```
**********************************************************
   87152 Sat May 26 07:33:22 2012
new/usr/src/uts/common/fs/smbclnt/smbfs/smbfs_vnops.c
*** NO COMMENTS ***
**********************************************************
______unchanged_portion_omitted_

  97 /*
  98  * Turning this on causes nodes to be created in the cache
  99  * during directory listings, normally avoiding a second
 100  * OtW attribute fetch just after a readdir.
 101  */
 102 int smbfs_fastlookup = 1;

 104 /* local static function defines */

 106 static int      smbfslookup_cache(vnode_t *, char *, int, vnode_t **,
 107                         cred_t *);
 108 static int      smbfslookup(vnode_t *dvp, char *nm, vnode_t **vpp, cred_t *cr,
 109                         int cache_ok, caller_context_t *);
 110 static int      smbfsrename(vnode_t *odvp, char *onm, vnode_t *ndvp, char *nnm,
 111                         cred_t *cr, caller_context_t *);
 112 static int      smbfssetattr(vnode_t *, struct vattr *, int, cred_t *);
 113 static int      smbfs_accessx(void *, int, cred_t *);
 114 static int      smbfs_readvdir(vnode_t *vp, uio_t *uio, cred_t *cr, int *eofp,
 115                         caller_context_t *);
 116 static void     smbfs_rele_fid(smbnode_t *, struct smb_cred *);

 118 /*
 119  * These are the vnode ops routines which implement the vnode interface to
 120  * the networked file system.  These routines just take their parameters,
 121  * make them look networkish by putting the right info into interface structs,
 122  * and then calling the appropriate remote routine(s) to do the work.
 123  *
 124  * Note on directory name lookup cacheing:  If we detect a stale fhandle,
 125  * we purge the directory cache relative to that vnode.  This way, the
 126  * user won't get burned by the cache repeatedly.  See <smbfs/smbnode.h> for
 127  * more details on smbnode locking.
 128  */

 130 static int      smbfs_open(vnode_t **, int, cred_t *, caller_context_t *);
 131 static int      smbfs_close(vnode_t *, int, int, offset_t, cred_t *,
 132                         caller_context_t *);
 133 static int      smbfs_read(vnode_t *, struct uio *, int, cred_t *,
 134                         caller_context_t *);
 135 static int      smbfs_write(vnode_t *, struct uio *, int, cred_t *,
 136                         caller_context_t *);
 137 static int      smbfs_ioctl(vnode_t *, int, intptr_t, int, cred_t *, int *,
 138                         caller_context_t *);
 139 static int      smbfs_getattr(vnode_t *, struct vattr *, int, cred_t *,
 140                         caller_context_t *);
 141 static int      smbfs_setattr(vnode_t *, struct vattr *, int, cred_t *,
 142                         caller_context_t *);
 143 static int      smbfs_access(vnode_t *, int, int, cred_t *, caller_context_t *);
 144 static int      smbfs_fsync(vnode_t *, int, cred_t *, caller_context_t *);
 145 static void     smbfs_inactive(vnode_t *, cred_t *, caller_context_t *);
 146 static int      smbfs_lookup(vnode_t *, char *, vnode_t **, struct pathname *,
 147                         int, vnode_t *, cred_t *, caller_context_t *,
 148                         int *, pathname_t *);
 149 static int      smbfs_create(vnode_t *, char *, struct vattr *, enum vcexcl,
 150                         int, vnode_t **, cred_t *, int, caller_context_t *,
 151                         vsecattr_t *);
 152 static int      smbfs_remove(vnode_t *, char *, cred_t *, caller_context_t *,
 153                         int);
 154 static int      smbfs_rename(vnode_t *, char *, vnode_t *, char *, cred_t *,
 155                         caller_context_t *, int);
```



```
 156 static int      smbfs_mkdir(vnode_t *, char *, struct vattr *, vnode_t **,
 157                         cred_t *, caller_context_t *, int, vsecattr_t *);
 158 static int      smbfs_rmdir(vnode_t *, char *, vnode_t *, cred_t *,
 159                         caller_context_t *, int);
 160 static int      smbfs_readdir(vnode_t *, struct uio *, cred_t *, int *,
 161                         caller_context_t *, int);
 162 static int      smbfs_rwlock(vnode_t *, int, caller_context_t *);
 163 static void     smbfs_rwunlock(vnode_t *, int, caller_context_t *);
 164 static int      smbfs_seek(vnode_t *, offset_t, offset_t *, caller_context_t *);
 165 static int      smbfs_frlock(vnode_t *, int, struct flock64 *, int, offset_t,
 166                         struct flk_callback *, cred_t *, caller_context_t *);
 167 static int      smbfs_space(vnode_t *, int, struct flock64 *, int, offset_t,
 168                         cred_t *, caller_context_t *);
 169 static int      smbfs_pathconf(vnode_t *, int, ulong_t *, cred_t *,
 170                         caller_context_t *);
 171 static int      smbfs_setsecattr(vnode_t *, vsecattr_t *, int, cred_t *,
 172                         caller_context_t *);
 173 static int      smbfs_getsecattr(vnode_t *, vsecattr_t *, int, cred_t *,
 174                         caller_context_t *);
 175 static int      smbfs_shrlock(vnode_t *, int, struct shrlock *, int, cred_t *,
 176                         caller_context_t *);

 178 static int smbfs_map(vnode_t *vp, offset_t off, struct as *as, caddr_t *addrp,
 179         size_t len, uchar_t prot, uchar_t maxprot, uint_t flags, cred_t *cr,
 180         caller_context_t *ct);

 182 static int smbfs_addmap(vnode_t *vp, offset_t off, struct as *as, caddr_t addr,
 183         size_t len, uchar_t prot, uchar_t maxprot, uint_t flags, cred_t *cr,
 184         caller_context_t *ct);

 186 static int smbfs_delmap(vnode_t *vp, offset_t off, struct as *as, caddr_t addr,
 187         size_t len, uint_t prot, uint_t maxprot, uint_t flags, cred_t *cr,
 188         caller_context_t *ct);

 190 static int smbfs_putpage(vnode_t *vp, offset_t off, size_t len, int flags,
 191         cred_t *cr, caller_context_t *ct);

 193 static int smbfs_putapage(vnode_t *vp, page_t *pp, u_offset_t *offp, size_t *len
 194         int flags, cred_t *cr);

 196 static int up_mapin(uio_t *uiop, page_t *pp);

 198 static int up_mapout(uio_t *uiop, page_t *pp);

 200 static int smbfs_getpage(vnode_t *vp, offset_t off, size_t len, uint_t *protp, p
 201         enum seg_rw rw, cred_t *cr, caller_context_t *ct);

 203 static int smbfs_getapage(vnode_t *vp, u_offset_t off, size_t len,
 204         uint_t *protp, page_t *pl[], size_t plsz, struct seg *seg, caddr_t addr,



 208 #endif /* ! codereview */
 209 /* Dummy function to use until correct function is ported in */
 210 int noop_vnodeop() {
 211         return (0);
 212 }

 214 struct vnodeops *smbfs_vnodeops = NULL;

 216 /*
 217  * Most unimplemented ops will return ENOSYS because of fs_nosys().
 218  * The only ops where that won't work are ACCESS (due to open(2)
 219  * failures) and ... (anything else left?)
 220  */
 221 const fs_operation_def_t smbfs_vnodeops_template[] = {
```

```
222         { VOPNAME_OPEN,         { .vop_open = smbfs_open } },
223         { VOPNAME_CLOSE,        { .vop_close = smbfs_close } },
224         { VOPNAME_READ,         { .vop_read = smbfs_read } },
225         { VOPNAME_WRITE,        { .vop_write = smbfs_write } },
226         { VOPNAME_IOCTL,        { .vop_ioctl = smbfs_ioctl } },
227         { VOPNAME_GETATTR,      { .vop_getattr = smbfs_getattr } },
228         { VOPNAME_SETATTR,      { .vop_setattr = smbfs_setattr } },
229         { VOPNAME_ACCESS,       { .vop_access = smbfs_access } },
230         { VOPNAME_LOOKUP,       { .vop_lookup = smbfs_lookup } },
231         { VOPNAME_CREATE,       { .vop_create = smbfs_create } },
232         { VOPNAME_REMOVE,       { .vop_remove = smbfs_remove } },
233         { VOPNAME_LINK,         { .error = fs_nosys } }, /* smbfs_link, */
234         { VOPNAME_RENAME,       { .vop_rename = smbfs_rename } },
235         { VOPNAME_MKDIR,        { .vop_mkdir = smbfs_mkdir } },
236         { VOPNAME_RMDIR,        { .vop_rmdir = smbfs_rmdir } },
237         { VOPNAME_READDIR,      { .vop_readdir = smbfs_readdir } },
238         { VOPNAME_SYMLINK,      { .error = fs_nosys } }, /* smbfs_symlink, */
239         { VOPNAME_READLINK,     { .error = fs_nosys } }, /* smbfs_readlink, */
240         { VOPNAME_FSYNC,        { .vop_fsync = smbfs_fsync } },
241         { VOPNAME_INACTIVE,     { .vop_inactive = smbfs_inactive } },
242         { VOPNAME_FID,          { .error = fs_nosys } }, /* smbfs_fid, */
243         { VOPNAME_RWLOCK,       { .vop_rwlock = smbfs_rwlock } },
244         { VOPNAME_RWUNLOCK,     { .vop_rwunlock = smbfs_rwunlock } },
245         { VOPNAME_SEEK,         { .vop_seek = smbfs_seek } },
246         { VOPNAME_FRLOCK,       { .vop_frlock = smbfs_frlock } },
247         { VOPNAME_SPACE,        { .vop_space = smbfs_space } },
248         { VOPNAME_REALVP,       { .error = fs_nosys } }, /* smbfs_realvp, */
249         { VOPNAME_GETPAGE,      { .vop_getpage = smbfs_getpage } }, /* smbfs_get
250         { VOPNAME_PUTPAGE,      { .vop_putpage = smbfs_putpage } }, /* smbfs_put
251         { VOPNAME_MAP,          { .vop_map = smbfs_map } }, /* smbfs_map, */
252         { VOPNAME_ADDMAP,       { .vop_addmap = smbfs_addmap } }, /* smbfs_addma
253         { VOPNAME_DELMAP,       { .vop_delmap = smbfs_delmap } }, /* smbfs_delma
178         { VOPNAME_GETPAGE,      { .error = fs_nosys } }, /* smbfs_getpage, */
179         { VOPNAME_PUTPAGE,      { .error = fs_nosys } }, /* smbfs_putpage, */
180         { VOPNAME_MAP,          { .error = fs_nosys } }, /* smbfs_map, */
181         { VOPNAME_ADDMAP,       { .error = fs_nosys } }, /* smbfs_addmap, */
182         { VOPNAME_DELMAP,       { .error = fs_nosys } }, /* smbfs_delmap, */
254         { VOPNAME_DUMP,         { .error = fs_nosys } }, /* smbfs_dump, */
255         { VOPNAME_PATHCONF,     { .vop_pathconf = smbfs_pathconf } },
256         { VOPNAME_PAGEIO,       { .error = fs_nosys } }, /* smbfs_pageio, */
257         { VOPNAME_SETSECATTR,   { .vop_setsecattr = smbfs_setsecattr } },
258         { VOPNAME_GETSECATTR,   { .vop_getsecattr = smbfs_getsecattr } },
259         { VOPNAME_SHRLOCK,      { .vop_shrlock = smbfs_shrlock } },
260         { NULL, NULL }
261 };
_____unchanged_portion_omitted_


3148 static int smbfs_map(vnode_t *vp, offset_t off, struct as *as, caddr_t *addrp,
3149         size_t len, uchar_t prot, uchar_t maxprot, uint_t flags, cred_t *cr,
3150         caller_context_t *ct) {
3151     smbnode_t *np;
3152     smbmntinfo_t *smi;
3153     struct vattr va;
3154     segvn_crargs_t vn_a;
3155     int error;

3157     np = VTOSMB(vp);
3158     smi = VTOSMI(vp);

3160     if (curproc->p_zone != smi->smi_zone_ref.zref_zone)
3161         return (EIO);

3163     if (smi->smi_flags & SMI_DEAD || vp->v_vfsp->vfs_flag & VFS_UNMOUNTED)
```



```
3164         return (EIO);

3166     if (vp->v_flag & VNOMAP || vp->v_flag & VNOCACHE)
3167         return (EAGAIN);

3169     if (vp->v_type != VREG)
3170         return (ENODEV);

3172     va.va_mask = AT_ALL;

3174     if (error = smbfsgetattr(vp, &va, cr))
3175         return (error);

3177     if (smbfs_rw_enter_sig(&np->r_lkserlock, RW_WRITER, SMBINTR(vp)))
3178         return (EINTR);

3180     if (MANDLOCK(vp, va.va_mode)) {
3181         error = EAGAIN;
3182         goto out;
3183     }

3185     as_rangelock(as);
3186     error = choose_addr(as, addrp, len, off, ADDR_VACALIGN, flags);

3188     if (error != 0) {
3189         as_rangeunlock(as);
3190         goto out;
3191     }

3193     vn_a.vp = vp;
3194     vn_a.offset = (u_offset_t) off;
3195     vn_a.type = flags & MAP_TYPE;
3196     vn_a.prot = (uchar_t) prot;
3197     vn_a.maxprot = (uchar_t) maxprot;
3198     vn_a.cred = cr;
3199     vn_a.amp = NULL;
3200     vn_a.flags = flags & ~MAP_TYPE;
3201     vn_a.szc = 0;
3202     vn_a.lgrp_mem_policy_flags = 0;

3204     error = as_map(as, *addrp, len, segvn_create, &vn_a);

3206     as_rangeunlock(as);

3208 out:
3209     smbfs_rw_exit(&np->r_lkserlock);

3211     return (error);
3212 }

3214 static int smbfs_addmap(vnode_t *vp, offset_t off, struct as *as, caddr_t addr,
3215         size_t len, uchar_t prot, uchar_t maxprot, uint_t flags, cred_t *cr,
3216         caller_context_t *ct) {
3217     atomic_add_long((ulong_t *) & VTOSMB(vp)->r_mapcnt, btopr(len));
3218     return (0);
3219 }

3221 static int smbfs_delmap(vnode_t *vp, offset_t off, struct as *as, caddr_t addr,
3222         size_t len, uint_t prot, uint_t maxprot, uint_t flags, cred_t *cr,
3223         caller_context_t *ct) {
3224     atomic_add_long((ulong_t *) & VTOSMB(vp)->r_mapcnt, -btopr(len));
3225     return (0);
3226 }

3228 static int smbfs_putpage(vnode_t *vp, offset_t off, size_t len, int flags,
3229         cred_t *cr, caller_context_t *ct) {
```

```
3231     smbnode_t *np;
3232     size_t io_len;
3233     u_offset_t io_off;
3234     u_offset_t eoff;
3235     int error = 0;
3236     page_t *pp;

3238     np = VTOSMB(vp);

3240     if (len == 0) {
3241         error = pvn_vplist_dirty(vp, off, smbfs_putapage, flags, cr);
3242     } else {

3244         eoff = off + len;

3246         mutex_enter(&np->r_statelock);
3247         if (eoff > np->r_size)
3248             eoff = np->r_size;
3249         mutex_exit(&np->r_statelock);

3251         for (io_off = off; io_off < eoff; io_off += io_len) {
3252             if ((flags & B_INVAL) || (flags & B_ASYNC) == 0) {
3253                 pp = page_lookup(vp, io_off,
3254                         (flags & (B_INVAL | B_FREE) ? SE_EXCL : SE_SHARED));
3255             } else {
3256                 pp = page_lookup_nowait(vp, io_off,
3257                         (flags & B_FREE) ? SE_EXCL : SE_SHARED);
3258             }

3260             if (pp == NULL || !pvn_getdirty(pp, flags))
3261                 io_len = PAGESIZE;
3262             else {
3263                 error = smbfs_putapage(vp, pp, &io_off, &io_len, flags, cr);
3264             }
3265         }

3267     }

3269     return (error);
3270 }

3272 static int smbfs_putapage(vnode_t *vp, page_t *pp, u_offset_t *offp, size_t *len
3273         int flags, cred_t *cr) {

3275     struct smb_cred scred;
3276     smbnode_t *np;
3277     smbmntinfo_t *smi;
3278     smb_share_t *ssp;
3279     uio_t uio;
3280     iovec_t uiov;

3282     u_offset_t off;
3283     size_t len;
3284     int error, timo;

3286     np = VTOSMB(vp);
3287     smi = VTOSMI(vp);
3288     ssp = smi->smi_share;

3290     off = pp->p_offset;
3291     len = PAGESIZE;

3293     if (off >= np->r_size) {
3294         error = 0;
3295         goto out;
```



```
3296     } else if (off + len > np->r_size) {
3297         int npages = btopr(np->r_size - off);
3298         page_t *trunc;

3300         page_list_break(&pp, &trunc, npages);
3301         if (trunc)
3302             pvn_write_done(trunc, flags);
3303         len = np->r_size - off;
3304     }

3306     timo = smb_timo_write;

3308     if (smbfs_rw_enter_sig(&np->r_lkserlock, RW_READER, SMBINTR(vp)))
3309         return (EINTR);
3310     smb_credinit(&scred, cr);

3312     if (np->n_vcgenid != ssp->ss_vcgenid)
3313         error = ESTALE;
3314     else {
3315         uiov.iov_base = 0;
3316         uiov.iov_len = 0;
3317         uio.uio_iov = &uiov;
3318         uio.uio_iovcnt = 1;
3319         uio.uio_loffset = off;
3320         uio.uio_resid = len;
3321         uio.uio_segflg = UIO_SYSSPACE;
3322         uio.uio_llimit = MAXOFFSET_T;
3323         error = up_mapin(&uio, pp);
3324         if (error == 0) {
3325             error = smb_rwuio(ssp, np->n_fid, UIO_WRITE, &uio, &scred, timo);
3326             if (error == 0) {
3327                 mutex_enter(&np->r_statelock);
3328                 np->n_flag |= (NFLUSHWIRE | NATTRCHANGED);
3329                 mutex_exit(&np->r_statelock);
3330                 (void) smbfs_smb_flush(np, &scred);
3331             }
3332             up_mapout(&uio, pp);
3333         }
3334     }
3335     smb_credrele(&scred);
3336     smbfs_rw_exit(&np->r_lkserlock);

3338 out:
3339     pvn_write_done(pp, B_WRITE | flags);

3341     return (error);
3342 }

3344 static int up_mapin(uio_t *uiop, page_t *pp) {
3345     u_offset_t off;
3346     size_t size;
3347     pgcnt_t npages;
3348     caddr_t kaddr;

3350     off = (uintptr_t) uiop->uio_loffset & PAGEOFFSET;
3351     size = P2ROUNDUP(uiop->uio_resid + off, PAGESIZE);
3352     npages = btop(size);

3354     if (npages == 1 && kpm_enable) {
3355         kaddr = hat_kpm_mapin(pp, NULL);
3356         uiop->uio_iov->iov_base = kaddr;
3357         uiop->uio_iov->iov_len = PAGESIZE;
3358         return (0);
3359     }
3360     return (EFAULT);
3361 }
```

```
3363 static int up_mapout(uio_t *uiop, page_t *pp) {
3364     u_offset_t off;
3365     size_t size;
3366     pgcnt_t npages;
3367     caddr_t kaddr;

3369     kaddr = uiop->uio_iov->iov_base;
3370     off = (uintptr_t) kaddr & PAGEOFFSET;
3371     size = P2ROUNDUP(uiop->uio_iov->iov_len + off, PAGESIZE);
3372     npages = btop(size);

3374     if (npages == 1 && kpm_enable) {
3375         kaddr = (caddr_t) ((uintptr_t) kaddr & MMU_PAGEMASK);
3376         hat_kpm_mapout(pp, NULL, kaddr);
3377         uiop->uio_iov->iov_base = 0;
3378         uiop->uio_iov->iov_len = 0;
3379         return (0);
3380     }
3381     return (EFAULT);
3382 }

3384 static int smbfs_getpage(vnode_t *vp, offset_t off, size_t len, uint_t *protp,
3385         page_t *pl[], size_t plsz, struct seg *seg, caddr_t addr,
3386         enum seg_rw rw, cred_t *cr, caller_context_t *ct) {
3387     int error;
3388     smbnode_t *np;

3390     np = VTOSMB(vp);

3392     mutex_enter(&np->r_statelock);
3393     if (off + len > np->r_size + PAGEOFFSET && seg != segkmap) {
3394         mutex_exit(&np->r_statelock);
3395         return (EFAULT);
3396     }
3397     mutex_exit(&np->r_statelock);

3399     if (len <= PAGESIZE) {
3400         error = smbfs_getapage(vp, off, len, protp, pl, plsz, seg, addr, rw,
3401                 cr);
3402     } else {
3403         error = pvn_getpages(smbfs_getapage, vp, off, len, protp, pl, plsz, seg,
3404                 addr, rw, cr);
3405     }
3406     return (error);
3407 }

3409 static int smbfs_getapage(vnode_t *vp, u_offset_t off, size_t len,
3410         uint_t *protp, page_t *pl[], size_t plsz, struct seg *seg, caddr_t addr,
3411         enum seg_rw rw, cred_t *cr) {

3413     smbnode_t *np;
3414     smbmntinfo_t *smi;
3415     smb_share_t *ssp;
3416     smb_cred_t scred;

3418     page_t *pagefound, *pp;
3419     uio_t uio;
3420     iovec_t uiov;

3422     int error = 0, timo;

3424     np = VTOSMB(vp);
3425     smi = VTOSMI(vp);
3426     ssp = smi->smi_share;
```



```
3428     if (len > PAGESIZE)
3429         return (EFAULT);
3430     len = PAGESIZE;

3432     if (pl == NULL)
3433         return (EFAULT);

3435     if (smbfs_rw_enter_sig(&np->r_lkserlock, RW_READER, SMBINTR(vp)))
3436         return EINTR;

3438     smb_credinit(&scred, cr);

3440 again:
3441     if ((pagefound = page_exists(vp, off)) == NULL) {
3442         if ((pp = page_create_va(vp, off, PAGESIZE, PG_WAIT | PG_EXCL, seg, addr
3443             goto again;
3444         if (rw == S_CREATE) {
3445             goto out;
3446         } else {
3447             timo = smb_timo_read;

3449             uiov.iov_base = 0;
3450             uiov.iov_len = 0;
3451             uio.uio_iov = &uiov;
3452             uio.uio_iovcnt = 1;
3453             uio.uio_loffset = off;
3454             uio.uio_resid = len;
3455             uio.uio_segflg = UIO_SYSSPACE;
3456             uio.uio_llimit = MAXOFFSET_T;
3457             error = up_mapin(&uio, pp);
3458             if (error == 0) {
3459                 error = smb_rwuio(ssp, np->n_fid, UIO_READ, &uio, &scred, timo);
3460                 up_mapout(&uio, pp);
3461             }
3462         }
3463     } else {
3464         se_t se = rw == S_CREATE ? SE_EXCL : SE_SHARED;
3465         if ((pp = page_lookup(vp, off, se)) == NULL) {
3466             goto again;
3467         }
3468     }

3470 out:
3471     if (pp) {
3472         pvn_plist_init(pp, pl, plsz, off, PAGESIZE, rw);
3473     }

3475     smb_credrele(&scred);
3476     smbfs_rw_exit(&np->r_lkserlock);

3478     return (error);
3479 }
3480 #endif /* ! codereview */
```