL);
out:
        vput(vp);
        return (error);
}

/*
 * Convert a user file descriptor to a kernel file entry and check that, if it
 * is a capability, the correct rights are present. A reference on the file
 * entry is held upon returning.
 */
int
getvnode(struct thread *td, int fd, cap_rights_t *rightsp, struct file **fpp)
{
        struct file *fp;
        int error;

        error = fget_unlocked(td->td_proc->p_fd, fd, rightsp, &fp, NULL);
        if (error != 0)
                return (error);

        /*
         * The file could be not of the vnode type, or it may be not
         * yet fully initialized, in which case the f_vnode pointer
         * may be set, but f_ops is still badfileops.  E.g.,
         * devfs_open() transiently create such situation to
         * facilitate csw d_fdopen().
         *
         * Dupfdopen() handling in kern_openat() installs the
         * half-baked file into the process descriptor table, allowing
         * other thread to dereference it. Guard against the race by
         * checking f_ops.
         */
        if (fp->f_vnode == NULL || fp->f_ops == &badfileops) {
                fdrop(fp, td);
                return (EINVAL);
        }
        *fpp = fp;
        return (0);
}


/*
 * Get an (NFS) file handle.
 */
#ifndef _SYS_SYSPROTO_H_
struct lgetfh_args {
        char    *fname;
        fhandle_t *fhp;
};
#endif
int
sys_lgetfh(struct thread *td, struct lgetfh_args *uap)
{
        struct nameidata nd;
        fhandle_t fh;
        struct vnode *vp;
        int error;

        error = priv_check(td, PRIV_VFS_GETFH);
        if (error != 0)
                return (error);
        NDINIT(&nd, LOOKUP, NOFOLLOW | LOCKLEAF | AUDITVNODE1, UIO_USERSPACE,
	    uap->fname, td);
	error = namei(&nd);
	if (error != 0)
		return (error);
	NDFREE(&nd, NDF_ONLY_PNBUF);
	vp = nd.ni_vp;
	bzero(&fh, sizeof(fh));
	fh.fh_fsid = vp->v_mount->mnt_stat.f_fsid;
	error = VOP_VPTOFH(vp, &fh.fh_fid);
	vput(vp);
	if (error == 0)
		error = copyout(&fh, uap->fhp, sizeof (fh));
	return (error);
}

#ifndef _SYS_SYSPROTO_H_
struct getfh_args {
	char	*fname;
	fhandle_t *fhp;
};
#endif
int
sys_getfh(struct thread *td, struct getfh_args *uap)
{
	struct nameidata nd;
	fhandle_t fh;
	struct vnode *vp;
	int error;

	error = priv_check(td, PRIV_VFS_GETFH);
	if (error != 0)
		return (error);
	NDINIT(&nd, LOOKUP, FOLLOW | LOCKLEAF | AUDITVNODE1, UIO_USERSPACE,
	    uap->fname, td);
	error = namei(&nd);
	if (error != 0)
		return (error);
	NDFREE(&nd, NDF_ONLY_PNBUF);
	vp = nd.ni_vp;
	bzero(&fh, sizeof(fh));
	fh.fh_fsid = vp->v_mount->mnt_stat.f_fsid;
	error = VOP_VPTOFH(vp, &fh.fh_fid);
	vput(vp);
	if (error == 0)
		error = copyout(&fh, uap->fhp, sizeof (fh));
	return (error);
}

/*
 * syscall for the rpc.lockd to use to translate a NFS file handle into an
 * open descriptor.
 *
 * warning: do not remove the priv_check() call or this becomes one giant
 * security hole.
 */
#ifndef _SYS_SYSPROTO_H_
struct fhopen_args {
	const struct fhandle *u_fhp;
	int flags;
};
#endif
int
sys_fhopen(struct thread *td, struct fhopen_args *uap)
{
	struct mount *mp;
	struct vnode *vp;
	struct fhandle fhp;
	struct file *fp;
	int fmode, error;
	int indx;

	error = priv_check(td, PRIV_VFS_FHOPEN);
	if (error != 0)
		return (error);
	indx = -1;
	fmode = FFLAGS(uap->flags);
	/* why not allow a non-read/write open for our lockd? */
	if (((fmode & (FREAD | FWRITE)) == 0) || (fmode & O_CREAT))
		return (EINVAL);
	error = copyin(uap->u_fhp, &fhp, sizeof(fhp));
	if (error != 0)
		return(error);
	/* find the mount point */
	mp = vfs_busyfs(&fhp.fh_fsid);
	if (mp == NULL)
		return (ESTALE);
	/* now give me my vnode, it gets returned to me locked */
	error = VFS_FHTOVP(mp, &fhp.fh_fid, LK_EXCLUSIVE, &vp);
	vfs_unbusy(mp);
	if (error != 0)
		return (error);

	error = falloc_noinstall(td, &fp);
	if (error != 0) {
		vput(vp);
		return (error);
	}
	/*
```

```
	 * An extra reference on `fp' has been held for us by
	 * falloc_noinstall().
	 */

#ifdef INVARIANTS
	td->td_dupfd = -1;
#endif
	error = vn_open_vnode(vp, fmode, td->td_ucred, td, fp);
	if (error != 0) {
		KASSERT(fp->f_ops == &badfileops,
		    ("VOP_OPEN in fhopen() set f_ops"));
		KASSERT(td->td_dupfd < 0,
		    ("fhopen() encountered fdopen()"));

		vput(vp);
		goto bad;
	}
#ifdef INVARIANTS
	td->td_dupfd = 0;
#endif
	fp->f_vnode = vp;
	fp->f_seqcount = 1;
	finit(fp, (fmode & FMASK) | (fp->f_flag & FHASLOCK), DTYPE_VNODE, vp,
	    &vnops);
	VOP_UNLOCK(vp, 0);
	if ((fmode & O_TRUNC) != 0) {
		error = fo_truncate(fp, 0, td->td_ucred, td);
		if (error != 0)
			goto bad;
	}

	error = finstall(td, fp, &indx, fmode, NULL);
bad:
	fdrop(fp, td);
	td->td_retval[0] = indx;
	return (error);
}

/*
 * Stat an (NFS) file handle.
 */
#ifndef _SYS_SYSPROTO_H_
struct fhstat_args {
	struct fhandle *u_fhp;
	struct stat *sb;
};
#endif
int
sys_fhstat(struct thread *td, struct fhstat_args *uap)
{
	struct stat sb;
	struct fhandle fh;
	int error;

	error = copyin(uap->u_fhp, &fh, sizeof(fh));
	if (error != 0)
		return (error);
	error = kern_fhstat(td, fh, &sb);
	if (error == 0)
		error = copyout(&sb, uap->sb, sizeof(sb));
	return (error);
}

int
kern_fhstat(struct thread *td, struct fhandle fh, struct stat *sb)
{
	struct mount *mp;
	struct vnode *vp;
	int error;

	error = priv_check(td, PRIV_VFS_FHSTAT);
	if (error != 0)
		return (error);
	if ((mp = vfs_busyfs(&fh.fh_fsid)) == NULL)
		return (ESTALE);
	error = VFS_FHTOVP(mp, &fh.fh_fid, LK_EXCLUSIVE, &vp);
	vfs_unbusy(mp);
	if (error != 0)
		return (error);
	error = vn_stat(vp, sb, td->td_ucred, NOCRED, td);
	vput(vp);
	return (error);
}

/*
 * Implement fstatfs() for (NFS) file handles.
 */
#ifndef _SYS_SYSPROTO_H_
struct fhstatfs_args {
	struct fhandle *u_fhp;
	struct statfs *buf;
};
#endif
int
sys_fhstatfs(struct thread *td, struct fhstatfs_args *uap)
{
	struct statfs *sfp;
	fhandle_t fh;
```

```c
	int error;

	error = copyin(uap->u_fhp, &fh, sizeof(fhandle_t));
	if (error != 0)
		return (error);
	sfp = malloc(sizeof(struct statfs), M_STATFS, M_WAITOK);
	error = kern_fhstatfs(td, fh, sfp);
	if (error == 0)
		error = copyout(sfp, uap->buf, sizeof(*sfp));
	free(sfp, M_STATFS);
	return (error);
}

int
kern_fhstatfs(struct thread *td, fhandle_t fh, struct statfs *buf)
{
	struct statfs *sp;
	struct mount *mp;
	struct vnode *vp;
	int error;

	error = priv_check(td, PRIV_VFS_FHSTATFS);
	if (error != 0)
		return (error);
	if ((mp = vfs_busyfs(&fh.fh_fsid)) == NULL)
		return (ESTALE);
	error = VFS_FHTOVP(mp, &fh.fh_fid, LK_EXCLUSIVE, &vp);
	if (error != 0) {
		vfs_unbusy(mp);
		return (error);
	}
	vput(vp);
	error = prison_canseemount(td->td_ucred, mp);
	if (error != 0)
		goto out;
#ifdef MAC
	error = mac_mount_check_stat(td->td_ucred, mp);
	if (error != 0)
		goto out;
#endif
	/*
	 * Set these in case the underlying filesystem fails to do so.
	 */
	sp = &mp->mnt_stat;
	sp->f_version = STATFS_VERSION;
	sp->f_namemax = NAME_MAX;
	sp->f_flags = mp->mnt_flag & MNT_VISFLAGMASK;
	error = VFS_STATFS(mp, sp);
	if (error == 0)
		*buf = *sp;
out:
	vfs_unbusy(mp);
	return (error);
}

int
kern_posix_fallocate(struct thread *td, int fd, off_t offset, off_t len)
{
	struct file *fp;
	struct mount *mp;
	struct vnode *vp;
	cap_rights_t rights;
	off_t olen, ooffset;
	int error;

	if (offset < 0 || len <= 0)
		return (EINVAL);
	/* Check for wrap. */
	if (offset > OFF_MAX - len)
		return (EFBIG);
	error = fget(td, fd, cap_rights_init(&rights, CAP_PWRITE), &fp);
	if (error != 0)
		return (error);
	if ((fp->f_ops->fo_flags & DFLAG_SEEKABLE) == 0) {
		error = ESPIPE;
		goto out;
	}
	if ((fp->f_flag & FWRITE) == 0) {
		error = EBADF;
		goto out;
	}
	if (fp->f_type != DTYPE_VNODE) {
		error = ENODEV;
		goto out;
	}
	vp = fp->f_vnode;
	if (vp->v_type != VREG) {
		error = ENODEV;
		goto out;
	}

	/* Allocating blocks may take a long time, so iterate. */
	for (;;) {
		olen = len;
		ooffset = offset;

		bwillwrite();
		mp = NULL;
		error = vn_start_write(vp, &mp, V_WAIT | PCATCH);
		if (error != 0)
			break;
		error = vn_lock(vp, LK_EXCLUSIVE);
		if (error != 0) {
			vn_finished_write(mp);
			break;
		}
#ifdef MAC
		error = mac_vnode_check_write(td->td_ucred, fp->f_cred, vp);
		if (error == 0)
#endif
			error = VOP_ALLOCATE(vp, &offset, &len);
		VOP_UNLOCK(vp, 0);
		vn_finished_write(mp);

		if (olen + ooffset != offset + len) {
			panic("offset + len changed from %jx/%jx to %jx/%jx",
			    ooffset, olen, offset, len);
		}
		if (error != 0 || len == 0)
			break;
		KASSERT(olen > len, ("Iteration did not make progress?"));
		maybe_yield();
	}
 out:
	fdrop(fp, td);
	return (error);
}

int
sys_posix_fallocate(struct thread *td, struct posix_fallocate_args *uap)
{
	int error;

	error = kern_posix_fallocate(td, uap->fd, uap->offset, uap->len);
	return (kern_posix_error(td, error));
}

/*
 * Unlike madvise(2), we do not make a best effort to remember every
 * possible caching hint.  Instead, we remember the last setting with
 * the exception that we will allow POSIX_FADV_NORMAL to adjust the
 * region of any current setting.
 */
int
kern_posix_fadvise(struct thread *td, int fd, off_t offset, off_t len,
    int advice)
{
	struct fadvise_info *fa, *new;
	struct file *fp;
	struct vnode *vp;
	cap_rights_t rights;
	off_t end;
	int error;

	if (offset < 0 || len < 0 || offset > OFF_MAX - len)
		return (EINVAL);
	switch (advice) {
	case POSIX_FADV_SEQUENTIAL:
	case POSIX_FADV_RANDOM:
	case POSIX_FADV_NOREUSE:
		new = malloc(sizeof(*fa), M_FADVISE, M_WAITOK);
		break;
	case POSIX_FADV_NORMAL:
	case POSIX_FADV_WILLNEED:
	case POSIX_FADV_DONTNEED:
		new = NULL;
		break;
	default:
		return (EINVAL);
	}
	/* XXX: CAP_POSIX_FADVISE? */
	error = fget(td, fd, cap_rights_init(&rights), &fp);
	if (error != 0)
		goto out;
	if ((fp->f_ops->fo_flags & DFLAG_SEEKABLE) == 0) {
		error = ESPIPE;
		goto out;
	}
	if (fp->f_type != DTYPE_VNODE) {
		error = ENODEV;
		goto out;
	}
	vp = fp->f_vnode;
	if (vp->v_type != VREG) {
		error = ENODEV;
		goto out;
	}
	if (len == 0)
		end = OFF_MAX;
	else
		end = offset + len - 1;
	switch (advice) {
	case POSIX_FADV_SEQUENTIAL:
	case POSIX_FADV_RANDOM:
	case POSIX_FADV_NOREUSE:
		/*
```

```
		 * Try to merge any existing non-standard region with
		 * this new region if possible, otherwise create a new
		 * non-standard region for this request.
		 */
		mtx_pool_lock(mtxpool_sleep, fp);
		fa = fp->f_advice;
		if (fa != NULL && fa->fa_advice == advice &&
		    ((fa->fa_start <= end && fa->fa_end >= offset) ||
		    (end != OFF_MAX && fa->fa_start == end + 1) ||
		    (fa->fa_end != OFF_MAX && fa->fa_end + 1 == offset))) {
			if (offset < fa->fa_start)
				fa->fa_start = offset;
			if (end > fa->fa_end)
				fa->fa_end = end;
		} else {
			new->fa_advice = advice;
			new->fa_start = offset;
			new->fa_end = end;
			fp->f_advice = new;
			new = fa;
		}
		mtx_pool_unlock(mtxpool_sleep, fp);
		break;
	case POSIX_FADV_NORMAL:
		/*
		 * If a the "normal" region overlaps with an existing
		 * non-standard region, trim or remove the
		 * non-standard region.
		 */
		mtx_pool_lock(mtxpool_sleep, fp);
		fa = fp->f_advice;
		if (fa != NULL) {
			if (offset <= fa->fa_start && end >= fa->fa_end) {
				new = fa;
				fp->f_advice = NULL;
			} else if (offset <= fa->fa_start &&
			    end >= fa->fa_start)
				fa->fa_start = end + 1;
			else if (offset <= fa->fa_end && end >= fa->fa_end)
				fa->fa_end = offset - 1;
			else if (offset >= fa->fa_start && end <= fa->fa_end) {
				/*
				 * If the "normal" region is a middle
				 * portion of the existing
				 * non-standard region, just remove
				 * the whole thing rather than picking
				 * one side or the other to
				 * preserve.
				 */
				new = fa;
				fp->f_advice = NULL;
			}
		}
		mtx_pool_unlock(mtxpool_sleep, fp);
		break;
	case POSIX_FADV_WILLNEED:
	case POSIX_FADV_DONTNEED:
		error = VOP_ADVISE(vp, offset, end, advice);
		break;
	}
out:
	if (fp != NULL)
		fdrop(fp, td);
	free(new, M_FADVISE);
	return (error);
}

int
sys_posix_fadvise(struct thread *td, struct posix_fadvise_args *uap)
{
	int error;

	error = kern_posix_fadvise(td, uap->fd, uap->offset, uap->len,
	    uap->advice);
	return (kern_posix_error(td, error));
}
```