```
**********************************************************
  116613 Thu Jan 31 10:01:31 2013
new/usr/src/uts/common/fs/nfs/nfs4_client.c
NFS4 data corruption (#3508)
If async calls are disabled, nfs4_async_putapage is supposed to do its
work synchronously. Due to a bug, it sometimes just does nothing, leaving
the page for later.
Unfortunately the caller has already reset the R4DIRTY flag.
Without R4DIRTY, nfs4_attrcache_va can't see that there are still
outstanding writes and accepts the file size from the server, which is
too low.
When the dirty page finally gets written back, the page size is truncated
to the file size, leaving some bytes unwritten.
Reviewed by: Marcel Telka <marcel@telka.sk>
Reviewed by: Robert Gordon <rbg@openrbg.com>
**********************************************************
______unchanged_portion_omitted_
```

```
1698 int
1699 nfs4_async_putapage(vnode_t *vp, page_t *pp, u_offset_t off, size_t len,
1700     int flags, cred_t *cr, int (*putapage)(vnode_t *, page_t *,
1701     u_offset_t, size_t, int, cred_t *))
1702 {
1703         rnode4_t *rp;
1704         mntinfo4_t *mi;
1705         struct nfs4_async_reqs *args;

1707         ASSERT(flags & B_ASYNC);
1708         ASSERT(vp->v_vfsp != NULL);

1710         rp = VTOR4(vp);
1711         ASSERT(rp->r_count > 0);

1713         mi = VTOMI4(vp);

1715         /*
1716          * If we can't allocate a request structure, do the putpage
1717          * operation synchronously in this thread's context.
1718          */
1719         if ((args = kmem_alloc(sizeof (*args), KM_NOSLEEP)) == NULL)
1720                 goto noasync;

1722         args->a_next = NULL;
1723 #ifdef DEBUG
1724         args->a_queuer = curthread;
1725 #endif
1726         VN_HOLD(vp);
1727         args->a_vp = vp;
1728         ASSERT(cr != NULL);
1729         crhold(cr);
1730         args->a_cred = cr;
1731         args->a_io = NFS4_PUTAPAGE;
1732         args->a_nfs4_putapage = putapage;
1733         args->a_nfs4_pp = pp;
1734         args->a_nfs4_off = off;
1735         args->a_nfs4_len = (uint_t)len;
1736         args->a_nfs4_flags = flags;

1738         mutex_enter(&mi->mi_async_lock);

1740         /*
1741          * If asyncio has been disabled, then make a synchronous request.
1742          * This check is done a second time in case async io was diabled
1743          * while this thread was blocked waiting for memory pressure to
1744          * reduce or for the queue to drain.
1745          */
```



```
1746         if (mi->mi_max_threads == 0) {
1747                 mutex_exit(&mi->mi_async_lock);

1749                 VN_RELE(vp);
1750                 crfree(cr);
1751                 kmem_free(args, sizeof (*args));
1752                 goto noasync;
1753         }

1755         /*
1756          * Link request structure into the async list and
1757          * wakeup async thread to do the i/o.
1758          */
1759         if (mi->mi_async_reqs[NFS4_PUTAPAGE] == NULL) {
1760                 mi->mi_async_reqs[NFS4_PUTAPAGE] = args;
1761                 mi->mi_async_tail[NFS4_PUTAPAGE] = args;
1762         } else {
1763                 mi->mi_async_tail[NFS4_PUTAPAGE]->a_next = args;
1764                 mi->mi_async_tail[NFS4_PUTAPAGE] = args;
1765         }

1767         mutex_enter(&rp->r_statelock);
1768         rp->r_count++;
1769         rp->r_awcount++;
1770         mutex_exit(&rp->r_statelock);

1772         if (mi->mi_io_kstats) {
1773                 mutex_enter(&mi->mi_lock);
1774                 kstat_waitq_enter(KSTAT_IO_PTR(mi->mi_io_kstats));
1775                 mutex_exit(&mi->mi_lock);
1776         }

1778         mi->mi_async_req_count++;
1779         ASSERT(mi->mi_async_req_count != 0);
1780         cv_signal(&mi->mi_async_reqs_cv);
1781         mutex_exit(&mi->mi_async_lock);
1782         return (0);

1784 noasync:

1786         if (curproc == proc_pageout || curproc == proc_fsflush) {
1786         if (curproc == proc_pageout || curproc == proc_fsflush ||
1787             nfs_zone() == mi->mi_zone) {
1787                 /*
1788                  * If we get here in the context of the pageout/fsflush,
1789                  * or we have run out of memory or we're attempting to
1790                  * unmount we refuse to do a sync write, because this may
1791                  * hang pageout/fsflush and the machine. In this case,
1792                  * we just re-mark the page as dirty and punt on the page.
1793                  *
1794                  * Make sure B_FORCE isn't set.  We can re-mark the
1795                  * pages as dirty and unlock the pages in one swoop by
1796                  * passing in B_ERROR to pvn_write_done().  However,
1797                  * we should make sure B_FORCE isn't set - we don't
1798                  * want the page tossed before it gets written out.
1799                  */
1800                 if (flags & B_FORCE)
1801                         flags &= ~(B_INVAL | B_FORCE);
1802                 pvn_write_done(pp, flags | B_ERROR);
1803                 return (0);
1804         }

1806         if (nfs_zone() != mi->mi_zone) {
1807 #endif /* ! codereview */
1808                 /*
1809                  * So this was a cross-zone sync putpage.
```

```
1807          * We'll get here only if (nfs_zone() != mi->mi_zone)
1808          * which means that this was a cross-zone sync putpage.
1810                 *
1811                 * We pass in B_ERROR to pvn_write_done() to re-mark the pages
1812                 * as dirty and unlock them.
1813                 *
1814                 * We don't want to clear B_FORCE here as the caller presumably
1815                 * knows what they're doing if they set it.
1816                 */
1817                pvn_write_done(pp, flags | B_ERROR);
1818                return (EPERM);
1819        }
1820        return ((*putapage)(vp, pp, off, len, flags, cr));
1821 #endif /* ! codereview */
1822 }

1824 int
1825 nfs4_async_pageio(vnode_t *vp, page_t *pp, u_offset_t io_off, size_t io_len,
1826     int flags, cred_t *cr, int (*pageio)(vnode_t *, page_t *, u_offset_t,
1827     size_t, int, cred_t *))
1828 {
1829        rnode4_t *rp;
1830        mntinfo4_t *mi;
1831        struct nfs4_async_reqs *args;

1833        ASSERT(flags & B_ASYNC);
1834        ASSERT(vp->v_vfsp != NULL);

1836        rp = VTOR4(vp);
1837        ASSERT(rp->r_count > 0);

1839        mi = VTOMI4(vp);

1841        /*
1842         * If we can't allocate a request structure, do the pageio
1843         * request synchronously in this thread's context.
1844         */
1845        if ((args = kmem_alloc(sizeof (*args), KM_NOSLEEP)) == NULL)
1846                goto noasync;

1848        args->a_next = NULL;
1849 #ifdef DEBUG
1850        args->a_queuer = curthread;
1851 #endif
1852        VN_HOLD(vp);
1853        args->a_vp = vp;
1854        ASSERT(cr != NULL);
1855        crhold(cr);
1856        args->a_cred = cr;
1857        args->a_io = NFS4_PAGEIO;
1858        args->a_nfs4_pageio = pageio;
1859        args->a_nfs4_pp = pp;
1860        args->a_nfs4_off = io_off;
1861        args->a_nfs4_len = (uint_t)io_len;
1862        args->a_nfs4_flags = flags;

1864        mutex_enter(&mi->mi_async_lock);

1866        /*
1867         * If asyncio has been disabled, then make a synchronous request.
1868         * This check is done a second time in case async io was diabled
1869         * while this thread was blocked waiting for memory pressure to
1870         * reduce or for the queue to drain.
1871         */
1872        if (mi->mi_max_threads == 0) {
1873                mutex_exit(&mi->mi_async_lock);
```



```
1875                VN_RELE(vp);
1876                crfree(cr);
1877                kmem_free(args, sizeof (*args));
1878                goto noasync;
1879        }

1881        /*
1882         * Link request structure into the async list and
1883         * wakeup async thread to do the i/o.
1884         */
1885        if (mi->mi_async_reqs[NFS4_PAGEIO] == NULL) {
1886                mi->mi_async_reqs[NFS4_PAGEIO] = args;
1887                mi->mi_async_tail[NFS4_PAGEIO] = args;
1888        } else {
1889                mi->mi_async_tail[NFS4_PAGEIO]->a_next = args;
1890                mi->mi_async_tail[NFS4_PAGEIO] = args;
1891        }

1893        mutex_enter(&rp->r_statelock);
1894        rp->r_count++;
1895        rp->r_awcount++;
1896        mutex_exit(&rp->r_statelock);

1898        if (mi->mi_io_kstats) {
1899                mutex_enter(&mi->mi_lock);
1900                kstat_waitq_enter(KSTAT_IO_PTR(mi->mi_io_kstats));
1901                mutex_exit(&mi->mi_lock);
1902        }

1904        mi->mi_async_req_count++;
1905        ASSERT(mi->mi_async_req_count != 0);
1906        cv_signal(&mi->mi_async_reqs_cv);
1907        mutex_exit(&mi->mi_async_lock);
1908        return (0);

1910 noasync:
1911        /*
1912         * If we can't do it ASYNC, for reads we do nothing (but cleanup
1913         * the page list), for writes we do it synchronously, except for
1914         * proc_pageout/proc_fsflush as described below.
1915         */
1916        if (flags & B_READ) {
1917                pvn_read_done(pp, flags | B_ERROR);
1918                return (0);
1919        }

1921        if (curproc == proc_pageout || curproc == proc_fsflush) {
1922                /*
1923                 * If we get here in the context of the pageout/fsflush,
1924                 * we refuse to do a sync write, because this may hang
1925                 * pageout/fsflush (and the machine). In this case, we just
1926                 * re-mark the page as dirty and punt on the page.
1927                 *
1928                 * Make sure B_FORCE isn't set.  We can re-mark the
1929                 * pages as dirty and unlock the pages in one swoop by
1930                 * passing in B_ERROR to pvn_write_done().  However,
1931                 * we should make sure B_FORCE isn't set - we don't
1932                 * want the page tossed before it gets written out.
1933                 */
1934                if (flags & B_FORCE)
1935                        flags &= ~(B_INVAL | B_FORCE);
1936                pvn_write_done(pp, flags | B_ERROR);
1937                return (0);
1938        }
```

```
1940         if (nfs_zone() != mi->mi_zone) {
1941                 /*
1942                  * So this was a cross-zone sync pageio.  We pass in B_ERROR
1943                  * to pvn_write_done() to re-mark the pages as dirty and unlock
1944                  * them.
1945                  *
1946                  * We don't want to clear B_FORCE here as the caller presumably
1947                  * knows what they're doing if they set it.
1948                  */
1949                 pvn_write_done(pp, flags | B_ERROR);
1950                 return (EPERM);
1951         }
1952         return ((*pageio)(vp, pp, io_off, io_len, flags, cr));
1953 }

1955 void
1956 nfs4_async_readdir(vnode_t *vp, rddir4_cache *rdc, cred_t *cr,
1957     int (*readdir)(vnode_t *, rddir4_cache *, cred_t *))
1958 {
1959         rnode4_t *rp;
1960         mntinfo4_t *mi;
1961         struct nfs4_async_reqs *args;

1963         rp = VTOR4(vp);
1964         ASSERT(rp->r_freef == NULL);

1966         mi = VTOMI4(vp);

1968         /*
1969          * If we can't allocate a request structure, skip the readdir.
1970          */
1971         if ((args = kmem_alloc(sizeof (*args), KM_NOSLEEP)) == NULL)
1972                 goto noasync;

1974         args->a_next = NULL;
1975 #ifdef DEBUG
1976         args->a_queuer = curthread;
1977 #endif
1978         VN_HOLD(vp);
1979         args->a_vp = vp;
1980         ASSERT(cr != NULL);
1981         crhold(cr);
1982         args->a_cred = cr;
1983         args->a_io = NFS4_READDIR;
1984         args->a_nfs4_readdir = readdir;
1985         args->a_nfs4_rdc = rdc;

1987         mutex_enter(&mi->mi_async_lock);

1989         /*
1990          * If asyncio has been disabled, then skip this request
1991          */
1992         if (mi->mi_max_threads == 0) {
1993                 mutex_exit(&mi->mi_async_lock);

1995                 VN_RELE(vp);
1996                 crfree(cr);
1997                 kmem_free(args, sizeof (*args));
1998                 goto noasync;
1999         }

2001         /*
2002          * Link request structure into the async list and
2003          * wakeup async thread to do the i/o.
2004          */
2005         if (mi->mi_async_reqs[NFS4_READDIR] == NULL) {
```



```
2006                 mi->mi_async_reqs[NFS4_READDIR] = args;
2007                 mi->mi_async_tail[NFS4_READDIR] = args;
2008         } else {
2009                 mi->mi_async_tail[NFS4_READDIR]->a_next = args;
2010                 mi->mi_async_tail[NFS4_READDIR] = args;
2011         }

2013         mutex_enter(&rp->r_statelock);
2014         rp->r_count++;
2015         mutex_exit(&rp->r_statelock);

2017         if (mi->mi_io_kstats) {
2018                 mutex_enter(&mi->mi_lock);
2019                 kstat_waitq_enter(KSTAT_IO_PTR(mi->mi_io_kstats));
2020                 mutex_exit(&mi->mi_lock);
2021         }

2023         mi->mi_async_req_count++;
2024         ASSERT(mi->mi_async_req_count != 0);
2025         cv_signal(&mi->mi_async_reqs_cv);
2026         mutex_exit(&mi->mi_async_lock);
2027         return;

2029 noasync:
2030         mutex_enter(&rp->r_statelock);
2031         rdc->entries = NULL;
2032         /*
2033          * Indicate that no one is trying to fill this entry and
2034          * it still needs to be filled.
2035          */
2036         rdc->flags &= ~RDDIR;
2037         rdc->flags |= RDDIRREQ;
2038         rddir4_cache_rele(rp, rdc);
2039         mutex_exit(&rp->r_statelock);
2040 }

2042 void
2043 nfs4_async_commit(vnode_t *vp, page_t *plist, offset3 offset, count3 count,
2044     cred_t *cr, void (*commit)(vnode_t *, page_t *, offset3, count3,
2045     cred_t *))
2046 {
2047         rnode4_t *rp;
2048         mntinfo4_t *mi;
2049         struct nfs4_async_reqs *args;
2050         page_t *pp;

2052         rp = VTOR4(vp);
2053         mi = VTOMI4(vp);

2055         /*
2056          * If we can't allocate a request structure, do the commit
2057          * operation synchronously in this thread's context.
2058          */
2059         if ((args = kmem_alloc(sizeof (*args), KM_NOSLEEP)) == NULL)
2060                 goto noasync;

2062         args->a_next = NULL;
2063 #ifdef DEBUG
2064         args->a_queuer = curthread;
2065 #endif
2066         VN_HOLD(vp);
2067         args->a_vp = vp;
2068         ASSERT(cr != NULL);
2069         crhold(cr);
2070         args->a_cred = cr;
2071         args->a_io = NFS4_COMMIT;
```

```
2072         args->a_nfs4_commit = commit;
2073         args->a_nfs4_plist = plist;
2074         args->a_nfs4_offset = offset;
2075         args->a_nfs4_count = count;

2077         mutex_enter(&mi->mi_async_lock);

2079         /*
2080          * If asyncio has been disabled, then make a synchronous request.
2081          * This check is done a second time in case async io was diabled
2082          * while this thread was blocked waiting for memory pressure to
2083          * reduce or for the queue to drain.
2084          */
2085         if (mi->mi_max_threads == 0) {
2086                 mutex_exit(&mi->mi_async_lock);

2088                 VN_RELE(vp);
2089                 crfree(cr);
2090                 kmem_free(args, sizeof (*args));
2091                 goto noasync;
2092         }

2094         /*
2095          * Link request structure into the async list and
2096          * wakeup async thread to do the i/o.
2097          */
2098         if (mi->mi_async_reqs[NFS4_COMMIT] == NULL) {
2099                 mi->mi_async_reqs[NFS4_COMMIT] = args;
2100                 mi->mi_async_tail[NFS4_COMMIT] = args;
2101         } else {
2102                 mi->mi_async_tail[NFS4_COMMIT]->a_next = args;
2103                 mi->mi_async_tail[NFS4_COMMIT] = args;
2104         }

2106         mutex_enter(&rp->r_statelock);
2107         rp->r_count++;
2108         mutex_exit(&rp->r_statelock);

2110         if (mi->mi_io_kstats) {
2111                 mutex_enter(&mi->mi_lock);
2112                 kstat_waitq_enter(KSTAT_IO_PTR(mi->mi_io_kstats));
2113                 mutex_exit(&mi->mi_lock);
2114         }

2116         mi->mi_async_req_count++;
2117         ASSERT(mi->mi_async_req_count != 0);
2118         cv_signal(&mi->mi_async_reqs_cv);
2119         mutex_exit(&mi->mi_async_lock);
2120         return;

2122 noasync:
2123         if (curproc == proc_pageout || curproc == proc_fsflush ||
2124             nfs_zone() != mi->mi_zone) {
2125                 while (plist != NULL) {
2126                         pp = plist;
2127                         page_sub(&plist, pp);
2128                         pp->p_fsdata = C_COMMIT;
2129                         page_unlock(pp);
2130                 }
2131                 return;
2132         }
2133         (*commit)(vp, plist, offset, count, cr);
2134 }

2136 /*
2137  * nfs4_async_inactive - hand off a VOP_INACTIVE call to a thread.  The
```



```
2138  * reference to the vnode is handed over to the thread; the caller should
2139  * no longer refer to the vnode.
2140  *
2141  * Unlike most of the async routines, this handoff is needed for
2142  * correctness reasons, not just performance.  So doing operations in the
2143  * context of the current thread is not an option.
2144  */
2145 void
2146 nfs4_async_inactive(vnode_t *vp, cred_t *cr)
2147 {
2148         mntinfo4_t *mi;
2149         struct nfs4_async_reqs *args;
2150         boolean_t signal_inactive_thread = B_FALSE;

2152         mi = VTOMI4(vp);

2154         args = kmem_alloc(sizeof (*args), KM_SLEEP);
2155         args->a_next = NULL;
2156 #ifdef DEBUG
2157         args->a_queuer = curthread;
2158 #endif
2159         args->a_vp = vp;
2160         ASSERT(cr != NULL);
2161         crhold(cr);
2162         args->a_cred = cr;
2163         args->a_io = NFS4_INACTIVE;

2165         /*
2166          * Note that we don’t check mi->mi_max_threads here, since we
2167          * *need* to get rid of this vnode regardless of whether someone
2168          * set nfs4_max_threads to zero in /etc/system.
2169          *
2170          * The manager thread knows about this and is willing to create
2171          * at least one thread to accommodate us.
2172          */
2173         mutex_enter(&mi->mi_async_lock);
2174         if (mi->mi_inactive_thread == NULL) {
2175                 rnode4_t *rp;
2176                 vnode_t *unldvp = NULL;
2177                 char *unlname;
2178                 cred_t *unlcred;

2180                 mutex_exit(&mi->mi_async_lock);
2181                 /*
2182                  * We just need to free up the memory associated with the
2183                  * vnode, which can be safely done from within the current
2184                  * context.
2185                  */
2186                 crfree(cr);     /* drop our reference */
2187                 kmem_free(args, sizeof (*args));
2188                 rp = VTOR4(vp);
2189                 mutex_enter(&rp->r_statelock);
2190                 if (rp->r_unldvp != NULL) {
2191                         unldvp = rp->r_unldvp;
2192                         rp->r_unldvp = NULL;
2193                         unlname = rp->r_unlname;
2194                         rp->r_unlname = NULL;
2195                         unlcred = rp->r_unlcred;
2196                         rp->r_unlcred = NULL;
2197                 }
2198                 mutex_exit(&rp->r_statelock);
2199                 /*
2200                  * No need to explicitly throw away any cached pages.  The
2201                  * eventual r4inactive() will attempt a synchronous
2202                  * VOP_PUTPAGE() which will immediately fail since the request
2203                  * is coming from the wrong zone, and then will proceed to call
```

```
2204                  * nfs4_invalidate_pages() which will clean things up for us.
2205                  *
2206                  * Throw away the delegation here so rp4_addfree()'s attempt to
2207                  * return any existing delegations becomes a no-op.
2208                  */
2209                 if (rp->r_deleg_type != OPEN_DELEGATE_NONE) {
2210                         (void) nfs_rw_enter_sig(&mi->mi_recovlock, RW_READER,
2211                             FALSE);
2212                         (void) nfs4delegreturn(rp, NFS4_DR_DISCARD);
2213                         nfs_rw_exit(&mi->mi_recovlock);
2214                 }
2215                 nfs4_clear_open_streams(rp);

2217                 rp4_addfree(rp, cr);
2218                 if (unldvp != NULL) {
2219                         kmem_free(unlname, MAXNAMELEN);
2220                         VN_RELE(unldvp);
2221                         crfree(unlcred);
2222                 }
2223                 return;
2224         }

2226         if (mi->mi_manager_thread == NULL) {
2227                 /*
2228                  * We want to talk to the inactive thread.
2229                  */
2230                 signal_inactive_thread = B_TRUE;
2231         }

2233         /*
2234          * Enqueue the vnode and wake up either the special thread (empty
2235          * list) or an async thread.
2236          */
2237         if (mi->mi_async_reqs[NFS4_INACTIVE] == NULL) {
2238                 mi->mi_async_reqs[NFS4_INACTIVE] = args;
2239                 mi->mi_async_tail[NFS4_INACTIVE] = args;
2240                 signal_inactive_thread = B_TRUE;
2241         } else {
2242                 mi->mi_async_tail[NFS4_INACTIVE]->a_next = args;
2243                 mi->mi_async_tail[NFS4_INACTIVE] = args;
2244         }
2245         if (signal_inactive_thread) {
2246                 cv_signal(&mi->mi_inact_req_cv);
2247         } else  {
2248                 mi->mi_async_req_count++;
2249                 ASSERT(mi->mi_async_req_count != 0);
2250                 cv_signal(&mi->mi_async_reqs_cv);
2251         }

2253         mutex_exit(&mi->mi_async_lock);
2254 }

2256 int
2257 writerp4(rnode4_t *rp, caddr_t base, int tcount, struct uio *uio, int pgcreated)
2258 {
2259         int pagecreate;
2260         int n;
2261         int saved_n;
2262         caddr_t saved_base;
2263         u_offset_t offset;
2264         int error;
2265         int sm_error;
2266         vnode_t *vp = RTOV(rp);

2268         ASSERT(tcount <= MAXBSIZE && tcount <= uio->uio_resid);
2269         ASSERT(nfs_rw_lock_held(&rp->r_rwlock, RW_WRITER));
```



```
2270         if (!vpm_enable) {
2271                 ASSERT(((uintptr_t)base & MAXBOFFSET) + tcount <= MAXBSIZE);
2272         }

2274         /*
2275          * Move bytes in at most PAGESIZE chunks. We must avoid
2276          * spanning pages in uiomove() because page faults may cause
2277          * the cache to be invalidated out from under us. The r_size is not
2278          * updated until after the uiomove. If we push the last page of a
2279          * file before r_size is correct, we will lose the data written past
2280          * the current (and invalid) r_size.
2281          */
2282         do {
2283                 offset = uio->uio_loffset;
2284                 pagecreate = 0;

2286                 /*
2287                  * n is the number of bytes required to satisfy the request
2288                  *   or the number of bytes to fill out the page.
2289                  */
2290                 n = (int)MIN((PAGESIZE - (offset & PAGEOFFSET)), tcount);

2292                 /*
2293                  * Check to see if we can skip reading in the page
2294                  * and just allocate the memory.  We can do this
2295                  * if we are going to rewrite the entire mapping
2296                  * or if we are going to write to or beyond the current
2297                  * end of file from the beginning of the mapping.
2298                  *
2299                  * The read of r_size is now protected by r_statelock.
2300                  */
2301                 mutex_enter(&rp->r_statelock);
2302                 /*
2303                  * When pgcreated is nonzero the caller has already done
2304                  * a segmap_getmapflt with forcefault 0 and S_WRITE. With
2305                  * segkpm this means we already have at least one page
2306                  * created and mapped at base.
2307                  */
2308                 pagecreate = pgcreated ||
2309                     ((offset & PAGEOFFSET) == 0 &&
2310                     (n == PAGESIZE || ((offset + n) >= rp->r_size)));

2312                 mutex_exit(&rp->r_statelock);

2314                 if (!vpm_enable && pagecreate) {
2315                         /*
2316                          * The last argument tells segmap_pagecreate() to
2317                          * always lock the page, as opposed to sometimes
2318                          * returning with the page locked. This way we avoid a
2319                          * fault on the ensuing uiomove(), but also
2320                          * more importantly (to fix bug 1094402) we can
2321                          * call segmap_fault() to unlock the page in all
2322                          * cases. An alternative would be to modify
2323                          * segmap_pagecreate() to tell us when it is
2324                          * locking a page, but that's a fairly major
2325                          * interface change.
2326                          */
2327                         if (pgcreated == 0)
2328                                 (void) segmap_pagecreate(segkmap, base,
2329                                     (uint_t)n, 1);
2330                         saved_base = base;
2331                         saved_n = n;
2332                 }

2334                 /*
2335                  * The number of bytes of data in the last page can not
```

```
2336                 * be accurately be determined while page is being
2337                 * uiomove'd to and the size of the file being updated.
2338                 * Thus, inform threads which need to know accurately
2339                 * how much data is in the last page of the file.  They
2340                 * will not do the i/o immediately, but will arrange for
2341                 * the i/o to happen later when this modify operation
2342                 * will have finished.
2343                 */
2344                ASSERT(!(rp->r_flags & R4MODINPROGRESS));
2345                mutex_enter(&rp->r_statelock);
2346                rp->r_flags |= R4MODINPROGRESS;
2347                rp->r_modaddr = (offset & MAXBMASK);
2348                mutex_exit(&rp->r_statelock);

2350                if (vpm_enable) {
2351                        /*
2352                         * Copy data. If new pages are created, part of
2353                         * the page that is not written will be initizliazed
2354                         * with zeros.
2355                         */
2356                        error = vpm_data_copy(vp, offset, n, uio,
2357                            !pagecreate, NULL, 0, S_WRITE);
2358                } else {
2359                        error = uiomove(base, n, UIO_WRITE, uio);
2360                }

2362                /*
2363                 * r_size is the maximum number of
2364                 * bytes known to be in the file.
2365                 * Make sure it is at least as high as the
2366                 * first unwritten byte pointed to by uio_loffset.
2367                 */
2368                mutex_enter(&rp->r_statelock);
2369                if (rp->r_size < uio->uio_loffset)
2370                        rp->r_size = uio->uio_loffset;
2371                rp->r_flags &= ~R4MODINPROGRESS;
2372                rp->r_flags |= R4DIRTY;
2373                mutex_exit(&rp->r_statelock);

2375                /* n = # of bytes written */
2376                n = (int)(uio->uio_loffset - offset);

2378                if (!vpm_enable) {
2379                        base += n;
2380                }

2382                tcount -= n;
2383                /*
2384                 * If we created pages w/o initializing them completely,
2385                 * we need to zero the part that wasn't set up.
2386                 * This happens on a most EOF write cases and if
2387                 * we had some sort of error during the uiomove.
2388                 */
2389                if (!vpm_enable && pagecreate) {
2390                        if ((uio->uio_loffset & PAGEOFFSET) || n == 0)
2391                                (void) kzero(base, PAGESIZE - n);

2393                        if (pgcreated) {
2394                                /*
2395                                 * Caller is responsible for this page,
2396                                 * it was not created in this loop.
2397                                 */
2398                                pgcreated = 0;
2399                        } else {
2400                                /*
2401                                 * For bug 1094402: segmap_pagecreate locks
```



```
2402                                 * page. Unlock it. This also unlocks the
2403                                 * pages allocated by page_create_va() in
2404                                 * segmap_pagecreate().
2405                                 */
2406                                sm_error = segmap_fault(kas.a_hat, segkmap,
2407                                    saved_base, saved_n,
2408                                    F_SOFTUNLOCK, S_WRITE);
2409                                if (error == 0)
2410                                        error = sm_error;
2411                        }
2412                }
2413        } while (tcount > 0 && error == 0);

2415        return (error);
2416 }

2418 int
2419 nfs4_putpages(vnode_t *vp, u_offset_t off, size_t len, int flags, cred_t *cr)
2420 {
2421        rnode4_t *rp;
2422        page_t *pp;
2423        u_offset_t eoff;
2424        u_offset_t io_off;
2425        size_t io_len;
2426        int error;
2427        int rdirty;
2428        int err;

2430        rp = VTOR4(vp);
2431        ASSERT(rp->r_count > 0);

2433        if (!nfs4_has_pages(vp))
2434                return (0);

2436        ASSERT(vp->v_type != VCHR);

2438        /*
2439         * If R4OUTOFSPACE is set, then all writes turn into B_INVAL
2440         * writes.  B_FORCE is set to force the VM system to actually
2441         * invalidate the pages, even if the i/o failed.  The pages
2442         * need to get invalidated because they can't be written out
2443         * because there isn't any space left on either the server's
2444         * file system or in the user's disk quota.  The B_FREE bit
2445         * is cleared to avoid confusion as to whether this is a
2446         * request to place the page on the freelist or to destroy
2447         * it.
2448         */
2449        if ((rp->r_flags & R4OUTOFSPACE) ||
2450            (vp->v_vfsp->vfs_flag & VFS_UNMOUNTED))
2451                flags = (flags & ~B_FREE) | B_INVAL | B_FORCE;

2453        if (len == 0) {
2454                /*
2455                 * If doing a full file synchronous operation, then clear
2456                 * the R4DIRTY bit.  If a page gets dirtied while the flush
2457                 * is happening, then R4DIRTY will get set again.  The
2458                 * R4DIRTY bit must get cleared before the flush so that
2459                 * we don't lose this information.
2460                 *
2461                 * If there are no full file async write operations
2462                 * pending and RDIRTY bit is set, clear it.
2463                 */
2464                if (off == (u_offset_t)0 &&
2465                    !(flags & B_ASYNC) &&
2466                    (rp->r_flags & R4DIRTY)) {
2467                        mutex_enter(&rp->r_statelock);
```

```
2468                        rdirty = (rp->r_flags & R4DIRTY);
2469                        rp->r_flags &= ~R4DIRTY;
2470                        mutex_exit(&rp->r_statelock);
2471                } else if (flags & B_ASYNC && off == (u_offset_t)0) {
2472                        mutex_enter(&rp->r_statelock);
2473                        if (rp->r_flags & R4DIRTY && rp->r_awcount == 0) {
2474                                rdirty = (rp->r_flags & R4DIRTY);
2475                                rp->r_flags &= ~R4DIRTY;
2476                        }
2477                        mutex_exit(&rp->r_statelock);
2478                } else
2479                        rdirty = 0;

2481                /*
2482                 * Search the entire vp list for pages >= off, and flush
2483                 * the dirty pages.
2484                 */
2485                error = pvn_vplist_dirty(vp, off, rp->r_putapage,
2486                    flags, cr);

2488                /*
2489                 * If an error occurred and the file was marked as dirty
2490                 * before and we aren't forcibly invalidating pages, then
2491                 * reset the R4DIRTY flag.
2492                 */
2493                if (error && rdirty &&
2494                    (flags & (B_INVAL | B_FORCE)) != (B_INVAL | B_FORCE)) {
2495                        mutex_enter(&rp->r_statelock);
2496                        rp->r_flags |= R4DIRTY;
2497                        mutex_exit(&rp->r_statelock);
2498                }
2499        } else {
2500                /*
2501                 * Do a range from [off...off + len) looking for pages
2502                 * to deal with.
2503                 */
2504                error = 0;
2505                io_len = 0;
2506                eoff = off + len;
2507                mutex_enter(&rp->r_statelock);
2508                for (io_off = off; io_off < eoff && io_off < rp->r_size;
2509                    io_off += io_len) {
2510                        mutex_exit(&rp->r_statelock);
2511                        /*
2512                         * If we are not invalidating, synchronously
2513                         * freeing or writing pages use the routine
2514                         * page_lookup_nowait() to prevent reclaiming
2515                         * them from the free list.
2516                         */
2517                        if ((flags & B_INVAL) || !(flags & B_ASYNC)) {
2518                                pp = page_lookup(vp, io_off,
2519                                    (flags & (B_INVAL | B_FREE)) ?
2520                                    SE_EXCL : SE_SHARED);
2521                        } else {
2522                                pp = page_lookup_nowait(vp, io_off,
2523                                    (flags & B_FREE) ? SE_EXCL : SE_SHARED);
2524                        }

2526                        if (pp == NULL || !pvn_getdirty(pp, flags))
2527                                io_len = PAGESIZE;
2528                        else {
2529                                err = (*rp->r_putapage)(vp, pp, &io_off,
2530                                    &io_len, flags, cr);
2531                                if (!error)
2532                                        error = err;
2533                                /*
```



```
2534                                 * "io_off" and "io_len" are returned as
2535                                 * the range of pages we actually wrote.
2536                                 * This allows us to skip ahead more quickly
2537                                 * since several pages may've been dealt
2538                                 * with by this iteration of the loop.
2539                                 */
2540                        }
2541                        mutex_enter(&rp->r_statelock);
2542                }
2543                mutex_exit(&rp->r_statelock);
2544        }

2546        return (error);
2547 }

2549 void
2550 nfs4_invalidate_pages(vnode_t *vp, u_offset_t off, cred_t *cr)
2551 {
2552        rnode4_t *rp;

2554        rp = VTOR4(vp);
2555        if (IS_SHADOW(vp, rp))
2556                vp = RTOV4(rp);
2557        mutex_enter(&rp->r_statelock);
2558        while (rp->r_flags & R4TRUNCATE)
2559                cv_wait(&rp->r_cv, &rp->r_statelock);
2560        rp->r_flags |= R4TRUNCATE;
2561        if (off == (u_offset_t)0) {
2562                rp->r_flags &= ~R4DIRTY;
2563                if (!(rp->r_flags & R4STALE))
2564                        rp->r_error = 0;
2565        }
2566        rp->r_truncaddr = off;
2567        mutex_exit(&rp->r_statelock);
2568        (void) pvn_vplist_dirty(vp, off, rp->r_putapage,
2569            B_INVAL | B_TRUNC, cr);
2570        mutex_enter(&rp->r_statelock);
2571        rp->r_flags &= ~R4TRUNCATE;
2572        cv_broadcast(&rp->r_cv);
2573        mutex_exit(&rp->r_statelock);
2574 }

2576 static int
2577 nfs4_mnt_kstat_update(kstat_t *ksp, int rw)
2578 {
2579        mntinfo4_t *mi;
2580        struct mntinfo_kstat *mik;
2581        vfs_t *vfsp;

2583        /* this is a read-only kstat. Bail out on a write */
2584        if (rw == KSTAT_WRITE)
2585                return (EACCES);


2588        /*
2589         * We don't want to wait here as kstat_chain_lock could be held by
2590         * dounmount(). dounmount() takes vfs_reflock before the chain lock
2591         * and thus could lead to a deadlock.
2592         */
2593        vfsp = (struct vfs *)ksp->ks_private;

2595        mi = VFTOMI4(vfsp);
2596        mik = (struct mntinfo_kstat *)ksp->ks_data;

2598        (void) strcpy(mik->mik_proto, mi->mi_curr_serv->sv_knconf->knc_proto);
```

```
2600         mik->mik_vers = (uint32_t)mi->mi_vers;
2601         mik->mik_flags = mi->mi_flags;
2602         /*
2603          * The sv_secdata holds the flavor the client specifies.
2604          * If the client uses default and a security negotiation
2605          * occurs, sv_currsec will point to the current flavor
2606          * selected from the server flavor list.
2607          * sv_currsec is NULL if no security negotiation takes place.
2608          */
2609         mik->mik_secmod = mi->mi_curr_serv->sv_currsec ?
2610             mi->mi_curr_serv->sv_currsec->secmod :
2611             mi->mi_curr_serv->sv_secdata->secmod;
2612         mik->mik_curread = (uint32_t)mi->mi_curread;
2613         mik->mik_curwrite = (uint32_t)mi->mi_curwrite;
2614         mik->mik_retrans = mi->mi_retrans;
2615         mik->mik_timeo = mi->mi_timeo;
2616         mik->mik_acregmin = HR2SEC(mi->mi_acregmin);
2617         mik->mik_acregmax = HR2SEC(mi->mi_acregmax);
2618         mik->mik_acdirmin = HR2SEC(mi->mi_acdirmin);
2619         mik->mik_acdirmax = HR2SEC(mi->mi_acdirmax);
2620         mik->mik_noresponse = (uint32_t)mi->mi_noresponse;
2621         mik->mik_failover = (uint32_t)mi->mi_failover;
2622         mik->mik_remap = (uint32_t)mi->mi_remap;

2624         (void) strcpy(mik->mik_curserver, mi->mi_curr_serv->sv_hostname);

2626         return (0);
2627 }

2629 void
2630 nfs4_mnt_kstat_init(struct vfs *vfsp)
2631 {
2632         mntinfo4_t *mi = VFTOMI4(vfsp);

2634         /*
2635          * PSARC 2001/697 Contract Private Interface
2636          * All nfs kstats are under SunMC contract
2637          * Please refer to the PSARC listed above and contact
2638          * SunMC before making any changes!
2639          *
2640          * Changes must be reviewed by Solaris File Sharing
2641          * Changes must be communicated to contract-2001-697@sun.com
2642          *
2643          */

2645         mi->mi_io_kstats = kstat_create_zone("nfs", getminor(vfsp->vfs_dev),
2646             NULL, "nfs", KSTAT_TYPE_IO, 1, 0, mi->mi_zone->zone_id);
2647         if (mi->mi_io_kstats) {
2648                 if (mi->mi_zone->zone_id != GLOBAL_ZONEID)
2649                         kstat_zone_add(mi->mi_io_kstats, GLOBAL_ZONEID);
2650                 mi->mi_io_kstats->ks_lock = &mi->mi_lock;
2651                 kstat_install(mi->mi_io_kstats);
2652         }

2654         if ((mi->mi_ro_kstats = kstat_create_zone("nfs",
2655             getminor(vfsp->vfs_dev), "mntinfo", "misc", KSTAT_TYPE_RAW,
2656             sizeof (struct mntinfo_kstat), 0, mi->mi_zone->zone_id)) != NULL) {
2657                 if (mi->mi_zone->zone_id != GLOBAL_ZONEID)
2658                         kstat_zone_add(mi->mi_ro_kstats, GLOBAL_ZONEID);
2659                 mi->mi_ro_kstats->ks_update = nfs4_mnt_kstat_update;
2660                 mi->mi_ro_kstats->ks_private = (void *)vfsp;
2661                 kstat_install(mi->mi_ro_kstats);
2662         }

2664         nfs4_mnt_recov_kstat_init(vfsp);
2665 }
```



```
2667 void
2668 nfs4_write_error(vnode_t *vp, int error, cred_t *cr)
2669 {
2670         mntinfo4_t *mi;
2671         clock_t now = ddi_get_lbolt();

2673         mi = VTOMI4(vp);
2674         /*
2675          * In case of forced unmount, do not print any messages
2676          * since it can flood the console with error messages.
2677          */
2678         if (mi->mi_vfsp->vfs_flag & VFS_UNMOUNTED)
2679                 return;

2681         /*
2682          * If the mount point is dead, not recoverable, do not
2683          * print error messages that can flood the console.
2684          */
2685         if (mi->mi_flags & MI4_RECOV_FAIL)
2686                 return;

2688         /*
2689          * No use in flooding the console with ENOSPC
2690          * messages from the same file system.
2691          */
2692         if ((error != ENOSPC && error != EDQUOT) ||
2693             now - mi->mi_printftime > 0) {
2694                 zoneid_t zoneid = mi->mi_zone->zone_id;

2696 #ifdef DEBUG
2697                 nfs_perror(error, "NFS%ld write error on host %s: %m.\n",
2698                     mi->mi_vers, VTOR4(vp)->r_server->sv_hostname, NULL);
2699 #else
2700                 nfs_perror(error, "NFS write error on host %s: %m.\n",
2701                     VTOR4(vp)->r_server->sv_hostname, NULL);
2702 #endif
2703                 if (error == ENOSPC || error == EDQUOT) {
2704                         zcmn_err(zoneid, CE_CONT,
2705                             "^File: userid=%d, groupid=%d\n",
2706                             crgetuid(cr), crgetgid(cr));
2707                         if (crgetuid(curthread->t_cred) != crgetuid(cr) ||
2708                             crgetgid(curthread->t_cred) != crgetgid(cr)) {
2709                                 zcmn_err(zoneid, CE_CONT,
2710                                     "^User: userid=%d, groupid=%d\n",
2711                                     crgetuid(curthread->t_cred),
2712                                     crgetgid(curthread->t_cred));
2713                         }
2714                         mi->mi_printftime = now +
2715                             nfs_write_error_interval * hz;
2716                 }
2717                 sfh4_printfhandle(VTOR4(vp)->r_fh);
2718 #ifdef DEBUG
2719                 if (error == EACCES) {
2720                         zcmn_err(zoneid, CE_CONT,
2721                             "nfs_bio: cred is%s kcred\n",
2722                             cr == kcred ? "" : " not");
2723                 }
2724 #endif
2725         }
2726 }

2728 /*
2729  * Return non-zero if the given file can be safely memory mapped.  Locks
2730  * are safe if whole-file (length and offset are both zero).
2731  */
```

```
2733 #define SAFE_LOCK(flk)  ((flk).l_start == 0 && (flk).l_len == 0)

2735 static int
2736 nfs4_safemap(const vnode_t *vp)
2737 {
2738         locklist_t      *llp, *next_llp;
2739         int             safe = 1;
2740         rnode4_t        *rp = VTOR4(vp);

2742         ASSERT(nfs_rw_lock_held(&rp->r_lkserlock, RW_WRITER));

2744         NFS4_DEBUG(nfs4_client_map_debug, (CE_NOTE, "nfs4_safemap: "
2745             "vp = %p", (void *)vp));

2747         /*
2748          * Review all the locks for the vnode, both ones that have been
2749          * acquired and ones that are pending.  We assume that
2750          * flk_active_locks_for_vp() has merged any locks that can be
2751          * merged (so that if a process has the entire file locked, it is
2752          * represented as a single lock).
2753          *
2754          * Note that we can't bail out of the loop if we find a non-safe
2755          * lock, because we have to free all the elements in the llp list.
2756          * We might be able to speed up this code slightly by not looking
2757          * at each lock's l_start and l_len fields once we've found a
2758          * non-safe lock.
2759          */

2761         llp = flk_active_locks_for_vp(vp);
2762         while (llp) {
2763                 NFS4_DEBUG(nfs4_client_map_debug, (CE_NOTE,
2764                     "nfs4_safemap: active lock (%" PRId64 ", %" PRId64 ")",
2765                     llp->ll_flock.l_start, llp->ll_flock.l_len));
2766                 if (!SAFE_LOCK(llp->ll_flock)) {
2767                         safe = 0;
2768                         NFS4_DEBUG(nfs4_client_map_debug, (CE_NOTE,
2769                             "nfs4_safemap: unsafe active lock (%" PRId64
2770                             ", %" PRId64 ")", llp->ll_flock.l_start,
2771                             llp->ll_flock.l_len));
2772                 }
2773                 next_llp = llp->ll_next;
2774                 VN_RELE(llp->ll_vp);
2775                 kmem_free(llp, sizeof (*llp));
2776                 llp = next_llp;
2777         }

2779         NFS4_DEBUG(nfs4_client_map_debug, (CE_NOTE, "nfs4_safemap: %s",
2780             safe ? "safe" : "unsafe"));
2781         return (safe);
2782 }

2784 /*
2785  * Return whether there is a lost LOCK or LOCKU queued up for the given
2786  * file that would make an mmap request unsafe.  cf. nfs4_safemap().
2787  */

2789 bool_t
2790 nfs4_map_lost_lock_conflict(vnode_t *vp)
2791 {
2792         bool_t conflict = FALSE;
2793         nfs4_lost_rqst_t *lrp;
2794         mntinfo4_t *mi = VTOMI4(vp);

2796         mutex_enter(&mi->mi_lock);
2797         for (lrp = list_head(&mi->mi_lost_state); lrp != NULL;
```



```
2798             lrp = list_next(&mi->mi_lost_state, lrp)) {
2799                 if (lrp->lr_op != OP_LOCK && lrp->lr_op != OP_LOCKU)
2800                         continue;
2801                 ASSERT(lrp->lr_vp != NULL);
2802                 if (!VOP_CMP(lrp->lr_vp, vp, NULL))
2803                         continue;       /* different file */
2804                 if (!SAFE_LOCK(*lrp->lr_flk)) {
2805                         conflict = TRUE;
2806                         break;
2807                 }
2808         }

2810         mutex_exit(&mi->mi_lock);
2811         return (conflict);
2812 }

2814 /*
2815  * nfs_lockcompletion:
2816  *
2817  * If the vnode has a lock that makes it unsafe to cache the file, mark it
2818  * as non cachable (set VNOCACHE bit).
2819  */

2821 void
2822 nfs4_lockcompletion(vnode_t *vp, int cmd)
2823 {
2824         rnode4_t *rp = VTOR4(vp);

2826         ASSERT(nfs_rw_lock_held(&rp->r_lkserlock, RW_WRITER));
2827         ASSERT(!IS_SHADOW(vp, rp));

2829         if (cmd == F_SETLK || cmd == F_SETLKW) {

2831                 if (!nfs4_safemap(vp)) {
2832                         mutex_enter(&vp->v_lock);
2833                         vp->v_flag |= VNOCACHE;
2834                         mutex_exit(&vp->v_lock);
2835                 } else {
2836                         mutex_enter(&vp->v_lock);
2837                         vp->v_flag &= ~VNOCACHE;
2838                         mutex_exit(&vp->v_lock);
2839                 }
2840         }
2841         /*
2842          * The cached attributes of the file are stale after acquiring
2843          * the lock on the file. They were updated when the file was
2844          * opened, but not updated when the lock was acquired. Therefore the
2845          * cached attributes are invalidated after the lock is obtained.
2846          */
2847         PURGE_ATTRCACHE4(vp);
2848 }

2850 /* ARGSUSED */
2851 static void *
2852 nfs4_mi_init(zoneid_t zoneid)
2853 {
2854         struct mi4_globals *mig;

2856         mig = kmem_alloc(sizeof (*mig), KM_SLEEP);
2857         mutex_init(&mig->mig_lock, NULL, MUTEX_DEFAULT, NULL);
2858         list_create(&mig->mig_list, sizeof (mntinfo4_t),
2859             offsetof(mntinfo4_t, mi_zone_node));
2860         mig->mig_destructor_called = B_FALSE;
2861         return (mig);
2862 }
```

```
2864 /*
2865  * Callback routine to tell all NFSv4 mounts in the zone to start tearing down
2866  * state and killing off threads.
2867  */
2868 /* ARGSUSED */
2869 static void
2870 nfs4_mi_shutdown(zoneid_t zoneid, void *data)
2871 {
2872         struct mi4_globals *mig = data;
2873         mntinfo4_t *mi;
2874         nfs4_server_t *np;

2876         NFS4_DEBUG(nfs4_client_zone_debug, (CE_NOTE,
2877             "nfs4_mi_shutdown zone %d\n", zoneid));
2878         ASSERT(mig != NULL);
2879         for (;;) {
2880                 mutex_enter(&mig->mig_lock);
2881                 mi = list_head(&mig->mig_list);
2882                 if (mi == NULL) {
2883                         mutex_exit(&mig->mig_lock);
2884                         break;
2885                 }

2887                 NFS4_DEBUG(nfs4_client_zone_debug, (CE_NOTE,
2888                     "nfs4_mi_shutdown stopping vfs %p\n", (void *)mi->mi_vfsp));
2889                 /*
2890                  * purge the DNLC for this filesystem
2891                  */
2892                 (void) dnlc_purge_vfsp(mi->mi_vfsp, 0);
2893                 /*
2894                  * Tell existing async worker threads to exit.
2895                  */
2896                 mutex_enter(&mi->mi_async_lock);
2897                 mi->mi_max_threads = 0;
2898                 NFS4_WAKEALL_ASYNC_WORKERS(mi->mi_async_work_cv);
2899                 /*
2900                  * Set the appropriate flags, signal and wait for both the
2901                  * async manager and the inactive thread to exit when they're
2902                  * done with their current work.
2903                  */
2904                 mutex_enter(&mi->mi_lock);
2905                 mi->mi_flags |= (MI4_ASYNC_MGR_STOP|MI4_DEAD);
2906                 mutex_exit(&mi->mi_lock);
2907                 mutex_exit(&mi->mi_async_lock);
2908                 if (mi->mi_manager_thread) {
2909                         nfs4_async_manager_stop(mi->mi_vfsp);
2910                 }
2911                 if (mi->mi_inactive_thread) {
2912                         mutex_enter(&mi->mi_async_lock);
2913                         cv_signal(&mi->mi_inact_req_cv);
2914                         /*
2915                          * Wait for the inactive thread to exit.
2916                          */
2917                         while (mi->mi_inactive_thread != NULL) {
2918                                 cv_wait(&mi->mi_async_cv, &mi->mi_async_lock);
2919                         }
2920                         mutex_exit(&mi->mi_async_lock);
2921                 }
2922                 /*
2923                  * Wait for the recovery thread to complete, that is, it will
2924                  * signal when it is done using the "mi" structure and about
2925                  * to exit
2926                  */
2927                 mutex_enter(&mi->mi_lock);
2928                 while (mi->mi_in_recovery > 0)
2929                         cv_wait(&mi->mi_cv_in_recov, &mi->mi_lock);
```



```
2930                 mutex_exit(&mi->mi_lock);
2931                 /*
2932                  * We're done when every mi has been done or the list is empty.
2933                  * This one is done, remove it from the list.
2934                  */
2935                 list_remove(&mig->mig_list, mi);
2936                 mutex_exit(&mig->mig_lock);
2937                 zone_rele_ref(&mi->mi_zone_ref, ZONE_REF_NFSV4);

2939                 /*
2940                  * Release hold on vfs and mi done to prevent race with zone
2941                  * shutdown. This releases the hold in nfs4_mi_zonelist_add.
2942                  */
2943                 VFS_RELE(mi->mi_vfsp);
2944                 MI4_RELE(mi);
2945         }
2946         /*
2947          * Tell each renew thread in the zone to exit
2948          */
2949         mutex_enter(&nfs4_server_lst_lock);
2950         for (np = nfs4_server_lst.forw; np != &nfs4_server_lst; np = np->forw) {
2951                 mutex_enter(&np->s_lock);
2952                 if (np->zoneid == zoneid) {
2953                         /*
2954                          * We add another hold onto the nfs4_server_t
2955                          * because this will make sure tha the nfs4_server_t
2956                          * stays around until nfs4_callback_fini_zone destroys
2957                          * the zone. This way, the renew thread can
2958                          * unconditionally release its holds on the
2959                          * nfs4_server_t.
2960                          */
2961                         np->s_refcnt++;
2962                         nfs4_mark_srv_dead(np);
2963                 }
2964                 mutex_exit(&np->s_lock);
2965         }
2966         mutex_exit(&nfs4_server_lst_lock);
2967 }

2969 static void
2970 nfs4_mi_free_globals(struct mi4_globals *mig)
2971 {
2972         list_destroy(&mig->mig_list);   /* makes sure the list is empty */
2973         mutex_destroy(&mig->mig_lock);
2974         kmem_free(mig, sizeof (*mig));
2975 }

2977 /* ARGSUSED */
2978 static void
2979 nfs4_mi_destroy(zoneid_t zoneid, void *data)
2980 {
2981         struct mi4_globals *mig = data;

2983         NFS4_DEBUG(nfs4_client_zone_debug, (CE_NOTE,
2984             "nfs4_mi_destroy zone %d\n", zoneid));
2985         ASSERT(mig != NULL);
2986         mutex_enter(&mig->mig_lock);
2987         if (list_head(&mig->mig_list) != NULL) {
2988                 /* Still waiting for VFS_FREEVFS() */
2989                 mig->mig_destructor_called = B_TRUE;
2990                 mutex_exit(&mig->mig_lock);
2991                 return;
2992         }
2993         nfs4_mi_free_globals(mig);
2994 }
```

```
2996 /*
2997  * Add an NFS mount to the per-zone list of NFS mounts.
2998  */
2999 void
3000 nfs4_mi_zonelist_add(mntinfo4_t *mi)
3001 {
3002         struct mi4_globals *mig;

3004         mig = zone_getspecific(mi4_list_key, mi->mi_zone);
3005         mutex_enter(&mig->mig_lock);
3006         list_insert_head(&mig->mig_list, mi);
3007         /*
3008          * hold added to eliminate race with zone shutdown -this will be
3009          * released in mi_shutdown
3010          */
3011         MI4_HOLD(mi);
3012         VFS_HOLD(mi->mi_vfsp);
3013         mutex_exit(&mig->mig_lock);
3014 }

3016 /*
3017  * Remove an NFS mount from the per-zone list of NFS mounts.
3018  */
3019 int
3020 nfs4_mi_zonelist_remove(mntinfo4_t *mi)
3021 {
3022         struct mi4_globals *mig;
3023         int ret = 0;

3025         mig = zone_getspecific(mi4_list_key, mi->mi_zone);
3026         mutex_enter(&mig->mig_lock);
3027         mutex_enter(&mi->mi_lock);
3028         /* if this mi is marked dead, then the zone already released it */
3029         if (!(mi->mi_flags & MI4_DEAD)) {
3030                 list_remove(&mig->mig_list, mi);
3031                 mutex_exit(&mi->mi_lock);

3033                 /* release the holds put on in zonelist_add(). */
3034                 VFS_RELE(mi->mi_vfsp);
3035                 MI4_RELE(mi);
3036                 ret = 1;
3037         } else {
3038                 mutex_exit(&mi->mi_lock);
3039         }

3041         /*
3042          * We can be called asynchronously by VFS_FREEVFS() after the zone
3043          * shutdown/destroy callbacks have executed; if so, clean up the zone's
3044          * mi globals.
3045          */
3046         if (list_head(&mig->mig_list) == NULL &&
3047             mig->mig_destructor_called == B_TRUE) {
3048                 nfs4_mi_free_globals(mig);
3049                 return (ret);
3050         }
3051         mutex_exit(&mig->mig_lock);
3052         return (ret);
3053 }

3055 void
3056 nfs_free_mi4(mntinfo4_t *mi)
3057 {
3058         nfs4_open_owner_t       *foop;
3059         nfs4_oo_hash_bucket_t   *bucketp;
3060         nfs4_debug_msg_t        *msgp;
3061         int i;
```



```
3062         servinfo4_t             *svp;

3064         /*
3065          * Code introduced here should be carefully evaluated to make
3066          * sure none of the freed resources are accessed either directly
3067          * or indirectly after freeing them. For eg: Introducing calls to
3068          * NFS4_DEBUG that use mntinfo4_t structure member after freeing
3069          * the structure members or other routines calling back into NFS
3070          * accessing freed mntinfo4_t structure member.
3071          */
3072         mutex_enter(&mi->mi_lock);
3073         ASSERT(mi->mi_recovthread == NULL);
3074         ASSERT(mi->mi_flags & MI4_ASYNC_MGR_STOP);
3075         mutex_exit(&mi->mi_lock);
3076         mutex_enter(&mi->mi_async_lock);
3077         ASSERT(mi->mi_threads[NFS4_ASYNC_QUEUE] == 0 &&
3078             mi->mi_threads[NFS4_ASYNC_PGOPS_QUEUE] == 0);
3079         ASSERT(mi->mi_manager_thread == NULL);
3080         mutex_exit(&mi->mi_async_lock);
3081         if (mi->mi_io_kstats) {
3082                 kstat_delete(mi->mi_io_kstats);
3083                 mi->mi_io_kstats = NULL;
3084         }
3085         if (mi->mi_ro_kstats) {
3086                 kstat_delete(mi->mi_ro_kstats);
3087                 mi->mi_ro_kstats = NULL;
3088         }
3089         if (mi->mi_recov_ksp) {
3090                 kstat_delete(mi->mi_recov_ksp);
3091                 mi->mi_recov_ksp = NULL;
3092         }
3093         mutex_enter(&mi->mi_msg_list_lock);
3094         while (msgp = list_head(&mi->mi_msg_list)) {
3095                 list_remove(&mi->mi_msg_list, msgp);
3096                 nfs4_free_msg(msgp);
3097         }
3098         mutex_exit(&mi->mi_msg_list_lock);
3099         list_destroy(&mi->mi_msg_list);
3100         if (mi->mi_fname != NULL)
3101                 fn_rele(&mi->mi_fname);
3102         if (mi->mi_rootfh != NULL)
3103                 sfh4_rele(&mi->mi_rootfh);
3104         if (mi->mi_srvparentfh != NULL)
3105                 sfh4_rele(&mi->mi_srvparentfh);
3106         svp = mi->mi_servers;
3107         sv4_free(svp);
3108         mutex_destroy(&mi->mi_lock);
3109         mutex_destroy(&mi->mi_async_lock);
3110         mutex_destroy(&mi->mi_msg_list_lock);
3111         nfs_rw_destroy(&mi->mi_recovlock);
3112         nfs_rw_destroy(&mi->mi_rename_lock);
3113         nfs_rw_destroy(&mi->mi_fh_lock);
3114         cv_destroy(&mi->mi_failover_cv);
3115         cv_destroy(&mi->mi_async_reqs_cv);
3116         cv_destroy(&mi->mi_async_work_cv[NFS4_ASYNC_QUEUE]);
3117         cv_destroy(&mi->mi_async_work_cv[NFS4_ASYNC_PGOPS_QUEUE]);
3118         cv_destroy(&mi->mi_async_cv);
3119         cv_destroy(&mi->mi_inact_req_cv);
3120         /*
3121          * Destroy the oo hash lists and mutexes for the cred hash table.
3122          */
3123         for (i = 0; i < NFS4_NUM_OO_BUCKETS; i++) {
3124                 bucketp = &(mi->mi_oo_list[i]);
3125                 /* Destroy any remaining open owners on the list */
3126                 foop = list_head(&bucketp->b_oo_hash_list);
3127                 while (foop != NULL) {
```

```
3128                        list_remove(&bucketp->b_oo_hash_list, foop);
3129                        nfs4_destroy_open_owner(foop);
3130                        foop = list_head(&bucketp->b_oo_hash_list);
3131                }
3132                list_destroy(&bucketp->b_oo_hash_list);
3133                mutex_destroy(&bucketp->b_lock);
3134        }
3135        /*
3136         * Empty and destroy the freed open owner list.
3137         */
3138        foop = list_head(&mi->mi_foo_list);
3139        while (foop != NULL) {
3140                list_remove(&mi->mi_foo_list, foop);
3141                nfs4_destroy_open_owner(foop);
3142                foop = list_head(&mi->mi_foo_list);
3143        }
3144        list_destroy(&mi->mi_foo_list);
3145        list_destroy(&mi->mi_bseqid_list);
3146        list_destroy(&mi->mi_lost_state);
3147        avl_destroy(&mi->mi_filehandles);
3148        kmem_free(mi, sizeof (*mi));
3149 }
3150 void
3151 mi_hold(mntinfo4_t *mi)
3152 {
3153        atomic_add_32(&mi->mi_count, 1);
3154        ASSERT(mi->mi_count != 0);
3155 }

3157 void
3158 mi_rele(mntinfo4_t *mi)
3159 {
3160        ASSERT(mi->mi_count != 0);
3161        if (atomic_add_32_nv(&mi->mi_count, -1) == 0) {
3162                nfs_free_mi4(mi);
3163        }
3164 }

3166 vnode_t    nfs4_xattr_notsupp_vnode;

3168 void
3169 nfs4_clnt_init(void)
3170 {
3171        nfs4_vnops_init();
3172        (void) nfs4_rnode_init();
3173        (void) nfs4_shadow_init();
3174        (void) nfs4_acache_init();
3175        (void) nfs4_subr_init();
3176        nfs4_acl_init();
3177        nfs_idmap_init();
3178        nfs4_callback_init();
3179        nfs4_secinfo_init();
3180 #ifdef  DEBUG
3181        tsd_create(&nfs4_tsd_key, NULL);
3182 #endif

3184        /*
3185         * Add a CPR callback so that we can update client
3186         * lease after a suspend and resume.
3187         */
3188        cid = callb_add(nfs4_client_cpr_callb, 0, CB_CL_CPR_RPC, "nfs4");

3190        zone_key_create(&mi4_list_key, nfs4_mi_init, nfs4_mi_shutdown,
3191            nfs4_mi_destroy);

3193        /*
```



```
3194         * Initialise the reference count of the notsupp xattr cache vnode to 1
3195         * so that it never goes away (VOP_INACTIVE isn't called on it).
3196         */
3197        nfs4_xattr_notsupp_vnode.v_count = 1;
3198 }

3200 void
3201 nfs4_clnt_fini(void)
3202 {
3203        (void) zone_key_delete(mi4_list_key);
3204        nfs4_vnops_fini();
3205        (void) nfs4_rnode_fini();
3206        (void) nfs4_shadow_fini();
3207        (void) nfs4_acache_fini();
3208        (void) nfs4_subr_fini();
3209        nfs_idmap_fini();
3210        nfs4_callback_fini();
3211        nfs4_secinfo_fini();
3212 #ifdef  DEBUG
3213        tsd_destroy(&nfs4_tsd_key);
3214 #endif
3215        if (cid)
3216                (void) callb_delete(cid);
3217 }

3219 /*ARGSUSED*/
3220 static boolean_t
3221 nfs4_client_cpr_callb(void *arg, int code)
3222 {
3223        /*
3224         * We get called for Suspend and Resume events.
3225         * For the suspend case we simply don't care!
3226         */
3227        if (code == CB_CODE_CPR_CHKPT) {
3228                return (B_TRUE);
3229        }

3231        /*
3232         * When we get to here we are in the process of
3233         * resuming the system from a previous suspend.
3234         */
3235        nfs4_client_resumed = gethrestime_sec();
3236        return (B_TRUE);
3237 }

3239 void
3240 nfs4_renew_lease_thread(nfs4_server_t *sp)
3241 {
3242        int     error = 0;
3243        time_t  tmp_last_renewal_time, tmp_time, tmp_now_time, kip_secs;
3244        clock_t tick_delay = 0;
3245        clock_t time_left = 0;
3246        callb_cpr_t cpr_info;
3247        kmutex_t cpr_lock;

3249        NFS4_DEBUG(nfs4_client_lease_debug, (CE_NOTE,
3250            "nfs4_renew_lease_thread: acting on sp 0x%p", (void*)sp));
3251        mutex_init(&cpr_lock, NULL, MUTEX_DEFAULT, NULL);
3252        CALLB_CPR_INIT(&cpr_info, &cpr_lock, callb_generic_cpr, "nfsv4Lease");

3254        mutex_enter(&sp->s_lock);
3255        /* sp->s_lease_time is set via a GETATTR */
3256        sp->last_renewal_time = gethrestime_sec();
3257        sp->lease_valid = NFS4_LEASE_UNINITIALIZED;
3258        ASSERT(sp->s_refcnt >= 1);
```

```
3260         for (;;) {
3261                 if (!sp->state_ref_count ||
3262                     sp->lease_valid != NFS4_LEASE_VALID) {
3263
3264                         kip_secs = MAX((sp->s_lease_time >> 1) -
3265                             (3 * sp->propagation_delay.tv_sec), 1);
3266
3267                         tick_delay = SEC_TO_TICK(kip_secs);
3268
3269                         NFS4_DEBUG(nfs4_client_lease_debug, (CE_NOTE,
3270                             "nfs4_renew_lease_thread: no renew : thread "
3271                             "wait %ld secs", kip_secs));
3272
3273                         NFS4_DEBUG(nfs4_client_lease_debug, (CE_NOTE,
3274                             "nfs4_renew_lease_thread: no renew : "
3275                             "state_ref_count %d, lease_valid %d",
3276                             sp->state_ref_count, sp->lease_valid));
3277
3278                         mutex_enter(&cpr_lock);
3279                         CALLB_CPR_SAFE_BEGIN(&cpr_info);
3280                         mutex_exit(&cpr_lock);
3281                         time_left = cv_reltimedwait(&sp->cv_thread_exit,
3282                             &sp->s_lock, tick_delay, TR_CLOCK_TICK);
3283                         mutex_enter(&cpr_lock);
3284                         CALLB_CPR_SAFE_END(&cpr_info, &cpr_lock);
3285                         mutex_exit(&cpr_lock);
3286
3287                         NFS4_DEBUG(nfs4_client_lease_debug, (CE_NOTE,
3288                             "nfs4_renew_lease_thread: no renew: "
3289                             "time left %ld", time_left));
3290
3291                         if (sp->s_thread_exit == NFS4_THREAD_EXIT)
3292                                 goto die;
3293                         continue;
3294                 }
3295
3296                 tmp_last_renewal_time = sp->last_renewal_time;
3297
3298                 tmp_time = gethrestime_sec() - sp->last_renewal_time +
3299                     (3 * sp->propagation_delay.tv_sec);
3300
3301                 NFS4_DEBUG(nfs4_client_lease_debug, (CE_NOTE,
3302                     "nfs4_renew_lease_thread: tmp_time %ld, "
3303                     "sp->last_renewal_time %ld", tmp_time,
3304                     sp->last_renewal_time));
3305
3306                 kip_secs = MAX((sp->s_lease_time >> 1) - tmp_time, 1);
3307
3308                 tick_delay = SEC_TO_TICK(kip_secs);
3309
3310                 NFS4_DEBUG(nfs4_client_lease_debug, (CE_NOTE,
3311                     "nfs4_renew_lease_thread: valid lease: sleep for %ld "
3312                     "secs", kip_secs));
3313
3314                 mutex_enter(&cpr_lock);
3315                 CALLB_CPR_SAFE_BEGIN(&cpr_info);
3316                 mutex_exit(&cpr_lock);
3317                 time_left = cv_reltimedwait(&sp->cv_thread_exit, &sp->s_lock,
3318                     tick_delay, TR_CLOCK_TICK);
3319                 mutex_enter(&cpr_lock);
3320                 CALLB_CPR_SAFE_END(&cpr_info, &cpr_lock);
3321                 mutex_exit(&cpr_lock);
3322
3323                 NFS4_DEBUG(nfs4_client_lease_debug, (CE_NOTE,
3324                     "nfs4_renew_lease_thread: valid lease: time left %ld :"
3325                     "sp last_renewal_time %ld, nfs4_client_resumed %ld, "
```



```
3326                     "tmp_last_renewal_time %ld", time_left,
3327                     sp->last_renewal_time, nfs4_client_resumed,
3328                     tmp_last_renewal_time));
3329
3330                 if (sp->s_thread_exit == NFS4_THREAD_EXIT)
3331                         goto die;
3332
3333                 if (tmp_last_renewal_time == sp->last_renewal_time ||
3334                     (nfs4_client_resumed != 0 &&
3335                     nfs4_client_resumed > sp->last_renewal_time)) {
3336                         /*
3337                          * Issue RENEW op since we haven't renewed the lease
3338                          * since we slept.
3339                          */
3340                         tmp_now_time = gethrestime_sec();
3341                         error = nfs4renew(sp);
3342                         /*
3343                          * Need to re-acquire sp's lock, nfs4renew()
3344                          * relinqueshes it.
3345                          */
3346                         mutex_enter(&sp->s_lock);
3347
3348                         /*
3349                          * See if someone changed s_thread_exit while we gave
3350                          * up s_lock.
3351                          */
3352                         if (sp->s_thread_exit == NFS4_THREAD_EXIT)
3353                                 goto die;
3354
3355                         if (!error) {
3356                                 /*
3357                                  * check to see if we implicitly renewed while
3358                                  * we waited for a reply for our RENEW call.
3359                                  */
3360                                 if (tmp_last_renewal_time ==
3361                                     sp->last_renewal_time) {
3362                                         /* no implicit renew came */
3363                                         sp->last_renewal_time = tmp_now_time;
3364                                 } else {
3365                                         NFS4_DEBUG(nfs4_client_lease_debug,
3366                                             (CE_NOTE, "renew_thread: did "
3367                                             "implicit renewal before reply "
3368                                             "from server for RENEW"));
3369                                 }
3370                         } else {
3371                                 /* figure out error */
3372                                 NFS4_DEBUG(nfs4_client_lease_debug, (CE_NOTE,
3373                                     "renew_thread: nfs4renew returned error"
3374                                     " %d", error));
3375                         }
3376
3377                 }
3378         }
3379
3380 die:
3381         NFS4_DEBUG(nfs4_client_lease_debug, (CE_NOTE,
3382             "nfs4_renew_lease_thread: thread exiting"));
3383
3384         while (sp->s_otw_call_count != 0) {
3385                 NFS4_DEBUG(nfs4_client_lease_debug, (CE_NOTE,
3386                     "nfs4_renew_lease_thread: waiting for outstanding "
3387                     "otw calls to finish for sp 0x%p, current "
3388                     "s_otw_call_count %d", (void *)sp,
3389                     sp->s_otw_call_count));
3390                 mutex_enter(&cpr_lock);
3391                 CALLB_CPR_SAFE_BEGIN(&cpr_info);
```

```
3392                    mutex_exit(&cpr_lock);
3393                    cv_wait(&sp->s_cv_otw_count, &sp->s_lock);
3394                    mutex_enter(&cpr_lock);
3395                    CALLB_CPR_SAFE_END(&cpr_info, &cpr_lock);
3396                    mutex_exit(&cpr_lock);
3397            }
3398            mutex_exit(&sp->s_lock);

3400            nfs4_server_rele(sp);           /* free the thread's reference */
3401            nfs4_server_rele(sp);           /* free the list's reference */
3402            sp = NULL;

3404    done:
3405            mutex_enter(&cpr_lock);
3406            CALLB_CPR_EXIT(&cpr_info);      /* drops cpr_lock */
3407            mutex_destroy(&cpr_lock);

3409            NFS4_DEBUG(nfs4_client_lease_debug, (CE_NOTE,
3410                "nfs4_renew_lease_thread: renew thread exit officially"));

3412            zthread_exit();
3413            /* NOT REACHED */
3414    }

3416    /*
3417     * Send out a RENEW op to the server.
3418     * Assumes sp is locked down.
3419     */
3420    static int
3421    nfs4renew(nfs4_server_t *sp)
3422    {
3423            COMPOUND4args_clnt args;
3424            COMPOUND4res_clnt res;
3425            nfs_argop4 argop[1];
3426            int doqueue = 1;
3427            int rpc_error;
3428            cred_t *cr;
3429            mntinfo4_t *mi;
3430            timespec_t prop_time, after_time;
3431            int needrecov = FALSE;
3432            nfs4_recov_state_t recov_state;
3433            nfs4_error_t e = { 0, NFS4_OK, RPC_SUCCESS };

3435            NFS4_DEBUG(nfs4_client_lease_debug, (CE_NOTE, "nfs4renew"));

3437            recov_state.rs_flags = 0;
3438            recov_state.rs_num_retry_despite_err = 0;

3440    recov_retry:
3441            mi = sp->mntinfo4_list;
3442            VFS_HOLD(mi->mi_vfsp);
3443            mutex_exit(&sp->s_lock);
3444            ASSERT(mi != NULL);

3446            e.error = nfs4_start_op(mi, NULL, NULL, &recov_state);
3447            if (e.error) {
3448                    VFS_RELE(mi->mi_vfsp);
3449                    return (e.error);
3450            }

3452            /* Check to see if we're dealing with a marked-dead sp */
3453            mutex_enter(&sp->s_lock);
3454            if (sp->s_thread_exit == NFS4_THREAD_EXIT) {
3455                    mutex_exit(&sp->s_lock);
3456                    nfs4_end_op(mi, NULL, NULL, &recov_state, needrecov);
3457                    VFS_RELE(mi->mi_vfsp);
```



```
3458                    return (0);
3459            }

3461            /* Make sure mi hasn't changed on us */
3462            if (mi != sp->mntinfo4_list) {
3463                    /* Must drop sp's lock to avoid a recursive mutex enter */
3464                    mutex_exit(&sp->s_lock);
3465                    nfs4_end_op(mi, NULL, NULL, &recov_state, needrecov);
3466                    VFS_RELE(mi->mi_vfsp);
3467                    mutex_enter(&sp->s_lock);
3468                    goto recov_retry;
3469            }
3470            mutex_exit(&sp->s_lock);

3472            args.ctag = TAG_RENEW;

3474            args.array_len = 1;
3475            args.array = argop;

3477            argop[0].argop = OP_RENEW;

3479            mutex_enter(&sp->s_lock);
3480            argop[0].nfs_argop4_u.oprenew.clientid = sp->clientid;
3481            cr = sp->s_cred;
3482            crhold(cr);
3483            mutex_exit(&sp->s_lock);

3485            ASSERT(cr != NULL);

3487            /* used to figure out RTT for sp */
3488            gethrestime(&prop_time);

3490            NFS4_DEBUG(nfs4_client_call_debug, (CE_NOTE,
3491                "nfs4renew: %s call, sp 0x%p", needrecov ? "recov" : "first",
3492                (void*)sp));
3493            NFS4_DEBUG(nfs4_client_lease_debug, (CE_NOTE, "before: %ld s %ld ns ",
3494                prop_time.tv_sec, prop_time.tv_nsec));

3496            DTRACE_PROBE2(nfs4__renew__start, nfs4_server_t *, sp,
3497                mntinfo4_t *, mi);

3499            rfs4call(mi, &args, &res, cr, &doqueue, 0, &e);
3500            crfree(cr);

3502            DTRACE_PROBE2(nfs4__renew__end, nfs4_server_t *, sp,
3503                mntinfo4_t *, mi);

3505            gethrestime(&after_time);

3507            mutex_enter(&sp->s_lock);
3508            sp->propagation_delay.tv_sec =
3509                MAX(1, after_time.tv_sec - prop_time.tv_sec);
3510            mutex_exit(&sp->s_lock);

3512            NFS4_DEBUG(nfs4_client_lease_debug, (CE_NOTE, "after : %ld s %ld ns ",
3513                after_time.tv_sec, after_time.tv_nsec));

3515            if (e.error == 0 && res.status == NFS4ERR_CB_PATH_DOWN) {
3516                    (void) xdr_free(xdr_COMPOUND4res_clnt, (caddr_t)&res);
3517                    nfs4_delegreturn_all(sp);
3518                    nfs4_end_op(mi, NULL, NULL, &recov_state, needrecov);
3519                    VFS_RELE(mi->mi_vfsp);
3520                    /*
3521                     * If the server returns CB_PATH_DOWN, it has renewed
3522                     * the lease and informed us that the callback path is
3523                     * down.  Since the lease is renewed, just return 0 and
```

```
3524                  * let the renew thread proceed as normal.
3525                  */
3526                 return (0);
3527         }

3529         needrecov = nfs4_needs_recovery(&e, FALSE, mi->mi_vfsp);
3530         if (!needrecov && e.error) {
3531                 nfs4_end_op(mi, NULL, NULL, &recov_state, needrecov);
3532                 VFS_RELE(mi->mi_vfsp);
3533                 return (e.error);
3534         }

3536         rpc_error = e.error;

3538         if (needrecov) {
3539                 NFS4_DEBUG(nfs4_client_recov_debug, (CE_NOTE,
3540                     "nfs4renew: initiating recovery\n"));

3542                 if (nfs4_start_recovery(&e, mi, NULL, NULL, NULL, NULL,
3543                     OP_RENEW, NULL, NULL, NULL) == FALSE) {
3544                         nfs4_end_op(mi, NULL, NULL, &recov_state, needrecov);
3545                         VFS_RELE(mi->mi_vfsp);
3546                         if (!e.error)
3547                                 (void) xdr_free(xdr_COMPOUND4res_clnt,
3548                                     (caddr_t)&res);
3549                         mutex_enter(&sp->s_lock);
3550                         goto recov_retry;
3551                 }
3552                 /* fall through for res.status case */
3553         }

3555         if (res.status) {
3556                 if (res.status == NFS4ERR_LEASE_MOVED) {
3557                         /*EMPTY*/
3558                         /*
3559                          * XXX need to try every mntinfo4 in sp->mntinfo4_list
3560                          * to renew the lease on that server
3561                          */
3562                 }
3563                 e.error = geterrno4(res.status);
3564         }

3566         if (!rpc_error)
3567                 (void) xdr_free(xdr_COMPOUND4res_clnt, (caddr_t)&res);

3569         nfs4_end_op(mi, NULL, NULL, &recov_state, needrecov);

3571         VFS_RELE(mi->mi_vfsp);

3573         return (e.error);
3574 }

3576 void
3577 nfs4_inc_state_ref_count(mntinfo4_t *mi)
3578 {
3579         nfs4_server_t   *sp;

3581         /* this locks down sp if it is found */
3582         sp = find_nfs4_server(mi);

3584         if (sp != NULL) {
3585                 nfs4_inc_state_ref_count_nolock(sp, mi);
3586                 mutex_exit(&sp->s_lock);
3587                 nfs4_server_rele(sp);
3588         }
3589 }
```



```
3591 /*
3592  * Bump the number of OPEN files (ie: those with state) so we know if this
3593  * nfs4_server has any state to maintain a lease for or not.
3594  *
3595  * Also, marks the nfs4_server's lease valid if it hasn't been done so already.
3596  */
3597 void
3598 nfs4_inc_state_ref_count_nolock(nfs4_server_t *sp, mntinfo4_t *mi)
3599 {
3600         ASSERT(mutex_owned(&sp->s_lock));

3602         sp->state_ref_count++;
3603         NFS4_DEBUG(nfs4_client_lease_debug, (CE_NOTE,
3604             "nfs4_inc_state_ref_count: state_ref_count now %d",
3605             sp->state_ref_count));

3607         if (sp->lease_valid == NFS4_LEASE_UNINITIALIZED)
3608                 sp->lease_valid = NFS4_LEASE_VALID;

3610         /*
3611          * If this call caused the lease to be marked valid and/or
3612          * took the state_ref_count from 0 to 1, then start the time
3613          * on lease renewal.
3614          */
3615         if (sp->lease_valid == NFS4_LEASE_VALID && sp->state_ref_count == 1)
3616                 sp->last_renewal_time = gethrestime_sec();

3618         /* update the number of open files for mi */
3619         mi->mi_open_files++;
3620 }

3622 void
3623 nfs4_dec_state_ref_count(mntinfo4_t *mi)
3624 {
3625         nfs4_server_t   *sp;

3627         /* this locks down sp if it is found */
3628         sp = find_nfs4_server_all(mi, 1);

3630         if (sp != NULL) {
3631                 nfs4_dec_state_ref_count_nolock(sp, mi);
3632                 mutex_exit(&sp->s_lock);
3633                 nfs4_server_rele(sp);
3634         }
3635 }

3637 /*
3638  * Decrement the number of OPEN files (ie: those with state) so we know if
3639  * this nfs4_server has any state to maintain a lease for or not.
3640  */
3641 void
3642 nfs4_dec_state_ref_count_nolock(nfs4_server_t *sp, mntinfo4_t *mi)
3643 {
3644         ASSERT(mutex_owned(&sp->s_lock));
3645         ASSERT(sp->state_ref_count != 0);
3646         sp->state_ref_count--;

3648         NFS4_DEBUG(nfs4_client_lease_debug, (CE_NOTE,
3649             "nfs4_dec_state_ref_count: state ref count now %d",
3650             sp->state_ref_count));

3652         mi->mi_open_files--;
3653         NFS4_DEBUG(nfs4_client_lease_debug, (CE_NOTE,
3654             "nfs4_dec_state_ref_count: mi open files %d, v4 flags 0x%x",
3655             mi->mi_open_files, mi->mi_flags));
```

```
3657         /* We don't have to hold the mi_lock to test mi_flags */
3658         if (mi->mi_open_files == 0 &&
3659             (mi->mi_flags & MI4_REMOVE_ON_LAST_CLOSE)) {
3660                 NFS4_DEBUG(nfs4_client_lease_debug, (CE_NOTE,
3661                     "nfs4_dec_state_ref_count: remove mntinfo4 %p since "
3662                     "we have closed the last open file", (void*)mi));
3663                 nfs4_remove_mi_from_server(mi, sp);
3664         }
3665 }

3667 bool_t
3668 inlease(nfs4_server_t *sp)
3669 {
3670         bool_t result;

3672         ASSERT(mutex_owned(&sp->s_lock));

3674         if (sp->lease_valid == NFS4_LEASE_VALID &&
3675             gethrestime_sec() < sp->last_renewal_time + sp->s_lease_time)
3676                 result = TRUE;
3677         else
3678                 result = FALSE;

3680         return (result);
3681 }


3684 /*
3685  * Return non-zero if the given nfs4_server_t is going through recovery.
3686  */

3688 int
3689 nfs4_server_in_recovery(nfs4_server_t *sp)
3690 {
3691         return (nfs_rw_lock_held(&sp->s_recovlock, RW_WRITER));
3692 }

3694 /*
3695  * Compare two shared filehandle objects.  Returns -1, 0, or +1, if the
3696  * first is less than, equal to, or greater than the second.
3697  */

3699 int
3700 sfh4cmp(const void *p1, const void *p2)
3701 {
3702         const nfs4_sharedfh_t *sfh1 = (const nfs4_sharedfh_t *)p1;
3703         const nfs4_sharedfh_t *sfh2 = (const nfs4_sharedfh_t *)p2;

3705         return (nfs4cmpfh(&sfh1->sfh_fh, &sfh2->sfh_fh));
3706 }

3708 /*
3709  * Create a table for shared filehandle objects.
3710  */

3712 void
3713 sfh4_createtab(avl_tree_t *tab)
3714 {
3715         avl_create(tab, sfh4cmp, sizeof (nfs4_sharedfh_t),
3716             offsetof(nfs4_sharedfh_t, sfh_tree));
3717 }

3719 /*
3720  * Return a shared filehandle object for the given filehandle.  The caller
3721  * is responsible for eventually calling sfh4_rele().
```



```
3722  */

3724 nfs4_sharedfh_t *
3725 sfh4_put(const nfs_fh4 *fh, mntinfo4_t *mi, nfs4_sharedfh_t *key)
3726 {
3727         nfs4_sharedfh_t *sfh, *nsfh;
3728         avl_index_t where;
3729         nfs4_sharedfh_t skey;

3731         if (!key) {
3732                 skey.sfh_fh = *fh;
3733                 key = &skey;
3734         }

3736         nsfh = kmem_alloc(sizeof (nfs4_sharedfh_t), KM_SLEEP);
3737         nsfh->sfh_fh.nfs_fh4_len = fh->nfs_fh4_len;
3738         /*
3739          * We allocate the largest possible filehandle size because it's
3740          * not that big, and it saves us from possibly having to resize the
3741          * buffer later.
3742          */
3743         nsfh->sfh_fh.nfs_fh4_val = kmem_alloc(NFS4_FHSIZE, KM_SLEEP);
3744         bcopy(fh->nfs_fh4_val, nsfh->sfh_fh.nfs_fh4_val, fh->nfs_fh4_len);
3745         mutex_init(&nsfh->sfh_lock, NULL, MUTEX_DEFAULT, NULL);
3746         nsfh->sfh_refcnt = 1;
3747         nsfh->sfh_flags = SFH4_IN_TREE;
3748         nsfh->sfh_mi = mi;
3749         NFS4_DEBUG(nfs4_sharedfh_debug, (CE_NOTE, "sfh4_get: new object (%p)",
3750             (void *)nsfh));

3752         (void) nfs_rw_enter_sig(&mi->mi_fh_lock, RW_WRITER, 0);
3753         sfh = avl_find(&mi->mi_filehandles, key, &where);
3754         if (sfh != NULL) {
3755                 mutex_enter(&sfh->sfh_lock);
3756                 sfh->sfh_refcnt++;
3757                 mutex_exit(&sfh->sfh_lock);
3758                 nfs_rw_exit(&mi->mi_fh_lock);
3759                 /* free our speculative allocs */
3760                 kmem_free(nsfh->sfh_fh.nfs_fh4_val, NFS4_FHSIZE);
3761                 kmem_free(nsfh, sizeof (nfs4_sharedfh_t));
3762                 return (sfh);
3763         }

3765         avl_insert(&mi->mi_filehandles, nsfh, where);
3766         nfs_rw_exit(&mi->mi_fh_lock);

3768         return (nsfh);
3769 }

3771 /*
3772  * Return a shared filehandle object for the given filehandle.  The caller
3773  * is responsible for eventually calling sfh4_rele().
3774  */

3776 nfs4_sharedfh_t *
3777 sfh4_get(const nfs_fh4 *fh, mntinfo4_t *mi)
3778 {
3779         nfs4_sharedfh_t *sfh;
3780         nfs4_sharedfh_t key;

3782         ASSERT(fh->nfs_fh4_len <= NFS4_FHSIZE);

3784 #ifdef DEBUG
3785         if (nfs4_sharedfh_debug) {
3786                 nfs4_fhandle_t fhandle;
```

```
3788                 fhandle.fh_len = fh->nfs_fh4_len;
3789                 bcopy(fh->nfs_fh4_val, fhandle.fh_buf, fhandle.fh_len);
3790                 zcmn_err(mi->mi_zone->zone_id, CE_NOTE, "sfh4_get:");
3791                 nfs4_printfhandle(&fhandle);
3792         }
3793 #endif

3795         /*
3796          * If there's already an object for the given filehandle, bump the
3797          * reference count and return it.  Otherwise, create a new object
3798          * and add it to the AVL tree.
3799          */

3801         key.sfh_fh = *fh;

3803         (void) nfs_rw_enter_sig(&mi->mi_fh_lock, RW_READER, 0);
3804         sfh = avl_find(&mi->mi_filehandles, &key, NULL);
3805         if (sfh != NULL) {
3806                 mutex_enter(&sfh->sfh_lock);
3807                 sfh->sfh_refcnt++;
3808                 NFS4_DEBUG(nfs4_sharedfh_debug, (CE_NOTE,
3809                     "sfh4_get: found existing %p, new refcnt=%d",
3810                     (void *)sfh, sfh->sfh_refcnt));
3811                 mutex_exit(&sfh->sfh_lock);
3812                 nfs_rw_exit(&mi->mi_fh_lock);
3813                 return (sfh);
3814         }
3815         nfs_rw_exit(&mi->mi_fh_lock);

3817         return (sfh4_put(fh, mi, &key));
3818 }

3820 /*
3821  * Get a reference to the given shared filehandle object.
3822  */

3824 void
3825 sfh4_hold(nfs4_sharedfh_t *sfh)
3826 {
3827         ASSERT(sfh->sfh_refcnt > 0);

3829         mutex_enter(&sfh->sfh_lock);
3830         sfh->sfh_refcnt++;
3831         NFS4_DEBUG(nfs4_sharedfh_debug,
3832             (CE_NOTE, "sfh4_hold %p, new refcnt=%d",
3833             (void *)sfh, sfh->sfh_refcnt));
3834         mutex_exit(&sfh->sfh_lock);
3835 }

3837 /*
3838  * Release a reference to the given shared filehandle object and null out
3839  * the given pointer.
3840  */

3842 void
3843 sfh4_rele(nfs4_sharedfh_t **sfhpp)
3844 {
3845         mntinfo4_t *mi;
3846         nfs4_sharedfh_t *sfh = *sfhpp;

3848         ASSERT(sfh->sfh_refcnt > 0);

3850         mutex_enter(&sfh->sfh_lock);
3851         if (sfh->sfh_refcnt > 1) {
3852                 sfh->sfh_refcnt--;
3853                 NFS4_DEBUG(nfs4_sharedfh_debug, (CE_NOTE,
```



```
3854                     "sfh4_rele %p, new refcnt=%d",
3855                     (void *)sfh, sfh->sfh_refcnt));
3856                 mutex_exit(&sfh->sfh_lock);
3857                 goto finish;
3858         }
3859         mutex_exit(&sfh->sfh_lock);

3861         /*
3862          * Possibly the last reference, so get the lock for the table in
3863          * case it's time to remove the object from the table.
3864          */
3865         mi = sfh->sfh_mi;
3866         (void) nfs_rw_enter_sig(&mi->mi_fh_lock, RW_WRITER, 0);
3867         mutex_enter(&sfh->sfh_lock);
3868         sfh->sfh_refcnt--;
3869         if (sfh->sfh_refcnt > 0) {
3870                 NFS4_DEBUG(nfs4_sharedfh_debug, (CE_NOTE,
3871                     "sfh4_rele %p, new refcnt=%d",
3872                     (void *)sfh, sfh->sfh_refcnt));
3873                 mutex_exit(&sfh->sfh_lock);
3874                 nfs_rw_exit(&mi->mi_fh_lock);
3875                 goto finish;
3876         }

3878         NFS4_DEBUG(nfs4_sharedfh_debug, (CE_NOTE,
3879             "sfh4_rele %p, last ref", (void *)sfh));
3880         if (sfh->sfh_flags & SFH4_IN_TREE) {
3881                 avl_remove(&mi->mi_filehandles, sfh);
3882                 sfh->sfh_flags &= ~SFH4_IN_TREE;
3883         }
3884         mutex_exit(&sfh->sfh_lock);
3885         nfs_rw_exit(&mi->mi_fh_lock);
3886         mutex_destroy(&sfh->sfh_lock);
3887         kmem_free(sfh->sfh_fh.nfs_fh4_val, NFS4_FHSIZE);
3888         kmem_free(sfh, sizeof (nfs4_sharedfh_t));

3890 finish:
3891         *sfhpp = NULL;
3892 }

3894 /*
3895  * Update the filehandle for the given shared filehandle object.
3896  */

3898 int nfs4_warn_dupfh = 0;        /* if set, always warn about dup fhs below */

3900 void
3901 sfh4_update(nfs4_sharedfh_t *sfh, const nfs_fh4 *newfh)
3902 {
3903         mntinfo4_t *mi = sfh->sfh_mi;
3904         nfs4_sharedfh_t *dupsfh;
3905         avl_index_t where;
3906         nfs4_sharedfh_t key;

3908 #ifdef DEBUG
3909         mutex_enter(&sfh->sfh_lock);
3910         ASSERT(sfh->sfh_refcnt > 0);
3911         mutex_exit(&sfh->sfh_lock);
3912 #endif
3913         ASSERT(newfh->nfs_fh4_len <= NFS4_FHSIZE);

3915         /*
3916          * The basic plan is to remove the shared filehandle object from
3917          * the table, update it to have the new filehandle, then reinsert
3918          * it.
3919          */
```

```
3921         (void) nfs_rw_enter_sig(&mi->mi_fh_lock, RW_WRITER, 0);
3922         mutex_enter(&sfh->sfh_lock);
3923         if (sfh->sfh_flags & SFH4_IN_TREE) {
3924                 avl_remove(&mi->mi_filehandles, sfh);
3925                 sfh->sfh_flags &= ~SFH4_IN_TREE;
3926         }
3927         mutex_exit(&sfh->sfh_lock);
3928         sfh->sfh_fh.nfs_fh4_len = newfh->nfs_fh4_len;
3929         bcopy(newfh->nfs_fh4_val, sfh->sfh_fh.nfs_fh4_val,
3930             sfh->sfh_fh.nfs_fh4_len);

3932         /*
3933          * XXX If there is already a shared filehandle object with the new
3934          * filehandle, we're in trouble, because the rnode code assumes
3935          * that there is only one shared filehandle object for a given
3936          * filehandle.  So issue a warning (for read-write mounts only)
3937          * and don't try to re-insert the given object into the table.
3938          * Hopefully the given object will quickly go away and everyone
3939          * will use the new object.
3940          */
3941         key.sfh_fh = *newfh;
3942         dupsfh = avl_find(&mi->mi_filehandles, &key, &where);
3943         if (dupsfh != NULL) {
3944                 if (!(mi->mi_vfsp->vfs_flag & VFS_RDONLY) || nfs4_warn_dupfh) {
3945                         zcmn_err(mi->mi_zone->zone_id, CE_WARN, "sfh4_update: "
3946                             "duplicate filehandle detected");
3947                         sfh4_printfhandle(dupsfh);
3948                 }
3949         } else {
3950                 avl_insert(&mi->mi_filehandles, sfh, where);
3951                 mutex_enter(&sfh->sfh_lock);
3952                 sfh->sfh_flags |= SFH4_IN_TREE;
3953                 mutex_exit(&sfh->sfh_lock);
3954         }
3955         nfs_rw_exit(&mi->mi_fh_lock);
3956 }

3958 /*
3959  * Copy out the current filehandle for the given shared filehandle object.
3960  */

3962 void
3963 sfh4_copyval(const nfs4_sharedfh_t *sfh, nfs4_fhandle_t *fhp)
3964 {
3965         mntinfo4_t *mi = sfh->sfh_mi;

3967         ASSERT(sfh->sfh_refcnt > 0);

3969         (void) nfs_rw_enter_sig(&mi->mi_fh_lock, RW_READER, 0);
3970         fhp->fh_len = sfh->sfh_fh.nfs_fh4_len;
3971         ASSERT(fhp->fh_len <= NFS4_FHSIZE);
3972         bcopy(sfh->sfh_fh.nfs_fh4_val, fhp->fh_buf, fhp->fh_len);
3973         nfs_rw_exit(&mi->mi_fh_lock);
3974 }

3976 /*
3977  * Print out the filehandle for the given shared filehandle object.
3978  */

3980 void
3981 sfh4_printfhandle(const nfs4_sharedfh_t *sfh)
3982 {
3983         nfs4_fhandle_t fhandle;

3985         sfh4_copyval(sfh, &fhandle);
```



```
3986         nfs4_printfhandle(&fhandle);
3987 }

3989 /*
3990  * Compare 2 fnames.  Returns -1 if the first is "less" than the second, 0
3991  * if they're the same, +1 if the first is "greater" than the second.  The
3992  * caller (or whoever's calling the AVL package) is responsible for
3993  * handling locking issues.
3994  */

3996 static int
3997 fncmp(const void *p1, const void *p2)
3998 {
3999         const nfs4_fname_t *f1 = p1;
4000         const nfs4_fname_t *f2 = p2;
4001         int res;

4003         res = strcmp(f1->fn_name, f2->fn_name);
4004         /*
4005          * The AVL package wants +/-1, not arbitrary positive or negative
4006          * integers.
4007          */
4008         if (res > 0)
4009                 res = 1;
4010         else if (res < 0)
4011                 res = -1;
4012         return (res);
4013 }

4015 /*
4016  * Get or create an fname with the given name, as a child of the given
4017  * fname.  The caller is responsible for eventually releasing the reference
4018  * (fn_rele()).  parent may be NULL.
4019  */

4021 nfs4_fname_t *
4022 fn_get(nfs4_fname_t *parent, char *name, nfs4_sharedfh_t *sfh)
4023 {
4024         nfs4_fname_t key;
4025         nfs4_fname_t *fnp;
4026         avl_index_t where;

4028         key.fn_name = name;

4030         /*
4031          * If there's already an fname registered with the given name, bump
4032          * its reference count and return it.  Otherwise, create a new one
4033          * and add it to the parent's AVL tree.
4034          *
4035          * fname entries we are looking for should match both name
4036          * and sfh stored in the fname.
4037          */
4038 again:
4039         if (parent != NULL) {
4040                 mutex_enter(&parent->fn_lock);
4041                 fnp = avl_find(&parent->fn_children, &key, &where);
4042                 if (fnp != NULL) {
4043                         /*
4044                          * This hold on fnp is released below later,
4045                          * in case this is not the fnp we want.
4046                          */
4047                         fn_hold(fnp);

4049                         if (fnp->fn_sfh == sfh) {
4050                                 /*
4051                                  * We have found our entry.
```

```
4052                                 * put an hold and return it.
4053                                 */
4054                                mutex_exit(&parent->fn_lock);
4055                                return (fnp);
4056                        }
4057
4058                        /*
4059                         * We have found an entry that has a mismatching
4060                         * fn_sfh. This could be a stale entry due to
4061                         * server side rename. We will remove this entry
4062                         * and make sure no such entries exist.
4063                         */
4064                        mutex_exit(&parent->fn_lock);
4065                        mutex_enter(&fnp->fn_lock);
4066                        if (fnp->fn_parent == parent) {
4067                                /*
4068                                 * Remove ourselves from parent's
4069                                 * fn_children tree.
4070                                 */
4071                                mutex_enter(&parent->fn_lock);
4072                                avl_remove(&parent->fn_children, fnp);
4073                                mutex_exit(&parent->fn_lock);
4074                                fn_rele(&fnp->fn_parent);
4075                        }
4076                        mutex_exit(&fnp->fn_lock);
4077                        fn_rele(&fnp);
4078                        goto again;
4079                }
4080        }
4081
4082        fnp = kmem_alloc(sizeof (nfs4_fname_t), KM_SLEEP);
4083        mutex_init(&fnp->fn_lock, NULL, MUTEX_DEFAULT, NULL);
4084        fnp->fn_parent = parent;
4085        if (parent != NULL)
4086                fn_hold(parent);
4087        fnp->fn_len = strlen(name);
4088        ASSERT(fnp->fn_len < MAXNAMELEN);
4089        fnp->fn_name = kmem_alloc(fnp->fn_len + 1, KM_SLEEP);
4090        (void) strcpy(fnp->fn_name, name);
4091        fnp->fn_refcnt = 1;
4092
4093        /*
4094         * This hold on sfh is later released
4095         * when we do the final fn_rele() on this fname.
4096         */
4097        sfh4_hold(sfh);
4098        fnp->fn_sfh = sfh;
4099
4100        avl_create(&fnp->fn_children, fncmp, sizeof (nfs4_fname_t),
4101            offsetof(nfs4_fname_t, fn_tree));
4102        NFS4_DEBUG(nfs4_fname_debug, (CE_NOTE,
4103            "fn_get %p:%s, a new nfs4_fname_t!",
4104            (void *)fnp, fnp->fn_name));
4105        if (parent != NULL) {
4106                avl_insert(&parent->fn_children, fnp, where);
4107                mutex_exit(&parent->fn_lock);
4108        }
4109
4110        return (fnp);
4111 }
4112
4113 void
4114 fn_hold(nfs4_fname_t *fnp)
4115 {
4116        atomic_add_32(&fnp->fn_refcnt, 1);
4117        NFS4_DEBUG(nfs4_fname_debug, (CE_NOTE,
```



```
4118            "fn_hold %p:%s, new refcnt=%d",
4119            (void *)fnp, fnp->fn_name, fnp->fn_refcnt));
4120 }
4121
4122 /*
4123  * Decrement the reference count of the given fname, and destroy it if its
4124  * reference count goes to zero.  Nulls out the given pointer.
4125  */
4126
4127 void
4128 fn_rele(nfs4_fname_t **fnpp)
4129 {
4130        nfs4_fname_t *parent;
4131        uint32_t newref;
4132        nfs4_fname_t *fnp;
4133
4134 recur:
4135        fnp = *fnpp;
4136        *fnpp = NULL;
4137
4138        mutex_enter(&fnp->fn_lock);
4139        parent = fnp->fn_parent;
4140        if (parent != NULL)
4141                mutex_enter(&parent->fn_lock);  /* prevent new references */
4142        newref = atomic_add_32_nv(&fnp->fn_refcnt, -1);
4143        if (newref > 0) {
4144                NFS4_DEBUG(nfs4_fname_debug, (CE_NOTE,
4145                    "fn_rele %p:%s, new refcnt=%d",
4146                    (void *)fnp, fnp->fn_name, fnp->fn_refcnt));
4147                if (parent != NULL)
4148                        mutex_exit(&parent->fn_lock);
4149                mutex_exit(&fnp->fn_lock);
4150                return;
4151        }
4152
4153        NFS4_DEBUG(nfs4_fname_debug, (CE_NOTE,
4154            "fn_rele %p:%s, last reference, deleting...",
4155            (void *)fnp, fnp->fn_name));
4156        if (parent != NULL) {
4157                avl_remove(&parent->fn_children, fnp);
4158                mutex_exit(&parent->fn_lock);
4159        }
4160        kmem_free(fnp->fn_name, fnp->fn_len + 1);
4161        sfh4_rele(&fnp->fn_sfh);
4162        mutex_destroy(&fnp->fn_lock);
4163        avl_destroy(&fnp->fn_children);
4164        kmem_free(fnp, sizeof (nfs4_fname_t));
4165        /*
4166         * Recursivly fn_rele the parent.
4167         * Use goto instead of a recursive call to avoid stack overflow.
4168         */
4169        if (parent != NULL) {
4170                fnpp = &parent;
4171                goto recur;
4172        }
4173 }
4174
4175 /*
4176  * Returns the single component name of the given fname, in a MAXNAMELEN
4177  * string buffer, which the caller is responsible for freeing.  Note that
4178  * the name may become invalid as a result of fn_move().
4179  */
4180
4181 char *
4182 fn_name(nfs4_fname_t *fnp)
4183 {
```

```
4184         char *name;

4186         ASSERT(fnp->fn_len < MAXNAMELEN);
4187         name = kmem_alloc(MAXNAMELEN, KM_SLEEP);
4188         mutex_enter(&fnp->fn_lock);
4189         (void) strcpy(name, fnp->fn_name);
4190         mutex_exit(&fnp->fn_lock);

4192         return (name);
4193 }


4196 /*
4197  * fn_path_realloc
4198  *
4199  * This function, used only by fn_path, constructs
4200  * a new string which looks like "prepend" + "/" + "current".
4201  * by allocating a new string and freeing the old one.
4202  */
4203 static void
4204 fn_path_realloc(char **curses, char *prepend)
4205 {
4206         int len, curlen = 0;
4207         char *news;

4209         if (*curses == NULL) {
4210                 /*
4211                  * Prime the pump, allocate just the
4212                  * space for prepend and return that.
4213                  */
4214                 len = strlen(prepend) + 1;
4215                 news = kmem_alloc(len, KM_SLEEP);
4216                 (void) strncpy(news, prepend, len);
4217         } else {
4218                 /*
4219                  * Allocate the space  for a new string
4220                  * +1 +1 is for the "/" and the NULL
4221                  * byte at the end of it all.
4222                  */
4223                 curlen = strlen(*curses);
4224                 len = curlen + strlen(prepend) + 1 + 1;
4225                 news = kmem_alloc(len, KM_SLEEP);
4226                 (void) strncpy(news, prepend, len);
4227                 (void) strcat(news, "/");
4228                 (void) strcat(news, *curses);
4229                 kmem_free(*curses, curlen + 1);
4230         }
4231         *curses = news;
4232 }

4234 /*
4235  * Returns the path name (starting from the fs root) for the given fname.
4236  * The caller is responsible for freeing.  Note that the path may be or
4237  * become invalid as a result of fn_move().
4238  */

4240 char *
4241 fn_path(nfs4_fname_t *fnp)
4242 {
4243         char *path;
4244         nfs4_fname_t *nextfnp;

4246         if (fnp == NULL)
4247                 return (NULL);

4249         path = NULL;
```



```
4251         /* walk up the tree constructing the pathname.  */

4253         fn_hold(fnp);                   /* adjust for later rele */
4254         do {
4255                 mutex_enter(&fnp->fn_lock);
4256                 /*
4257                  * Add fn_name in front of the current path
4258                  */
4259                 fn_path_realloc(&path, fnp->fn_name);
4260                 nextfnp = fnp->fn_parent;
4261                 if (nextfnp != NULL)
4262                         fn_hold(nextfnp);
4263                 mutex_exit(&fnp->fn_lock);
4264                 fn_rele(&fnp);
4265                 fnp = nextfnp;
4266         } while (fnp != NULL);

4268         return (path);
4269 }

4271 /*
4272  * Return a reference to the parent of the given fname, which the caller is
4273  * responsible for eventually releasing.
4274  */

4276 nfs4_fname_t *
4277 fn_parent(nfs4_fname_t *fnp)
4278 {
4279         nfs4_fname_t *parent;

4281         mutex_enter(&fnp->fn_lock);
4282         parent = fnp->fn_parent;
4283         if (parent != NULL)
4284                 fn_hold(parent);
4285         mutex_exit(&fnp->fn_lock);

4287         return (parent);
4288 }

4290 /*
4291  * Update fnp so that its parent is newparent and its name is newname.
4292  */

4294 void
4295 fn_move(nfs4_fname_t *fnp, nfs4_fname_t *newparent, char *newname)
4296 {
4297         nfs4_fname_t *parent, *tmpfnp;
4298         ssize_t newlen;
4299         nfs4_fname_t key;
4300         avl_index_t where;

4302         /*
4303          * This assert exists to catch the client trying to rename
4304          * a dir to be a child of itself.  This happened at a recent
4305          * bakeoff against a 3rd party (broken) server which allowed
4306          * the rename to succeed.  If it trips it means that:
4307          *      a) the code in nfs4rename that detects this case is broken
4308          *      b) the server is broken (since it allowed the bogus rename)
4309          *
4310          * For non-DEBUG kernels, prepare for a recursive mutex_enter
4311          * panic below from:  mutex_enter(&newparent->fn_lock);
4312          */
4313         ASSERT(fnp != newparent);

4315         /*
```

```
4316          * Remove fnp from its current parent, change its name, then add it
4317          * to newparent. It might happen that fnp was replaced by another
4318          * nfs4_fname_t with the same fn_name in parent->fn_children.
4319          * In such case, fnp->fn_parent is NULL and we skip the removal
4320          * of fnp from its current parent.
4321          */
4322         mutex_enter(&fnp->fn_lock);
4323         parent = fnp->fn_parent;
4324         if (parent != NULL) {
4325                 mutex_enter(&parent->fn_lock);
4326                 avl_remove(&parent->fn_children, fnp);
4327                 mutex_exit(&parent->fn_lock);
4328                 fn_rele(&fnp->fn_parent);
4329         }

4331         newlen = strlen(newname);
4332         if (newlen != fnp->fn_len) {
4333                 ASSERT(newlen < MAXNAMELEN);
4334                 kmem_free(fnp->fn_name, fnp->fn_len + 1);
4335                 fnp->fn_name = kmem_alloc(newlen + 1, KM_SLEEP);
4336                 fnp->fn_len = newlen;
4337         }
4338         (void) strcpy(fnp->fn_name, newname);

4340 again:
4341         mutex_enter(&newparent->fn_lock);
4342         key.fn_name = fnp->fn_name;
4343         tmpfnp = avl_find(&newparent->fn_children, &key, &where);
4344         if (tmpfnp != NULL) {
4345                 /*
4346                  * This could be due to a file that was unlinked while
4347                  * open, or perhaps the rnode is in the free list.  Remove
4348                  * it from newparent and let it go away on its own.  The
4349                  * contorted code is to deal with lock order issues and
4350                  * race conditions.
4351                  */
4352                 fn_hold(tmpfnp);
4353                 mutex_exit(&newparent->fn_lock);
4354                 mutex_enter(&tmpfnp->fn_lock);
4355                 if (tmpfnp->fn_parent == newparent) {
4356                         mutex_enter(&newparent->fn_lock);
4357                         avl_remove(&newparent->fn_children, tmpfnp);
4358                         mutex_exit(&newparent->fn_lock);
4359                         fn_rele(&tmpfnp->fn_parent);
4360                 }
4361                 mutex_exit(&tmpfnp->fn_lock);
4362                 fn_rele(&tmpfnp);
4363                 goto again;
4364         }
4365         fnp->fn_parent = newparent;
4366         fn_hold(newparent);
4367         avl_insert(&newparent->fn_children, fnp, where);
4368         mutex_exit(&newparent->fn_lock);
4369         mutex_exit(&fnp->fn_lock);
4370 }

4372 #ifdef DEBUG
4373 /*
4374  * Return non-zero if the type information makes sense for the given vnode.
4375  * Otherwise panic.
4376  */
4377 int
4378 nfs4_consistent_type(vnode_t *vp)
4379 {
4380         rnode4_t *rp = VTOR4(vp);
```



```
4382         if (nfs4_vtype_debug && vp->v_type != VNON &&
4383             rp->r_attr.va_type != VNON && vp->v_type != rp->r_attr.va_type) {
4384                 cmn_err(CE_PANIC, "vnode %p type mismatch; v_type=%d, "
4385                     "rnode attr type=%d", (void *)vp, vp->v_type,
4386                     rp->r_attr.va_type);
4387         }

4389         return (1);
4390 }
4391 #endif /* DEBUG */
```