****
29374 Fri May 8 18:04:02 2015
new/usr/src/uts/common/cpr/cpr_dump.c
patch lower-case-segops
****
_____***unchanged_portion_omitted_***

```
 661 /*
 662  * Count pages within each kernel segment; call cpr_sparse_seg_check()
 663  * to find out whether a sparsely filled segment needs special
 664  * treatment (e.g. kvseg).
 665  * Todo: A "segop_cpr" like segop_dump should be introduced, the cpr
 665  * Todo: A "SEGOP_CPR" like SEGOP_DUMP should be introduced, the cpr
 666  *       module shouldn't need to know segment details like if it is
 667  *       sparsely filled or not (makes kseg_table obsolete).
 668  */
 669 pgcnt_t
 670 cpr_count_seg_pages(int mapflag, bitfunc_t bitfunc)
 671 {
 672         struct seg *segp;
 673         pgcnt_t pages;
 674         ksegtbl_entry_t *ste;

 676         pages = 0;
 677         for (segp = AS_SEGFIRST(&kas); segp; segp = AS_SEGNEXT(&kas, segp)) {
 678                 if (ste = cpr_sparse_seg_check(segp)) {
 679                         pages += (ste->st_fcn)(mapflag, bitfunc, segp);
 680                 } else {
 681                         pages += cpr_count_pages(segp->s_base,
 682                             segp->s_size, mapflag, bitfunc, DBG_SHOWRANGE);
 683                 }
 684         }

 686         return (pages);
 687 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   55632 Fri May  8 18:04:02 2015
new/usr/src/uts/common/exec/elf/elf.c
patch lower-case-segops
**********************************************************
______unchanged_portion_omitted_

#ifdef _ELF32_COMPAT
extern size_t elf_datasz_max;
#else
size_t elf_datasz_max = 1 * 1024 * 1024;
#endif

/*
 * This function processes mappings that correspond to load objects to
 * examine their respective sections for elfcore(). It's called once with
 * v set to NULL to count the number of sections that we're going to need
 * and then again with v set to some allocated buffer that we fill in with
 * all the section data.
 */
static int
process_scns(core_content_t content, proc_t *p, cred_t *credp, vnode_t *vp,
    Shdr *v, int nv, rlim64_t rlimit, Off *doffsetp, int *nshdrsp)
{
        vnode_t *lastvp = NULL;
        struct seg *seg;
        int i, j;
        void *data = NULL;
        size_t datasz = 0;
        shstrtab_t shstrtab;
        struct as *as = p->p_as;
        int error = 0;

        if (v != NULL)
                shstrtab_init(&shstrtab);

        i = 1;
        for (seg = AS_SEGFIRST(as); seg != NULL; seg = AS_SEGNEXT(as, seg)) {
                uint_t prot;
                vnode_t *mvp;
                void *tmp = NULL;
                caddr_t saddr = seg->s_base;
                caddr_t naddr;
                caddr_t eaddr;
                size_t segsize;

                Ehdr ehdr;
                int nshdrs, shstrndx, nphdrs;
                caddr_t shbase;
                ssize_t shsize;
                char *shstrbase;
                ssize_t shstrsize;

                Shdr *shdr;
                const char *name;
                size_t sz;
                uintptr_t off;

                int ctf_ndx = 0;
                int symtab_ndx = 0;

                /*
                 * Since we're just looking for text segments of load
                 * objects, we only care about the protection bits; we don't
                 * care about the actual size of the segment so we use the
                 * reserved size. If the segment's size is zero, there's
```



```
                 * something fishy going on so we ignore this segment.
                 */
                if (seg->s_ops != &segvn_ops ||
                    segop_getvp(seg, seg->s_base, &mvp) != 0 ||
                    SEGOP_GETVP(seg, seg->s_base, &mvp) != 0 ||
                    mvp == lastvp || mvp == NULL || mvp->v_type != VREG ||
                    (segsize = pr_getsegsize(seg, 1)) == 0)
                        continue;

                eaddr = saddr + segsize;
                prot = pr_getprot(seg, 1, &tmp, &saddr, &naddr, eaddr);
                pr_getprot_done(&tmp);

                /*
                 * Skip this segment unless the protection bits look like
                 * what we'd expect for a text segment.
                 */
                if ((prot & (PROT_WRITE | PROT_EXEC)) != PROT_EXEC)
                        continue;

                if (getelfhead(mvp, credp, &ehdr, &nshdrs, &shstrndx,
                    &nphdrs) != 0 ||
                    getelfshdr(mvp, credp, &ehdr, nshdrs, shstrndx,
                    &shbase, &shsize, &shstrbase, &shstrsize) != 0)
                        continue;

                off = ehdr.e_shentsize;
                for (j = 1; j < nshdrs; j++, off += ehdr.e_shentsize) {
                        Shdr *symtab = NULL, *strtab;

                        shdr = (Shdr *)(shbase + off);

                        if (shdr->sh_name >= shstrsize)
                                continue;

                        name = shstrbase + shdr->sh_name;

                        if (strcmp(name, shstrtab_data[STR_CTF]) == 0) {
                                if ((content & CC_CONTENT_CTF) == 0 ||
                                    ctf_ndx != 0)
                                        continue;

                                if (shdr->sh_link > 0 &&
                                    shdr->sh_link < nshdrs) {
                                        symtab = (Shdr *)(shbase +
                                            shdr->sh_link * ehdr.e_shentsize);
                                }

                                if (v != NULL && i < nv - 1) {
                                        if (shdr->sh_size > datasz &&
                                            shdr->sh_size <= elf_datasz_max) {
                                                if (data != NULL)
                                                        kmem_free(data, datasz);

                                                datasz = shdr->sh_size;
                                                data = kmem_alloc(datasz,
                                                    KM_SLEEP);
                                        }

                                        v[i].sh_name = shstrtab_ndx(&shstrtab,
                                            STR_CTF);
                                        v[i].sh_addr = (Addr)(uintptr_t)saddr;
                                        v[i].sh_type = SHT_PROGBITS;
                                        v[i].sh_addralign = 4;
                                        *doffsetp = roundup(*doffsetp,
                                            v[i].sh_addralign);
```

```
1548                                    v[i].sh_offset = *doffsetp;
1549                                    v[i].sh_size = shdr->sh_size;
1550                                    if (symtab == NULL)  {
1551                                            v[i].sh_link = 0;
1552                                    } else if (symtab->sh_type ==
1553                                        SHT_SYMTAB &&
1554                                        symtab_ndx != 0) {
1555                                            v[i].sh_link =
1556                                                symtab_ndx;
1557                                    } else {
1558                                            v[i].sh_link = i + 1;
1559                                    }

1561                                    copy_scn(shdr, mvp, &v[i], vp,
1562                                        doffsetp, data, datasz, credp,
1563                                        rlimit);
1564                            }

1566                            ctf_ndx = i++;

1568                            /*
1569                             * We've already dumped the symtab.
1570                             */
1571                            if (symtab != NULL &&
1572                                symtab->sh_type == SHT_SYMTAB &&
1573                                symtab_ndx != 0)
1574                                    continue;

1576                    } else if (strcmp(name,
1577                        shstrtab_data[STR_SYMTAB]) == 0) {
1578                            if ((content & CC_CONTENT_SYMTAB) == 0 ||
1579                                symtab != 0)
1580                                    continue;

1582                            symtab = shdr;
1583                    }

1585                    if (symtab != NULL) {
1586                            if ((symtab->sh_type != SHT_DYNSYM &&
1587                                symtab->sh_type != SHT_SYMTAB) ||
1588                                symtab->sh_link == 0 ||
1589                                symtab->sh_link >= nshdrs)
1590                                    continue;

1592                            strtab = (Shdr *)(shbase +
1593                                symtab->sh_link * ehdr.e_shentsize);

1595                            if (strtab->sh_type != SHT_STRTAB)
1596                                    continue;

1598                            if (v != NULL && i < nv - 2) {
1599                                    sz = MAX(symtab->sh_size,
1600                                        strtab->sh_size);
1601                                    if (sz > datasz &&
1602                                        sz <= elf_datasz_max) {
1603                                            if (data != NULL)
1604                                                    kmem_free(data, datasz);

1606                                            datasz = sz;
1607                                            data = kmem_alloc(datasz,
1608                                                KM_SLEEP);
1609                                    }

1611                                    if (symtab->sh_type == SHT_DYNSYM) {
1612                                            v[i].sh_name = shstrtab_ndx(
1613                                                &shstrtab, STR_DYNSYM);
```



```
1614                                            v[i + 1].sh_name = shstrtab_ndx(
1615                                                &shstrtab, STR_DYNSTR);
1616                                    } else {
1617                                            v[i].sh_name = shstrtab_ndx(
1618                                                &shstrtab, STR_SYMTAB);
1619                                            v[i + 1].sh_name = shstrtab_ndx(
1620                                                &shstrtab, STR_STRTAB);
1621                                    }

1623                                    v[i].sh_type = symtab->sh_type;
1624                                    v[i].sh_addr = symtab->sh_addr;
1625                                    if (ehdr.e_type == ET_DYN ||
1626                                        v[i].sh_addr == 0)
1627                                            v[i].sh_addr +=
1628                                                (Addr)(uintptr_t)saddr;
1629                                    v[i].sh_addralign =
1630                                        symtab->sh_addralign;
1631                                    *doffsetp = roundup(*doffsetp,
1632                                        v[i].sh_addralign);
1633                                    v[i].sh_offset = *doffsetp;
1634                                    v[i].sh_size = symtab->sh_size;
1635                                    v[i].sh_link = i + 1;
1636                                    v[i].sh_entsize = symtab->sh_entsize;
1637                                    v[i].sh_info = symtab->sh_info;

1639                                    copy_scn(symtab, mvp, &v[i], vp,
1640                                        doffsetp, data, datasz, credp,
1641                                        rlimit);

1643                                    v[i + 1].sh_type = SHT_STRTAB;
1644                                    v[i + 1].sh_flags = SHF_STRINGS;
1645                                    v[i + 1].sh_addr = symtab->sh_addr;
1646                                    if (ehdr.e_type == ET_DYN ||
1647                                        v[i + 1].sh_addr == 0)
1648                                            v[i + 1].sh_addr +=
1649                                                (Addr)(uintptr_t)saddr;
1650                                    v[i + 1].sh_addralign =
1651                                        strtab->sh_addralign;
1652                                    *doffsetp = roundup(*doffsetp,
1653                                        v[i + 1].sh_addralign);
1654                                    v[i + 1].sh_offset = *doffsetp;
1655                                    v[i + 1].sh_size = strtab->sh_size;

1657                                    copy_scn(strtab, mvp, &v[i + 1], vp,
1658                                        doffsetp, data, datasz, credp,
1659                                        rlimit);
1660                            }

1662                            if (symtab->sh_type == SHT_SYMTAB)
1663                                    symtab_ndx = i;
1664                            i += 2;
1665                    }
1666            }

1668            kmem_free(shstrbase, shstrsize);
1669            kmem_free(shbase, shsize);

1671            lastvp = mvp;
1672    }

1674    if (v == NULL) {
1675            if (i == 1)
1676                    *nshdrsp = 0;
1677            else
1678                    *nshdrsp = i + 1;
1679            goto done;
```

```
1680         }

1682         if (i != nv - 1) {
1683                 cmn_err(CE_WARN, "elfcore: core dump failed for "
1684                     "process %d; address space is changing", p->p_pid);
1685                 error = EIO;
1686                 goto done;
1687         }

1689         v[i].sh_name = shstrtab_ndx(&shstrtab, STR_SHSTRTAB);
1690         v[i].sh_size = shstrtab_size(&shstrtab);
1691         v[i].sh_addralign = 1;
1692         *doffsetp = roundup(*doffsetp, v[i].sh_addralign);
1693         v[i].sh_offset = *doffsetp;
1694         v[i].sh_flags = SHF_STRINGS;
1695         v[i].sh_type = SHT_STRTAB;

1697         if (v[i].sh_size > datasz) {
1698                 if (data != NULL)
1699                         kmem_free(data, datasz);

1701                 datasz = v[i].sh_size;
1702                 data = kmem_alloc(datasz,
1703                     KM_SLEEP);
1704         }

1706         shstrtab_dump(&shstrtab, data);

1708         if ((error = core_write(vp, UIO_SYSSPACE, *doffsetp,
1709             data, v[i].sh_size, rlimit, credp)) != 0)
1710                 goto done;

1712         *doffsetp += v[i].sh_size;

1714 done:
1715         if (data != NULL)
1716                 kmem_free(data, datasz);

1718         return (error);
1719 }

1721 int
1722 elfcore(vnode_t *vp, proc_t *p, cred_t *credp, rlim64_t rlimit, int sig,
1723     core_content_t content)
1724 {
1725         offset_t poffset, soffset;
1726         Off doffset;
1727         int error, i, nphdrs, nshdrs;
1728         int overflow = 0;
1729         struct seg *seg;
1730         struct as *as = p->p_as;
1731         union {
1732                 Ehdr ehdr;
1733                 Phdr phdr[1];
1734                 Shdr shdr[1];
1735         } *bigwad;
1736         size_t bigsize;
1737         size_t phdrsz, shdrsz;
1738         Ehdr *ehdr;
1739         Phdr *v;
1740         caddr_t brkbase;
1741         size_t brksize;
1742         caddr_t stkbase;
1743         size_t stksize;
1744         int ntries = 0;
1745         klwp_t *lwp = ttolwp(curthread);
```



```
1747 top:
1748         /*
1749          * Make sure we have everything we need (registers, etc.).
1750          * All other lwps have already stopped and are in an orderly state.
1751          */
1752         ASSERT(p == ttoproc(curthread));
1753         prstop(0, 0);

1755         AS_LOCK_ENTER(as, &as->a_lock, RW_WRITER);
1756         nphdrs = prnsegs(as, 0) + 2;            /* two CORE note sections */

1758         /*
1759          * Count the number of section headers we're going to need.
1760          */
1761         nshdrs = 0;
1762         if (content & (CC_CONTENT_CTF | CC_CONTENT_SYMTAB)) {
1763                 (void) process_scns(content, p, credp, NULL, NULL, NULL, 0,
1764                     NULL, &nshdrs);
1765         }
1766         AS_LOCK_EXIT(as, &as->a_lock);

1768         ASSERT(nshdrs == 0 || nshdrs > 1);

1770         /*
1771          * The core file contents may required zero section headers, but if
1772          * we overflow the 16 bits allotted to the program header count in
1773          * the ELF header, we'll need that program header at index zero.
1774          */
1775         if (nshdrs == 0 && nphdrs >= PN_XNUM)
1776                 nshdrs = 1;

1778         phdrsz = nphdrs * sizeof (Phdr);
1779         shdrsz = nshdrs * sizeof (Shdr);

1781         bigsize = MAX(sizeof (*bigwad), MAX(phdrsz, shdrsz));
1782         bigwad = kmem_alloc(bigsize, KM_SLEEP);

1784         ehdr = &bigwad->ehdr;
1785         bzero(ehdr, sizeof (*ehdr));

1787         ehdr->e_ident[EI_MAG0] = ELFMAG0;
1788         ehdr->e_ident[EI_MAG1] = ELFMAG1;
1789         ehdr->e_ident[EI_MAG2] = ELFMAG2;
1790         ehdr->e_ident[EI_MAG3] = ELFMAG3;
1791         ehdr->e_ident[EI_CLASS] = ELFCLASS;
1792         ehdr->e_type = ET_CORE;

1794 #if !defined(_LP64) || defined(_ELF32_COMPAT)

1796 #if defined(__sparc)
1797         ehdr->e_ident[EI_DATA] = ELFDATA2MSB;
1798         ehdr->e_machine = EM_SPARC;
1799 #elif defined(__i386) || defined(__i386_COMPAT)
1800         ehdr->e_ident[EI_DATA] = ELFDATA2LSB;
1801         ehdr->e_machine = EM_386;
1802 #else
1803 #error "no recognized machine type is defined"
1804 #endif

1806 #else   /* !defined(_LP64) || defined(_ELF32_COMPAT) */

1808 #if defined(__sparc)
1809         ehdr->e_ident[EI_DATA] = ELFDATA2MSB;
1810         ehdr->e_machine = EM_SPARCV9;
1811 #elif defined(__amd64)
```

```
1812         ehdr->e_ident[EI_DATA] = ELFDATA2LSB;
1813         ehdr->e_machine = EM_AMD64;
1814 #else
1815 #error "no recognized 64-bit machine type is defined"
1816 #endif

1818 #endif  /* !defined(_LP64) || defined(_ELF32_COMPAT) */

1820         /*
1821          * If the count of program headers or section headers or the index
1822          * of the section string table can't fit in the mere 16 bits
1823          * shortsightedly allotted to them in the ELF header, we use the
1824          * extended formats and put the real values in the section header
1825          * as index 0.
1826          */
1827         ehdr->e_version = EV_CURRENT;
1828         ehdr->e_ehsize = sizeof (Ehdr);

1830         if (nphdrs >= PN_XNUM)
1831                 ehdr->e_phnum = PN_XNUM;
1832         else
1833                 ehdr->e_phnum = (unsigned short)nphdrs;

1835         ehdr->e_phoff = sizeof (Ehdr);
1836         ehdr->e_phentsize = sizeof (Phdr);

1838         if (nshdrs > 0) {
1839                 if (nshdrs >= SHN_LORESERVE)
1840                         ehdr->e_shnum = 0;
1841                 else
1842                         ehdr->e_shnum = (unsigned short)nshdrs;

1844                 if (nshdrs - 1 >= SHN_LORESERVE)
1845                         ehdr->e_shstrndx = SHN_XINDEX;
1846                 else
1847                         ehdr->e_shstrndx = (unsigned short)(nshdrs - 1);

1849                 ehdr->e_shoff = ehdr->e_phoff + ehdr->e_phentsize * nphdrs;
1850                 ehdr->e_shentsize = sizeof (Shdr);
1851         }

1853         if (error = core_write(vp, UIO_SYSSPACE, (offset_t)0, ehdr,
1854             sizeof (Ehdr), rlimit, credp))
1855                 goto done;

1857         poffset = sizeof (Ehdr);
1858         soffset = sizeof (Ehdr) + phdrsz;
1859         doffset = sizeof (Ehdr) + phdrsz + shdrsz;

1861         v = &bigwad->phdr[0];
1862         bzero(v, phdrsz);

1864         setup_old_note_header(&v[0], p);
1865         v[0].p_offset = doffset = roundup(doffset, sizeof (Word));
1866         doffset += v[0].p_filesz;

1868         setup_note_header(&v[1], p);
1869         v[1].p_offset = doffset = roundup(doffset, sizeof (Word));
1870         doffset += v[1].p_filesz;

1872         mutex_enter(&p->p_lock);

1874         brkbase = p->p_brkbase;
1875         brksize = p->p_brksize;

1877         stkbase = p->p_usrstack - p->p_stksize;
```



```
1878         stksize = p->p_stksize;

1880         mutex_exit(&p->p_lock);

1882         AS_LOCK_ENTER(as, &as->a_lock, RW_WRITER);
1883         i = 2;
1884         for (seg = AS_SEGFIRST(as); seg != NULL; seg = AS_SEGNEXT(as, seg)) {
1885                 caddr_t eaddr = seg->s_base + pr_getsegsize(seg, 0);
1886                 caddr_t saddr, naddr;
1887                 void *tmp = NULL;
1888                 extern struct seg_ops segspt_shmops;

1890                 for (saddr = seg->s_base; saddr < eaddr; saddr = naddr) {
1891                         uint_t prot;
1892                         size_t size;
1893                         int type;
1894                         vnode_t *mvp;

1896                         prot = pr_getprot(seg, 0, &tmp, &saddr, &naddr, eaddr);
1897                         prot &= PROT_READ | PROT_WRITE | PROT_EXEC;
1898                         if ((size = (size_t)(naddr - saddr)) == 0)
1899                                 continue;
1900                         if (i == nphdrs) {
1901                                 overflow++;
1902                                 continue;
1903                         }
1904                         v[i].p_type = PT_LOAD;
1905                         v[i].p_vaddr = (Addr)(uintptr_t)saddr;
1906                         v[i].p_memsz = size;
1907                         if (prot & PROT_READ)
1908                                 v[i].p_flags |= PF_R;
1909                         if (prot & PROT_WRITE)
1910                                 v[i].p_flags |= PF_W;
1911                         if (prot & PROT_EXEC)
1912                                 v[i].p_flags |= PF_X;

1914                         /*
1915                          * Figure out which mappings to include in the core.
1916                          */
1917                         type = segop_gettype(seg, saddr);
1917                         type = SEGOP_GETTYPE(seg, saddr);

1919                         if (saddr == stkbase && size == stksize) {
1920                                 if (!(content & CC_CONTENT_STACK))
1921                                         goto exclude;

1923                         } else if (saddr == brkbase && size == brksize) {
1924                                 if (!(content & CC_CONTENT_HEAP))
1925                                         goto exclude;

1927                         } else if (seg->s_ops == &segspt_shmops) {
1928                                 if (type & MAP_NORESERVE) {
1929                                         if (!(content & CC_CONTENT_DISM))
1930                                                 goto exclude;
1931                                 } else {
1932                                         if (!(content & CC_CONTENT_ISM))
1933                                                 goto exclude;
1934                                 }

1936                         } else if (seg->s_ops != &segvn_ops) {
1937                                 goto exclude;

1939                         } else if (type & MAP_SHARED) {
1940                                 if (shmgetid(p, saddr) != SHMID_NONE) {
1941                                         if (!(content & CC_CONTENT_SHM))
1942                                                 goto exclude;
```

```
1944                                } else if (segop_getvp(seg, seg->s_base,
1944                                } else if (SEGOP_GETVP(seg, seg->s_base,
1945                                    &mvp) != 0 || mvp == NULL ||
1946                                    mvp->v_type != VREG) {
1947                                        if (!(content & CC_CONTENT_SHANON))
1948                                                goto exclude;

1950                                } else {
1951                                        if (!(content & CC_CONTENT_SHFILE))
1952                                                goto exclude;
1953                                }

1955                            } else if (segop_getvp(seg, seg->s_base, &mvp) != 0 ||
1955                            } else if (SEGOP_GETVP(seg, seg->s_base, &mvp) != 0 ||
1956                                mvp == NULL || mvp->v_type != VREG) {
1957                                    if (!(content & CC_CONTENT_ANON))
1958                                            goto exclude;

1960                            } else if (prot == (PROT_READ | PROT_EXEC)) {
1961                                    if (!(content & CC_CONTENT_TEXT))
1962                                            goto exclude;

1964                            } else if (prot == PROT_READ) {
1965                                    if (!(content & CC_CONTENT_RODATA))
1966                                            goto exclude;

1968                            } else {
1969                                    if (!(content & CC_CONTENT_DATA))
1970                                            goto exclude;
1971                            }

1973                            doffset = roundup(doffset, sizeof (Word));
1974                            v[i].p_offset = doffset;
1975                            v[i].p_filesz = size;
1976                            doffset += size;
1977 exclude:
1978                            i++;
1979                    }
1980                    ASSERT(tmp == NULL);
1981            }
1982            AS_LOCK_EXIT(as, &as->a_lock);

1984            if (overflow || i != nphdrs) {
1985                    if (ntries++ == 0) {
1986                            kmem_free(bigwad, bigsize);
1987                            overflow = 0;
1988                            goto top;
1989                    }
1990                    cmn_err(CE_WARN, "elfcore: core dump failed for "
1991                        "process %d; address space is changing", p->p_pid);
1992                    error = EIO;
1993                    goto done;
1994            }

1996            if ((error = core_write(vp, UIO_SYSSPACE, poffset,
1997                v, phdrsz, rlimit, credp)) != 0)
1998                    goto done;

2000            if ((error = write_old_elfnotes(p, sig, vp, v[0].p_offset, rlimit,
2001                credp)) != 0)
2002                    goto done;

2004            if ((error = write_elfnotes(p, sig, vp, v[1].p_offset, rlimit,
2005                credp, content)) != 0)
2006                    goto done;
```



```
2008            for (i = 2; i < nphdrs; i++) {
2009                    prkillinfo_t killinfo;
2010                    sigqueue_t *sq;
2011                    int sig, j;

2013                    if (v[i].p_filesz == 0)
2014                            continue;

2016                    /*
2017                     * If dumping out this segment fails, rather than failing
2018                     * the core dump entirely, we reset the size of the mapping
2019                     * to zero to indicate that the data is absent from the core
2020                     * file and or in the PF_SUNW_FAILURE flag to differentiate
2021                     * this from mappings that were excluded due to the core file
2022                     * content settings.
2023                     */
2024                    if ((error = core_seg(p, vp, v[i].p_offset,
2025                        (caddr_t)(uintptr_t)v[i].p_vaddr, v[i].p_filesz,
2026                        rlimit, credp)) == 0) {
2027                            continue;
2028                    }

2030                    if ((sig = lwp->lwp_cursig) == 0) {
2031                            /*
2032                             * We failed due to something other than a signal.
2033                             * Since the space reserved for the segment is now
2034                             * unused, we stash the errno in the first four
2035                             * bytes. This undocumented interface will let us
2036                             * understand the nature of the failure.
2037                             */
2038                            (void) core_write(vp, UIO_SYSSPACE, v[i].p_offset,
2039                                &error, sizeof (error), rlimit, credp);

2041                            v[i].p_filesz = 0;
2042                            v[i].p_flags |= PF_SUNW_FAILURE;
2043                            if ((error = core_write(vp, UIO_SYSSPACE,
2044                                poffset + sizeof (v[i]) * i, &v[i], sizeof (v[i]),
2045                                rlimit, credp)) != 0)
2046                                    goto done;

2048                            continue;
2049                    }

2051                    /*
2052                     * We took a signal.  We want to abort the dump entirely, but
2053                     * we also want to indicate what failed and why.  We therefore
2054                     * use the space reserved for the first failing segment to
2055                     * write our error (which, for purposes of compatability with
2056                     * older core dump readers, we set to EINTR) followed by any
2057                     * siginfo associated with the signal.
2058                     */
2059                    bzero(&killinfo, sizeof (killinfo));
2060                    killinfo.prk_error = EINTR;

2062                    sq = sig == SIGKILL ? curproc->p_killsqp : lwp->lwp_curinfo;

2064                    if (sq != NULL) {
2065                            bcopy(&sq->sq_info, &killinfo.prk_info,
2066                                sizeof (sq->sq_info));
2067                    } else {
2068                            killinfo.prk_info.si_signo = lwp->lwp_cursig;
2069                            killinfo.prk_info.si_code = SI_NOINFO;
2070                    }

2072 #if (defined(_SYSCALL32_IMPL) || defined(_LP64))
```

```
2073                 /*
2074                  * If this is a 32-bit process, we need to translate from the
2075                  * native siginfo to the 32-bit variant.  (Core readers must
2076                  * always have the same data model as their target or must
2077                  * be aware of -- and compensate for -- data model differences.)
2078                  */
2079                 if (curproc->p_model == DATAMODEL_ILP32) {
2080                         siginfo32_t si32;

2082                         siginfo_kto32((k_siginfo_t *)&killinfo.prk_info, &si32);
2083                         bcopy(&si32, &killinfo.prk_info, sizeof (si32));
2084                 }
2085 #endif

2087                 (void) core_write(vp, UIO_SYSSPACE, v[i].p_offset,
2088                     &killinfo, sizeof (killinfo), rlimit, credp);

2090                 /*
2091                  * For the segment on which we took the signal, indicate that
2092                  * its data now refers to a siginfo.
2093                  */
2094                 v[i].p_filesz = 0;
2095                 v[i].p_flags |= PF_SUNW_FAILURE | PF_SUNW_KILLED |
2096                     PF_SUNW_SIGINFO;

2098                 /*
2099                  * And for every other segment, indicate that its absence
2100                  * is due to a signal.
2101                  */
2102                 for (j = i + 1; j < nphdrs; j++) {
2103                         v[j].p_filesz = 0;
2104                         v[j].p_flags |= PF_SUNW_FAILURE | PF_SUNW_KILLED;
2105                 }

2107                 /*
2108                  * Finally, write out our modified program headers.
2109                  */
2110                 if ((error = core_write(vp, UIO_SYSSPACE,
2111                     poffset + sizeof (v[i]) * i, &v[i],
2112                     sizeof (v[i]) * (nphdrs - i), rlimit, credp)) != 0)
2113                         goto done;

2115                 break;
2116         }

2118         if (nshdrs > 0) {
2119                 bzero(&bigwad->shdr[0], shdrsz);

2121                 if (nshdrs >= SHN_LORESERVE)
2122                         bigwad->shdr[0].sh_size = nshdrs;

2124                 if (nshdrs - 1 >= SHN_LORESERVE)
2125                         bigwad->shdr[0].sh_link = nshdrs - 1;

2127                 if (nphdrs >= PN_XNUM)
2128                         bigwad->shdr[0].sh_info = nphdrs;

2130                 if (nshdrs > 1) {
2131                         AS_LOCK_ENTER(as, &as->a_lock, RW_WRITER);
2132                         if ((error = process_scns(content, p, credp, vp,
2133                             &bigwad->shdr[0], nshdrs, rlimit, &doffset,
2134                             NULL)) != 0) {
2135                                 AS_LOCK_EXIT(as, &as->a_lock);
2136                                 goto done;
2137                         }
2138                         AS_LOCK_EXIT(as, &as->a_lock);
```



```
2139                 }

2141                 if ((error = core_write(vp, UIO_SYSSPACE, soffset,
2142                     &bigwad->shdr[0], shdrsz, rlimit, credp)) != 0)
2143                         goto done;
2144         }

2146 done:
2147         kmem_free(bigwad, bigsize);
2148         return (error);
2149 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
  171811 Fri May  8 18:04:02 2015
new/usr/src/uts/common/fs/nfs/nfs3_vnops.c
patch lower-case-segops
**********************************************************
______unchanged_portion_omitted_

5543 /*
5544  * Setup and add an address space callback to do the work of the delmap call.
5545  * The callback will (and must be) deleted in the actual callback function.
5546  *
5547  * This is done in order to take care of the problem that we have with holding
5548  * the address space's a_lock for a long period of time (e.g. if the NFS server
5549  * is down).  Callbacks will be executed in the address space code while the
5550  * a_lock is not held.  Holding the address space's a_lock causes things such
5551  * as ps and fork to hang because they are trying to acquire this lock as well.
5552  */
5553 /* ARGSUSED */
5554 static int
5555 nfs3_delmap(vnode_t *vp, offset_t off, struct as *as, caddr_t addr,
5556         size_t len, uint_t prot, uint_t maxprot, uint_t flags,
5557         cred_t *cr, caller_context_t *ct)
5558 {
5559         int                     caller_found;
5560         int                     error;
5561         rnode_t                 *rp;
5562         nfs_delmap_args_t       *dmapp;
5563         nfs_delmapcall_t        *delmap_call;

5565         if (vp->v_flag & VNOMAP)
5566                 return (ENOSYS);
5567         /*
5568          * A process may not change zones if it has NFS pages mmap'ed
5569          * in, so we can't legitimately get here from the wrong zone.
5570          */
5571         ASSERT(nfs_zone() == VTOMI(vp)->mi_zone);

5573         rp = VTOR(vp);

5575         /*
5576          * The way that the address space of this process deletes its mapping
5577          * of this file is via the following call chains:
5578          * - as_free()->segop_unmap()/segvn_unmap()->VOP_DELMAP()/nfs3_delmap()
5579          * - as_unmap()->segop_unmap()/segvn_unmap()->VOP_DELMAP()/nfs3_delmap()
5578          * - as_free()->SEGOP_UNMAP()/segvn_unmap()->VOP_DELMAP()/nfs3_delmap()
5579          * - as_unmap()->SEGOP_UNMAP()/segvn_unmap()->VOP_DELMAP()/nfs3_delmap()
5580          *
5581          * With the use of address space callbacks we are allowed to drop the
5582          * address space lock, a_lock, while executing the NFS operations that
5583          * need to go over the wire.  Returning EAGAIN to the caller of this
5584          * function is what drives the execution of the callback that we add
5585          * below.  The callback will be executed by the address space code
5586          * after dropping the a_lock.  When the callback is finished, since
5587          * we dropped the a_lock, it must be re-acquired and segvn_unmap()
5588          * is called again on the same segment to finish the rest of the work
5589          * that needs to happen during unmapping.
5590          *
5591          * This action of calling back into the segment driver causes
5592          * nfs3_delmap() to get called again, but since the callback was
5593          * already executed at this point, it already did the work and there
5594          * is nothing left for us to do.
5595          *
5596          * To Summarize:
5597          * - The first time nfs3_delmap is called by the current thread is when
5598          * we add the caller associated with this delmap to the delmap caller
5599          * list, add the callback, and return EAGAIN.
```



```
5600          * - The second time in this call chain when nfs3_delmap is called we
5601          * will find this caller in the delmap caller list and realize there
5602          * is no more work to do thus removing this caller from the list and
5603          * returning the error that was set in the callback execution.
5604          */
5605         caller_found = nfs_find_and_delete_delmapcall(rp, &error);
5606         if (caller_found) {
5607                 /*
5608                  * 'error' is from the actual delmap operations.  To avoid
5609                  * hangs, we need to handle the return of EAGAIN differently
5610                  * since this is what drives the callback execution.
5611                  * In this case, we don't want to return EAGAIN and do the
5612                  * callback execution because there are none to execute.
5613                  */
5614                 if (error == EAGAIN)
5615                         return (0);
5616                 else
5617                         return (error);
5618         }

5620         /* current caller was not in the list */
5621         delmap_call = nfs_init_delmapcall();

5623         mutex_enter(&rp->r_statelock);
5624         list_insert_tail(&rp->r_indelmap, delmap_call);
5625         mutex_exit(&rp->r_statelock);

5627         dmapp = kmem_alloc(sizeof (nfs_delmap_args_t), KM_SLEEP);

5629         dmapp->vp = vp;
5630         dmapp->off = off;
5631         dmapp->addr = addr;
5632         dmapp->len = len;
5633         dmapp->prot = prot;
5634         dmapp->maxprot = maxprot;
5635         dmapp->flags = flags;
5636         dmapp->cr = cr;
5637         dmapp->caller = delmap_call;

5639         error = as_add_callback(as, nfs3_delmap_callback, dmapp,
5640             AS_UNMAP_EVENT, addr, len, KM_SLEEP);

5642         return (error ? error : EAGAIN);
5643 }
______unchanged_portion_omitted_
```

****
429784 Fri May 8 18:04:03 2015
new/usr/src/uts/common/fs/nfs/nfs4_vnops.c
patch lower-case-segops
****
_____unchanged_portion_omitted_

```
11024 /*
11025  * Setup and add an address space callback to do the work of the delmap call.
11026  * The callback will (and must be) deleted in the actual callback function.
11027  *
11028  * This is done in order to take care of the problem that we have with holding
11029  * the address space's a_lock for a long period of time (e.g. if the NFS server
11030  * is down).  Callbacks will be executed in the address space code while the
11031  * a_lock is not held.  Holding the address space's a_lock causes things such
11032  * as ps and fork to hang because they are trying to acquire this lock as well.
11033  */
11034 /* ARGSUSED */
11035 static int
11036 nfs4_delmap(vnode_t *vp, offset_t off, struct as *as, caddr_t addr,
11037     size_t len, uint_t prot, uint_t maxprot, uint_t flags, cred_t *cr,
11038     caller_context_t *ct)
11039 {
11040         int                     caller_found;
11041         int                     error;
11042         rnode4_t                *rp;
11043         nfs4_delmap_args_t      *dmapp;
11044         nfs4_delmapcall_t       *delmap_call;

11046         if (vp->v_flag & VNOMAP)
11047                 return (ENOSYS);

11049         /*
11050          * A process may not change zones if it has NFS pages mmap'ed
11051          * in, so we can't legitimately get here from the wrong zone.
11052          */
11053         ASSERT(nfs_zone() == VTOMI4(vp)->mi_zone);

11055         rp = VTOR4(vp);

11057         /*
11058          * The way that the address space of this process deletes its mapping
11059          * of this file is via the following call chains:
11060          * - as_free()->segop_unmap()/segvn_unmap()->VOP_DELMAP()/nfs4_delmap()
11061          * - as_unmap()->segop_unmap()/segvn_unmap()->VOP_DELMAP()/nfs4_delmap()
11060          * - as_free()->SEGOP_UNMAP()/segvn_unmap()->VOP_DELMAP()/nfs4_delmap()
11061          * - as_unmap()->SEGOP_UNMAP()/segvn_unmap()->VOP_DELMAP()/nfs4_delmap()
11062          *
11063          * With the use of address space callbacks we are allowed to drop the
11064          * address space lock, a_lock, while executing the NFS operations that
11065          * need to go over the wire.  Returning EAGAIN to the caller of this
11066          * function is what drives the execution of the callback that we add
11067          * below.  The callback will be executed by the address space code
11068          * after dropping the a_lock.  When the callback is finished, since
11069          * we dropped the a_lock, it must be re-acquired and segvn_unmap()
11070          * is called again on the same segment to finish the rest of the work
11071          * that needs to happen during unmapping.
11072          *
11073          * This action of calling back into the segment driver causes
11074          * nfs4_delmap() to get called again, but since the callback was
11075          * already executed at this point, it already did the work and there
11076          * is nothing left for us to do.
11077          *
11078          * To Summarize:
11079          * - The first time nfs4_delmap is called by the current thread is when
11080          * we add the caller associated with this delmap to the delmap caller
```



```
11081          * list, add the callback, and return EAGAIN.
11082          * - The second time in this call chain when nfs4_delmap is called we
11083          * will find this caller in the delmap caller list and realize there
11084          * is no more work to do thus removing this caller from the list and
11085          * returning the error that was set in the callback execution.
11086          */
11087         caller_found = nfs4_find_and_delete_delmapcall(rp, &error);
11088         if (caller_found) {
11089                 /*
11090                  * 'error' is from the actual delmap operations.  To avoid
11091                  * hangs, we need to handle the return of EAGAIN differently
11092                  * since this is what drives the callback execution.
11093                  * In this case, we don't want to return EAGAIN and do the
11094                  * callback execution because there are none to execute.
11095                  */
11096                 if (error == EAGAIN)
11097                         return (0);
11098                 else
11099                         return (error);
11100         }

11102         /* current caller was not in the list */
11103         delmap_call = nfs4_init_delmapcall();

11105         mutex_enter(&rp->r_statelock);
11106         list_insert_tail(&rp->r_indelmap, delmap_call);
11107         mutex_exit(&rp->r_statelock);

11109         dmapp = kmem_alloc(sizeof (nfs4_delmap_args_t), KM_SLEEP);

11111         dmapp->vp = vp;
11112         dmapp->off = off;
11113         dmapp->addr = addr;
11114         dmapp->len = len;
11115         dmapp->prot = prot;
11116         dmapp->maxprot = maxprot;
11117         dmapp->flags = flags;
11118         dmapp->cr = cr;
11119         dmapp->caller = delmap_call;

11121         error = as_add_callback(as, nfs4_delmap_callback, dmapp,
11122             AS_UNMAP_EVENT, addr, len, KM_SLEEP);

11124         return (error ? error : EAGAIN);
11125 }
```

_____unchanged_portion_omitted_

```
**********************************************************
  130899 Fri May  8 18:04:04 2015
new/usr/src/uts/common/fs/nfs/nfs_vnops.c
patch lower-case-segops
**********************************************************
______unchanged_portion_omitted_

4640 /*
4641  * Setup and add an address space callback to do the work of the delmap call.
4642  * The callback will (and must be) deleted in the actual callback function.
4643  *
4644  * This is done in order to take care of the problem that we have with holding
4645  * the address space's a_lock for a long period of time (e.g. if the NFS server
4646  * is down).  Callbacks will be executed in the address space code while the
4647  * a_lock is not held.  Holding the address space's a_lock causes things such
4648  * as ps and fork to hang because they are trying to acquire this lock as well.
4649  */
4650 /* ARGSUSED */
4651 static int
4652 nfs_delmap(vnode_t *vp, offset_t off, struct as *as, caddr_t addr,
4653         size_t len, uint_t prot, uint_t maxprot, uint_t flags, cred_t *cr,
4654         caller_context_t *ct)
4655 {
4656         int                     caller_found;
4657         int                     error;
4658         rnode_t                 *rp;
4659         nfs_delmap_args_t       *dmapp;
4660         nfs_delmapcall_t        *delmap_call;

4662         if (vp->v_flag & VNOMAP)
4663                 return (ENOSYS);
4664         /*
4665          * A process may not change zones if it has NFS pages mmap'ed
4666          * in, so we can't legitimately get here from the wrong zone.
4667          */
4668         ASSERT(nfs_zone() == VTOMI(vp)->mi_zone);

4670         rp = VTOR(vp);

4672         /*
4673          * The way that the address space of this process deletes its mapping
4674          * of this file is via the following call chains:
4675          * - as_free()->segop_unmap()/segvn_unmap()->VOP_DELMAP()/nfs_delmap()
4676          * - as_unmap()->segop_unmap()/segvn_unmap()->VOP_DELMAP()/nfs_delmap()
4675          * - as_free()->SEGOP_UNMAP()/segvn_unmap()->VOP_DELMAP()/nfs_delmap()
4676          * - as_unmap()->SEGOP_UNMAP()/segvn_unmap()->VOP_DELMAP()/nfs_delmap()
4677          *
4678          * With the use of address space callbacks we are allowed to drop the
4679          * address space lock, a_lock, while executing the NFS operations that
4680          * need to go over the wire.  Returning EAGAIN to the caller of this
4681          * function is what drives the execution of the callback that we add
4682          * below.  The callback will be executed by the address space code
4683          * after dropping the a_lock.  When the callback is finished, since
4684          * we dropped the a_lock, it must be re-acquired and segvn_unmap()
4685          * is called again on the same segment to finish the rest of the work
4686          * that needs to happen during unmapping.
4687          *
4688          * This action of calling back into the segment driver causes
4689          * nfs_delmap() to get called again, but since the callback was
4690          * already executed at this point, it already did the work and there
4691          * is nothing left for us to do.
4692          *
4693          * To Summarize:
4694          * - The first time nfs_delmap is called by the current thread is when
4695          * we add the caller associated with this delmap to the delmap caller
4696          * list, add the callback, and return EAGAIN.
```



```
4697          * - The second time in this call chain when nfs_delmap is called we
4698          * will find this caller in the delmap caller list and realize there
4699          * is no more work to do thus removing this caller from the list and
4700          * returning the error that was set in the callback execution.
4701          */
4702         caller_found = nfs_find_and_delete_delmapcall(rp, &error);
4703         if (caller_found) {
4704                 /*
4705                  * 'error' is from the actual delmap operations.  To avoid
4706                  * hangs, we need to handle the return of EAGAIN differently
4707                  * since this is what drives the callback execution.
4708                  * In this case, we don't want to return EAGAIN and do the
4709                  * callback execution because there are none to execute.
4710                  */
4711                 if (error == EAGAIN)
4712                         return (0);
4713                 else
4714                         return (error);
4715         }

4717         /* current caller was not in the list */
4718         delmap_call = nfs_init_delmapcall();

4720         mutex_enter(&rp->r_statelock);
4721         list_insert_tail(&rp->r_indelmap, delmap_call);
4722         mutex_exit(&rp->r_statelock);

4724         dmapp = kmem_alloc(sizeof (nfs_delmap_args_t), KM_SLEEP);

4726         dmapp->vp = vp;
4727         dmapp->off = off;
4728         dmapp->addr = addr;
4729         dmapp->len = len;
4730         dmapp->prot = prot;
4731         dmapp->maxprot = maxprot;
4732         dmapp->flags = flags;
4733         dmapp->cr = cr;
4734         dmapp->caller = delmap_call;

4736         error = as_add_callback(as, nfs_delmap_callback, dmapp,
4737             AS_UNMAP_EVENT, addr, len, KM_SLEEP);

4739         return (error ? error : EAGAIN);
4740 }
______unchanged_portion_omitted_
```

```
**********************************************************
   93906 Fri May  8 18:04:04 2015
new/usr/src/uts/common/fs/proc/prioctl.c
patch lower-case-segops
**********************************************************
______unchanged_portion_omitted_

3117 /*
3118  * Common code for PIOCOPENM
3119  * Returns with the process unlocked.
3120  */
3121 static int
3122 propenm(prnode_t *pnp, caddr_t cmaddr, caddr_t va, int *rvalp, cred_t *cr)
3123 {
3124         proc_t *p = pnp->pr_common->prc_proc;
3125         struct as *as = p->p_as;
3126         int error = 0;
3127         struct seg *seg;
3128         struct vnode *xvp;
3129         int n;

3131         /*
3132          * By fiat, a system process has no address space.
3133          */
3134         if ((p->p_flag & SSYS) || as == &kas) {
3135                 error = EINVAL;
3136         } else if (cmaddr) {
3137                 /*
3138                  * We drop p_lock before grabbing the address
3139                  * space lock in order to avoid a deadlock with
3140                  * the clock thread.  The process will not
3141                  * disappear and its address space will not
3142                  * change because it is marked P_PR_LOCK.
3143                  */
3144                 mutex_exit(&p->p_lock);
3145                 AS_LOCK_ENTER(as, &as->a_lock, RW_READER);
3146                 seg = as_segat(as, va);
3147                 if (seg != NULL &&
3148                     seg->s_ops == &segvn_ops &&
3149                     segop_getvp(seg, va, &xvp) == 0 &&
3149                     SEGOP_GETVP(seg, va, &xvp) == 0 &&
3150                     xvp != NULL &&
3151                     xvp->v_type == VREG) {
3152                         VN_HOLD(xvp);
3153                 } else {
3154                         error = EINVAL;
3155                 }
3156                 AS_LOCK_EXIT(as, &as->a_lock);
3157                 mutex_enter(&p->p_lock);
3158         } else if ((xvp = p->p_exec) == NULL) {
3159                 error = EINVAL;
3160         } else {
3161                 VN_HOLD(xvp);
3162         }

3164         prunlock(pnp);

3166         if (error == 0) {
3167                 if ((error = VOP_ACCESS(xvp, VREAD, 0, cr, NULL)) == 0)
3168                         error = fassign(&xvp, FREAD, &n);
3169                 if (error) {
3170                         VN_RELE(xvp);
3171                 } else {
3172                         *rvalp = n;
3173                 }
3174         }
```



```
3176         return (error);
3177 }
______unchanged_portion_omitted_

3511 /*
3512  * Return an array of structures with memory map information.
3513  * We allocate here; the caller must deallocate.
3514  * The caller is also responsible to append the zero-filled entry
3515  * that terminates the PIOCMAP output buffer.
3516  */
3517 static int
3518 oprgetmap(proc_t *p, list_t *iolhead)
3519 {
3520         struct as *as = p->p_as;
3521         prmap_t *mp;
3522         struct seg *seg;
3523         struct seg *brkseg, *stkseg;
3524         uint_t prot;

3526         ASSERT(as != &kas && AS_WRITE_HELD(as, &as->a_lock));

3528         /*
3529          * Request an initial buffer size that doesn't waste memory
3530          * if the address space has only a small number of segments.
3531          */
3532         pr_iol_initlist(iolhead, sizeof (*mp), avl_numnodes(&as->a_segtree));

3534         if ((seg = AS_SEGFIRST(as)) == NULL)
3535                 return (0);

3537         brkseg = break_seg(p);
3538         stkseg = as_segat(as, prgetstackbase(p));

3540         do {
3541                 caddr_t eaddr = seg->s_base + pr_getsegsize(seg, 0);
3542                 caddr_t saddr, naddr;
3543                 void *tmp = NULL;

3545                 for (saddr = seg->s_base; saddr < eaddr; saddr = naddr) {
3546                         prot = pr_getprot(seg, 0, &tmp, &saddr, &naddr, eaddr);
3547                         if (saddr == naddr)
3548                                 continue;

3550                         mp = pr_iol_newbuf(iolhead, sizeof (*mp));

3552                         mp->pr_vaddr = saddr;
3553                         mp->pr_size = naddr - saddr;
3554                         mp->pr_off = segop_getoffset(seg, saddr);
3554                         mp->pr_off = SEGOP_GETOFFSET(seg, saddr);
3555                         mp->pr_mflags = 0;
3556                         if (prot & PROT_READ)
3557                                 mp->pr_mflags |= MA_READ;
3558                         if (prot & PROT_WRITE)
3559                                 mp->pr_mflags |= MA_WRITE;
3560                         if (prot & PROT_EXEC)
3561                                 mp->pr_mflags |= MA_EXEC;
3562                         if (segop_gettype(seg, saddr) & MAP_SHARED)
3562                         if (SEGOP_GETTYPE(seg, saddr) & MAP_SHARED)
3563                                 mp->pr_mflags |= MA_SHARED;
3564                         if (seg == brkseg)
3565                                 mp->pr_mflags |= MA_BREAK;
3566                         else if (seg == stkseg)
3567                                 mp->pr_mflags |= MA_STACK;
3568                         mp->pr_pagesize = PAGESIZE;
3569                 }
```

```
3570                 ASSERT(tmp == NULL);
3571         } while ((seg = AS_SEGNEXT(as, seg)) != NULL);

3573         return (0);
3574 }

3576 #ifdef _SYSCALL32_IMPL
3577 static int
3578 oprgetmap32(proc_t *p, list_t *iolhead)
3579 {
3580         struct as *as = p->p_as;
3581         ioc_prmap32_t *mp;
3582         struct seg *seg;
3583         struct seg *brkseg, *stkseg;
3584         uint_t prot;

3586         ASSERT(as != &kas && AS_WRITE_HELD(as, &as->a_lock));

3588         /*
3589          * Request an initial buffer size that doesn't waste memory
3590          * if the address space has only a small number of segments.
3591          */
3592         pr_iol_initlist(iolhead, sizeof (*mp), avl_numnodes(&as->a_segtree));

3594         if ((seg = AS_SEGFIRST(as)) == NULL)
3595                 return (0);

3597         brkseg = break_seg(p);
3598         stkseg = as_segat(as, prgetstackbase(p));

3600         do {
3601                 caddr_t eaddr = seg->s_base + pr_getsegsize(seg, 0);
3602                 caddr_t saddr, naddr;
3603                 void *tmp = NULL;

3605                 for (saddr = seg->s_base; saddr < eaddr; saddr = naddr) {
3606                         prot = pr_getprot(seg, 0, &tmp, &saddr, &naddr, eaddr);
3607                         if (saddr == naddr)
3608                                 continue;

3610                         mp = pr_iol_newbuf(iolhead, sizeof (*mp));

3612                         mp->pr_vaddr = (caddr32_t)(uintptr_t)saddr;
3613                         mp->pr_size = (size32_t)(naddr - saddr);
3614                         mp->pr_off = (off32_t)segop_getoffset(seg, saddr);
3614                         mp->pr_off = (off32_t)SEGOP_GETOFFSET(seg, saddr);
3615                         mp->pr_mflags = 0;
3616                         if (prot & PROT_READ)
3617                                 mp->pr_mflags |= MA_READ;
3618                         if (prot & PROT_WRITE)
3619                                 mp->pr_mflags |= MA_WRITE;
3620                         if (prot & PROT_EXEC)
3621                                 mp->pr_mflags |= MA_EXEC;
3622                         if (segop_gettype(seg, saddr) & MAP_SHARED)
3622                         if (SEGOP_GETTYPE(seg, saddr) & MAP_SHARED)
3623                                 mp->pr_mflags |= MA_SHARED;
3624                         if (seg == brkseg)
3625                                 mp->pr_mflags |= MA_BREAK;
3626                         else if (seg == stkseg)
3627                                 mp->pr_mflags |= MA_STACK;
3628                         mp->pr_pagesize = PAGESIZE;
3629                 }
3630                 ASSERT(tmp == NULL);
3631         } while ((seg = AS_SEGNEXT(as, seg)) != NULL);

3633         return (0);
```



```
3634 }
_____unchanged_portion_omitted_
3698 #endif  /* _SYSCALL32_IMPL */

3700 /*
3701  * Read old /proc page data information.
3702  */
3703 int
3704 oprpdread(struct as *as, uint_t hatid, struct uio *uiop)
3705 {
3706         caddr_t buf;
3707         size_t size;
3708         prpageheader_t *php;
3709         prasmap_t *pmp;
3710         struct seg *seg;
3711         int error;

3713 again:
3714         AS_LOCK_ENTER(as, &as->a_lock, RW_WRITER);

3716         if ((seg = AS_SEGFIRST(as)) == NULL) {
3717                 AS_LOCK_EXIT(as, &as->a_lock);
3718                 return (0);
3719         }
3720         size = oprpdsize(as);
3721         if (uiop->uio_resid < size) {
3722                 AS_LOCK_EXIT(as, &as->a_lock);
3723                 return (E2BIG);
3724         }

3726         buf = kmem_zalloc(size, KM_SLEEP);
3727         php = (prpageheader_t *)buf;
3728         pmp = (prasmap_t *)(buf + sizeof (prpageheader_t));

3730         hrt2ts(gethrtime(), &php->pr_tstamp);
3731         php->pr_nmap = 0;
3732         php->pr_npage = 0;
3733         do {
3734                 caddr_t eaddr = seg->s_base + pr_getsegsize(seg, 0);
3735                 caddr_t saddr, naddr;
3736                 void *tmp = NULL;

3738                 for (saddr = seg->s_base; saddr < eaddr; saddr = naddr) {
3739                         size_t len;
3740                         size_t npage;
3741                         uint_t prot;
3742                         uintptr_t next;

3744                         prot = pr_getprot(seg, 0, &tmp, &saddr, &naddr, eaddr);
3745                         if ((len = naddr - saddr) == 0)
3746                                 continue;
3747                         npage = len / PAGESIZE;
3748                         next = (uintptr_t)(pmp + 1) + roundlong(npage);
3749                         /*
3750                          * It's possible that the address space can change
3751                          * subtlely even though we're holding as->a_lock
3752                          * due to the nondeterminism of page_exists() in
3753                          * the presence of asychronously flushed pages or
3754                          * mapped files whose sizes are changing.
3755                          * page_exists() may be called indirectly from
3756                          * pr_getprot() by a segop_incore() routine.
3756                          * pr_getprot() by a SEGOP_INCORE() routine.
3757                          * If this happens we need to make sure we don't
3758                          * overrun the buffer whose size we computed based
3759                          * on the initial iteration through the segments.
3760                          * Once we've detected an overflow, we need to clean
```

```
3761                          * up the temporary memory allocated in pr_getprot()
3762                          * and retry. If there's a pending signal, we return
3763                          * EINTR so that this thread can be dislodged if
3764                          * a latent bug causes us to spin indefinitely.
3765                          */
3766                         if (next > (uintptr_t)buf + size) {
3767                                 pr_getprot_done(&tmp);
3768                                 AS_LOCK_EXIT(as, &as->a_lock);

3770                                 kmem_free(buf, size);

3772                                 if (ISSIG(curthread, JUSTLOOKING))
3773                                         return (EINTR);

3775                                 goto again;
3776                         }

3778                         php->pr_nmap++;
3779                         php->pr_npage += npage;
3780                         pmp->pr_vaddr = saddr;
3781                         pmp->pr_npage = npage;
3782                         pmp->pr_off = segop_getoffset(seg, saddr);
3782                         pmp->pr_off = SEGOP_GETOFFSET(seg, saddr);
3783                         pmp->pr_mflags = 0;
3784                         if (prot & PROT_READ)
3785                                 pmp->pr_mflags |= MA_READ;
3786                         if (prot & PROT_WRITE)
3787                                 pmp->pr_mflags |= MA_WRITE;
3788                         if (prot & PROT_EXEC)
3789                                 pmp->pr_mflags |= MA_EXEC;
3790                         if (segop_gettype(seg, saddr) & MAP_SHARED)
3790                         if (SEGOP_GETTYPE(seg, saddr) & MAP_SHARED)
3791                                 pmp->pr_mflags |= MA_SHARED;
3792                         pmp->pr_pagesize = PAGESIZE;
3793                         hat_getstat(as, saddr, len, hatid,
3794                             (char *)(pmp + 1), HAT_SYNC_ZERORM);
3795                         pmp = (prasmap_t *)next;
3796                 }
3797                 ASSERT(tmp == NULL);
3798         } while ((seg = AS_SEGNEXT(as, seg)) != NULL);

3800         AS_LOCK_EXIT(as, &as->a_lock);

3802         ASSERT((uintptr_t)pmp <= (uintptr_t)buf + size);
3803         error = uiomove(buf, (caddr_t)pmp - buf, UIO_READ, uiop);
3804         kmem_free(buf, size);

3806         return (error);
3807 }

3809 #ifdef _SYSCALL32_IMPL
3810 int
3811 oprpdread32(struct as *as, uint_t hatid, struct uio *uiop)
3812 {
3813         caddr_t buf;
3814         size_t size;
3815         ioc_prpageheader32_t *php;
3816         ioc_prasmap32_t *pmp;
3817         struct seg *seg;
3818         int error;

3820 again:
3821         AS_LOCK_ENTER(as, &as->a_lock, RW_WRITER);

3823         if ((seg = AS_SEGFIRST(as)) == NULL) {
3824                 AS_LOCK_EXIT(as, &as->a_lock);
```



```
3825                 return (0);
3826         }
3827         size = oprpdsize32(as);
3828         if (uiop->uio_resid < size) {
3829                 AS_LOCK_EXIT(as, &as->a_lock);
3830                 return (E2BIG);
3831         }

3833         buf = kmem_zalloc(size, KM_SLEEP);
3834         php = (ioc_prpageheader32_t *)buf;
3835         pmp = (ioc_prasmap32_t *)(buf + sizeof (ioc_prpageheader32_t));

3837         hrt2ts32(gethrtime(), &php->pr_tstamp);
3838         php->pr_nmap = 0;
3839         php->pr_npage = 0;
3840         do {
3841                 caddr_t eaddr = seg->s_base + pr_getsegsize(seg, 0);
3842                 caddr_t saddr, naddr;
3843                 void *tmp = NULL;

3845                 for (saddr = seg->s_base; saddr < eaddr; saddr = naddr) {
3846                         size_t len;
3847                         size_t npage;
3848                         uint_t prot;
3849                         uintptr_t next;

3851                         prot = pr_getprot(seg, 0, &tmp, &saddr, &naddr, eaddr);
3852                         if ((len = naddr - saddr) == 0)
3853                                 continue;
3854                         npage = len / PAGESIZE;
3855                         next = (uintptr_t)(pmp + 1) + round4(npage);
3856                         /*
3857                          * It's possible that the address space can change
3858                          * subtlely even though we're holding as->a_lock
3859                          * due to the nondeterminism of page_exists() in
3860                          * the presence of asychronously flushed pages or
3861                          * mapped files whose sizes are changing.
3862                          * page_exists() may be called indirectly from
3863                          * pr_getprot() by a segop_incore() routine.
3863                          * pr_getprot() by a SEGOP_INCORE() routine.
3864                          * If this happens we need to make sure we don't
3865                          * overrun the buffer whose size we computed based
3866                          * on the initial iteration through the segments.
3867                          * Once we've detected an overflow, we need to clean
3868                          * up the temporary memory allocated in pr_getprot()
3869                          * and retry. If there's a pending signal, we return
3870                          * EINTR so that this thread can be dislodged if
3871                          * a latent bug causes us to spin indefinitely.
3872                          */
3873                         if (next > (uintptr_t)buf + size) {
3874                                 pr_getprot_done(&tmp);
3875                                 AS_LOCK_EXIT(as, &as->a_lock);

3877                                 kmem_free(buf, size);

3879                                 if (ISSIG(curthread, JUSTLOOKING))
3880                                         return (EINTR);

3882                                 goto again;
3883                         }

3885                         php->pr_nmap++;
3886                         php->pr_npage += npage;
3887                         pmp->pr_vaddr = (uint32_t)(uintptr_t)saddr;
3888                         pmp->pr_npage = (uint32_t)npage;
3889                         pmp->pr_off = (int32_t)segop_getoffset(seg, saddr);
```

```
3889                        pmp->pr_off = (int32_t)SEGOP_GETOFFSET(seg, saddr);
3890                        pmp->pr_mflags = 0;
3891                        if (prot & PROT_READ)
3892                                pmp->pr_mflags |= MA_READ;
3893                        if (prot & PROT_WRITE)
3894                                pmp->pr_mflags |= MA_WRITE;
3895                        if (prot & PROT_EXEC)
3896                                pmp->pr_mflags |= MA_EXEC;
3897                        if (segop_gettype(seg, saddr) & MAP_SHARED)
3897                        if (SEGOP_GETTYPE(seg, saddr) & MAP_SHARED)
3898                                pmp->pr_mflags |= MA_SHARED;
3899                        pmp->pr_pagesize = PAGESIZE;
3900                        hat_getstat(as, saddr, len, hatid,
3901                            (char *)(pmp + 1), HAT_SYNC_ZERORM);
3902                        pmp = (ioc_prasmap32_t *)next;
3903                }
3904                ASSERT(tmp == NULL);
3905        } while ((seg = AS_SEGNEXT(as, seg)) != NULL);

3907        AS_LOCK_EXIT(as, &as->a_lock);

3909        ASSERT((uintptr_t)pmp == (uintptr_t)buf + size);
3910        error = uiomove(buf, (caddr_t)pmp - buf, UIO_READ, uiop);
3911        kmem_free(buf, size);

3913        return (error);
3914 }
```

_____unchanged_portion_omitted_

```
**********************************************************
  112635 Fri May  8 18:04:04 2015
new/usr/src/uts/common/fs/proc/prsubr.c
patch lower-case-segops
**********************************************************
______unchanged_portion_omitted_

1475 struct seg *
1476 break_seg(proc_t *p)
1477 {
1478         caddr_t addr = p->p_brkbase;
1479         struct seg *seg;
1480         struct vnode *vp;

1482         if (p->p_brksize != 0)
1483                 addr += p->p_brksize - 1;
1484         seg = as_segat(p->p_as, addr);
1485         if (seg != NULL && seg->s_ops == &segvn_ops &&
1486             (segop_getvp(seg, seg->s_base, &vp) != 0 || vp == NULL))
1486             (SEGOP_GETVP(seg, seg->s_base, &vp) != 0 || vp == NULL))
1487                 return (seg);
1488         return (NULL);
1489 }
______unchanged_portion_omitted_

1607 /*
1608  * Return an array of structures with memory map information.
1609  * We allocate here; the caller must deallocate.
1610  */
1611 int
1612 prgetmap(proc_t *p, int reserved, list_t *iolhead)
1613 {
1614         struct as *as = p->p_as;
1615         prmap_t *mp;
1616         struct seg *seg;
1617         struct seg *brkseg, *stkseg;
1618         struct vnode *vp;
1619         struct vattr vattr;
1620         uint_t prot;

1622         ASSERT(as != &kas && AS_WRITE_HELD(as, &as->a_lock));

1624         /*
1625          * Request an initial buffer size that doesn't waste memory
1626          * if the address space has only a small number of segments.
1627          */
1628         pr_iol_initlist(iolhead, sizeof (*mp), avl_numnodes(&as->a_segtree));

1630         if ((seg = AS_SEGFIRST(as)) == NULL)
1631                 return (0);

1633         brkseg = break_seg(p);
1634         stkseg = as_segat(as, prgetstackbase(p));

1636         do {
1637                 caddr_t eaddr = seg->s_base + pr_getsegsize(seg, reserved);
1638                 caddr_t saddr, naddr;
1639                 void *tmp = NULL;

1641                 for (saddr = seg->s_base; saddr < eaddr; saddr = naddr) {
1642                         prot = pr_getprot(seg, reserved, &tmp,
1643                             &saddr, &naddr, eaddr);
1644                         if (saddr == naddr)
1645                                 continue;

1647                         mp = pr_iol_newbuf(iolhead, sizeof (*mp));
```



```
1649                         mp->pr_vaddr = (uintptr_t)saddr;
1650                         mp->pr_size = naddr - saddr;
1651                         mp->pr_offset = segop_getoffset(seg, saddr);
1651                         mp->pr_offset = SEGOP_GETOFFSET(seg, saddr);
1652                         mp->pr_mflags = 0;
1653                         if (prot & PROT_READ)
1654                                 mp->pr_mflags |= MA_READ;
1655                         if (prot & PROT_WRITE)
1656                                 mp->pr_mflags |= MA_WRITE;
1657                         if (prot & PROT_EXEC)
1658                                 mp->pr_mflags |= MA_EXEC;
1659                         if (segop_gettype(seg, saddr) & MAP_SHARED)
1659                         if (SEGOP_GETTYPE(seg, saddr) & MAP_SHARED)
1660                                 mp->pr_mflags |= MA_SHARED;
1661                         if (segop_gettype(seg, saddr) & MAP_NORESERVE)
1661                         if (SEGOP_GETTYPE(seg, saddr) & MAP_NORESERVE)
1662                                 mp->pr_mflags |= MA_NORESERVE;
1663                         if (seg->s_ops == &segspt_shmops ||
1664                             (seg->s_ops == &segvn_ops &&
1665                             (segop_getvp(seg, saddr, &vp) != 0 || vp == NULL)))
1665                             (SEGOP_GETVP(seg, saddr, &vp) != 0 || vp == NULL)))
1666                                 mp->pr_mflags |= MA_ANON;
1667                         if (seg == brkseg)
1668                                 mp->pr_mflags |= MA_BREAK;
1669                         else if (seg == stkseg) {
1670                                 mp->pr_mflags |= MA_STACK;
1671                                 if (reserved) {
1672                                         size_t maxstack =
1673                                             ((size_t)p->p_stk_ctl +
1674                                             PAGEOFFSET) & PAGEMASK;
1675                                         mp->pr_vaddr =
1676                                             (uintptr_t)prgetstackbase(p) +
1677                                             p->p_stksize - maxstack;
1678                                         mp->pr_size = (uintptr_t)naddr -
1679                                             mp->pr_vaddr;
1680                                 }
1681                         }
1682                         if (seg->s_ops == &segspt_shmops)
1683                                 mp->pr_mflags |= MA_ISM | MA_SHM;
1684                         mp->pr_pagesize = PAGESIZE;

1686                         /*
1687                          * Manufacture a filename for the "object" directory.
1688                          */
1689                         vattr.va_mask = AT_FSID|AT_NODEID;
1690                         if (seg->s_ops == &segvn_ops &&
1691                             segop_getvp(seg, saddr, &vp) == 0 &&
1691                             SEGOP_GETVP(seg, saddr, &vp) == 0 &&
1692                             vp != NULL && vp->v_type == VREG &&
1693                             VOP_GETATTR(vp, &vattr, 0, CRED(), NULL) == 0) {
1694                                 if (vp == p->p_exec)
1695                                         (void) strcpy(mp->pr_mapname, "a.out");
1696                                 else
1697                                         pr_object_name(mp->pr_mapname,
1698                                             vp, &vattr);
1699                         }

1701                         /*
1702                          * Get the SysV shared memory id, if any.
1703                          */
1704                         if ((mp->pr_mflags & MA_SHARED) && p->p_segacct &&
1705                             (mp->pr_shmid = shmgetid(p, seg->s_base)) !=
1706                             SHMID_NONE) {
1707                                 if (mp->pr_shmid == SHMID_FREE)
1708                                         mp->pr_shmid = -1;
```

```
1710                                    mp->pr_mflags |= MA_SHM;
1711                            } else {
1712                                    mp->pr_shmid = -1;
1713                            }
1714                    }
1715                    ASSERT(tmp == NULL);
1716            } while ((seg = AS_SEGNEXT(as, seg)) != NULL);

1718            return (0);
1719    }

1721    #ifdef _SYSCALL32_IMPL
1722    int
1723    prgetmap32(proc_t *p, int reserved, list_t *iolhead)
1724    {
1725            struct as *as = p->p_as;
1726            prmap32_t *mp;
1727            struct seg *seg;
1728            struct seg *brkseg, *stkseg;
1729            struct vnode *vp;
1730            struct vattr vattr;
1731            uint_t prot;

1733            ASSERT(as != &kas && AS_WRITE_HELD(as, &as->a_lock));

1735            /*
1736             * Request an initial buffer size that doesn't waste memory
1737             * if the address space has only a small number of segments.
1738             */
1739            pr_iol_initlist(iolhead, sizeof (*mp), avl_numnodes(&as->a_segtree));

1741            if ((seg = AS_SEGFIRST(as)) == NULL)
1742                    return (0);

1744            brkseg = break_seg(p);
1745            stkseg = as_segat(as, prgetstackbase(p));

1747            do {
1748                    caddr_t eaddr = seg->s_base + pr_getsegsize(seg, reserved);
1749                    caddr_t saddr, naddr;
1750                    void *tmp = NULL;

1752                    for (saddr = seg->s_base; saddr < eaddr; saddr = naddr) {
1753                            prot = pr_getprot(seg, reserved, &tmp,
1754                                &saddr, &naddr, eaddr);
1755                            if (saddr == naddr)
1756                                    continue;

1758                            mp = pr_iol_newbuf(iolhead, sizeof (*mp));

1760                            mp->pr_vaddr = (caddr32_t)(uintptr_t)saddr;
1761                            mp->pr_size = (size32_t)(naddr - saddr);
1762                            mp->pr_offset = segop_getoffset(seg, saddr);
1762                            mp->pr_offset = SEGOP_GETOFFSET(seg, saddr);
1763                            mp->pr_mflags = 0;
1764                            if (prot & PROT_READ)
1765                                    mp->pr_mflags |= MA_READ;
1766                            if (prot & PROT_WRITE)
1767                                    mp->pr_mflags |= MA_WRITE;
1768                            if (prot & PROT_EXEC)
1769                                    mp->pr_mflags |= MA_EXEC;
1770                            if (segop_gettype(seg, saddr) & MAP_SHARED)
1770                            if (SEGOP_GETTYPE(seg, saddr) & MAP_SHARED)
1771                                    mp->pr_mflags |= MA_SHARED;
1772                            if (segop_gettype(seg, saddr) & MAP_NORESERVE)
```



```
1772                            if (SEGOP_GETTYPE(seg, saddr) & MAP_NORESERVE)
1773                                    mp->pr_mflags |= MA_NORESERVE;
1774                            if (seg->s_ops == &segspt_shmops ||
1775                                (seg->s_ops == &segvn_ops &&
1776                                segop_getvp(seg, saddr, &vp) != 0 || vp == NULL)))
1776                                (SEGOP_GETVP(seg, saddr, &vp) != 0 || vp == NULL)))
1777                                    mp->pr_mflags |= MA_ANON;
1778                            if (seg == brkseg)
1779                                    mp->pr_mflags |= MA_BREAK;
1780                            else if (seg == stkseg) {
1781                                    mp->pr_mflags |= MA_STACK;
1782                                    if (reserved) {
1783                                            size_t maxstack =
1784                                                ((size_t)p->p_stk_ctl +
1785                                                PAGEOFFSET) & PAGEMASK;
1786                                            uintptr_t vaddr =
1787                                                (uintptr_t)prgetstackbase(p) +
1788                                                p->p_stksize - maxstack;
1789                                            mp->pr_vaddr = (caddr32_t)vaddr;
1790                                            mp->pr_size = (size32_t)
1791                                                ((uintptr_t)naddr - vaddr);
1792                                    }
1793                            }
1794                            if (seg->s_ops == &segspt_shmops)
1795                                    mp->pr_mflags |= MA_ISM | MA_SHM;
1796                            mp->pr_pagesize = PAGESIZE;

1798                            /*
1799                             * Manufacture a filename for the "object" directory.
1800                             */
1801                            vattr.va_mask = AT_FSID|AT_NODEID;
1802                            if (seg->s_ops == &segvn_ops &&
1803                                segop_getvp(seg, saddr, &vp) == 0 &&
1803                                SEGOP_GETVP(seg, saddr, &vp) == 0 &&
1804                                vp != NULL && vp->v_type == VREG &&
1805                                VOP_GETATTR(vp, &vattr, 0, CRED(), NULL) == 0) {
1806                                    if (vp == p->p_exec)
1807                                            (void) strcpy(mp->pr_mapname, "a.out");
1808                                    else
1809                                            pr_object_name(mp->pr_mapname,
1810                                                vp, &vattr);
1811                            }

1813                            /*
1814                             * Get the SysV shared memory id, if any.
1815                             */
1816                            if ((mp->pr_mflags & MA_SHARED) && p->p_segacct &&
1817                                (mp->pr_shmid = shmgetid(p, seg->s_base)) !=
1818                                SHMID_NONE) {
1819                                    if (mp->pr_shmid == SHMID_FREE)
1820                                            mp->pr_shmid = -1;

1822                                    mp->pr_mflags |= MA_SHM;
1823                            } else {
1824                                    mp->pr_shmid = -1;
1825                            }
1826                    }
1827                    ASSERT(tmp == NULL);
1828            } while ((seg = AS_SEGNEXT(as, seg)) != NULL);

1830            return (0);
1831    }
_____unchanged_portion_omitted_
1895    #endif  /* _SYSCALL32_IMPL */

1897    /*
```

```
 * Read page data information.
 */
int
prpdread(proc_t *p, uint_t hatid, struct uio *uiop)
{
        struct as *as = p->p_as;
        caddr_t buf;
        size_t size;
        prpageheader_t *php;
        prasmap_t *pmp;
        struct seg *seg;
        int error;

again:
        AS_LOCK_ENTER(as, &as->a_lock, RW_WRITER);

        if ((seg = AS_SEGFIRST(as)) == NULL) {
                AS_LOCK_EXIT(as, &as->a_lock);
                return (0);
        }
        size = prpdsize(as);
        if (uiop->uio_resid < size) {
                AS_LOCK_EXIT(as, &as->a_lock);
                return (E2BIG);
        }

        buf = kmem_zalloc(size, KM_SLEEP);
        php = (prpageheader_t *)buf;
        pmp = (prasmap_t *)(buf + sizeof (prpageheader_t));

        hrt2ts(gethrtime(), &php->pr_tstamp);
        php->pr_nmap = 0;
        php->pr_npage = 0;
        do {
                caddr_t eaddr = seg->s_base + pr_getsegsize(seg, 0);
                caddr_t saddr, naddr;
                void *tmp = NULL;

                for (saddr = seg->s_base; saddr < eaddr; saddr = naddr) {
                        struct vnode *vp;
                        struct vattr vattr;
                        size_t len;
                        size_t npage;
                        uint_t prot;
                        uintptr_t next;

                        prot = pr_getprot(seg, 0, &tmp, &saddr, &naddr, eaddr);
                        if ((len = (size_t)(naddr - saddr)) == 0)
                                continue;
                        npage = len / PAGESIZE;
                        next = (uintptr_t)(pmp + 1) + round8(npage);
                        /*
                         * It's possible that the address space can change
                         * subtlely even though we're holding as->a_lock
                         * due to the nondeterminism of page_exists() in
                         * the presence of asychronously flushed pages or
                         * mapped files whose sizes are changing.
                         * page_exists() may be called indirectly from
                         * pr_getprot() by a segop_incore() routine.
                         * pr_getprot() by a SEGOP_INCORE() routine.
                         * If this happens we need to make sure we don't
                         * overrun the buffer whose size we computed based
                         * on the initial iteration through the segments.
                         * Once we've detected an overflow, we need to clean
                         * up the temporary memory allocated in pr_getprot()
                         * and retry. If there's a pending signal, we return
```



```
                         * EINTR so that this thread can be dislodged if
                         * a latent bug causes us to spin indefinitely.
                         */
                        if (next > (uintptr_t)buf + size) {
                                pr_getprot_done(&tmp);
                                AS_LOCK_EXIT(as, &as->a_lock);

                                kmem_free(buf, size);

                                if (ISSIG(curthread, JUSTLOOKING))
                                        return (EINTR);

                                goto again;
                        }

                        php->pr_nmap++;
                        php->pr_npage += npage;
                        pmp->pr_vaddr = (uintptr_t)saddr;
                        pmp->pr_npage = npage;
                        pmp->pr_offset = segop_getoffset(seg, saddr);
                        pmp->pr_offset = SEGOP_GETOFFSET(seg, saddr);
                        pmp->pr_mflags = 0;
                        if (prot & PROT_READ)
                                pmp->pr_mflags |= MA_READ;
                        if (prot & PROT_WRITE)
                                pmp->pr_mflags |= MA_WRITE;
                        if (prot & PROT_EXEC)
                                pmp->pr_mflags |= MA_EXEC;
                        if (segop_gettype(seg, saddr) & MAP_SHARED)
                        if (SEGOP_GETTYPE(seg, saddr) & MAP_SHARED)
                                pmp->pr_mflags |= MA_SHARED;
                        if (segop_gettype(seg, saddr) & MAP_NORESERVE)
                        if (SEGOP_GETTYPE(seg, saddr) & MAP_NORESERVE)
                                pmp->pr_mflags |= MA_NORESERVE;
                        if (seg->s_ops == &segspt_shmops ||
                            (seg->s_ops == &segvn_ops &&
                            (segop_getvp(seg, saddr, &vp) != 0 || vp == NULL)))
                            (SEGOP_GETVP(seg, saddr, &vp) != 0 || vp == NULL)))
                                pmp->pr_mflags |= MA_ANON;
                        if (seg->s_ops == &segspt_shmops)
                                pmp->pr_mflags |= MA_ISM | MA_SHM;
                        pmp->pr_pagesize = PAGESIZE;
                        /*
                         * Manufacture a filename for the "object" directory.
                         */
                        vattr.va_mask = AT_FSID|AT_NODEID;
                        if (seg->s_ops == &segvn_ops &&
                            segop_getvp(seg, saddr, &vp) == 0 &&
                            SEGOP_GETVP(seg, saddr, &vp) == 0 &&
                            vp != NULL && vp->v_type == VREG &&
                            VOP_GETATTR(vp, &vattr, 0, CRED(), NULL) == 0) {
                                if (vp == p->p_exec)
                                        (void) strcpy(pmp->pr_mapname, "a.out");
                                else
                                        pr_object_name(pmp->pr_mapname,
                                            vp, &vattr);
                        }

                        /*
                         * Get the SysV shared memory id, if any.
                         */
                        if ((pmp->pr_mflags & MA_SHARED) && p->p_segacct &&
                            (pmp->pr_shmid = shmgetid(p, seg->s_base)) !=
                            SHMID_NONE) {
                                if (pmp->pr_shmid == SHMID_FREE)
                                        pmp->pr_shmid = -1;
```

```
2025                                pmp->pr_mflags |= MA_SHM;
2026                        } else {
2027                                pmp->pr_shmid = -1;
2028                        }

2030                        hat_getstat(as, saddr, len, hatid,
2031                            (char *)(pmp + 1), HAT_SYNC_ZERORM);
2032                        pmp = (prasmap_t *)next;
2033                }
2034                ASSERT(tmp == NULL);
2035        } while ((seg = AS_SEGNEXT(as, seg)) != NULL);

2037        AS_LOCK_EXIT(as, &as->a_lock);

2039        ASSERT((uintptr_t)pmp <= (uintptr_t)buf + size);
2040        error = uiomove(buf, (caddr_t)pmp - buf, UIO_READ, uiop);
2041        kmem_free(buf, size);

2043        return (error);
2044 }

2046 #ifdef _SYSCALL32_IMPL
2047 int
2048 prpdread32(proc_t *p, uint_t hatid, struct uio *uiop)
2049 {
2050        struct as *as = p->p_as;
2051        caddr_t buf;
2052        size_t size;
2053        prpageheader32_t *php;
2054        prasmap32_t *pmp;
2055        struct seg *seg;
2056        int error;

2058 again:
2059        AS_LOCK_ENTER(as, &as->a_lock, RW_WRITER);

2061        if ((seg = AS_SEGFIRST(as)) == NULL) {
2062                AS_LOCK_EXIT(as, &as->a_lock);
2063                return (0);
2064        }
2065        size = prpdsize32(as);
2066        if (uiop->uio_resid < size) {
2067                AS_LOCK_EXIT(as, &as->a_lock);
2068                return (E2BIG);
2069        }

2071        buf = kmem_zalloc(size, KM_SLEEP);
2072        php = (prpageheader32_t *)buf;
2073        pmp = (prasmap32_t *)(buf + sizeof (prpageheader32_t));

2075        hrt2ts32(gethrtime(), &php->pr_tstamp);
2076        php->pr_nmap = 0;
2077        php->pr_npage = 0;
2078        do {
2079                caddr_t eaddr = seg->s_base + pr_getsegsize(seg, 0);
2080                caddr_t saddr, naddr;
2081                void *tmp = NULL;

2083                for (saddr = seg->s_base; saddr < eaddr; saddr = naddr) {
2084                        struct vnode *vp;
2085                        struct vattr vattr;
2086                        size_t len;
2087                        size_t npage;
2088                        uint_t prot;
2089                        uintptr_t next;
```



```
2091                        prot = pr_getprot(seg, 0, &tmp, &saddr, &naddr, eaddr);
2092                        if ((len = (size_t)(naddr - saddr)) == 0)
2093                                continue;
2094                        npage = len / PAGESIZE;
2095                        next = (uintptr_t)(pmp + 1) + round8(npage);
2096                        /*
2097                         * It's possible that the address space can change
2098                         * subtlely even though we're holding as->a_lock
2099                         * due to the nondeterminism of page_exists() in
2100                         * the presence of asychronously flushed pages or
2101                         * mapped files whose sizes are changing.
2102                         * page_exists() may be called indirectly from
2103                         * pr_getprot() by a segop_incore() routine.
2103                         * pr_getprot() by a SEGOP_INCORE() routine.
2104                         * If this happens we need to make sure we don't
2105                         * overrun the buffer whose size we computed based
2106                         * on the initial iteration through the segments.
2107                         * Once we've detected an overflow, we need to clean
2108                         * up the temporary memory allocated in pr_getprot()
2109                         * and retry. If there's a pending signal, we return
2110                         * EINTR so that this thread can be dislodged if
2111                         * a latent bug causes us to spin indefinitely.
2112                         */
2113                        if (next > (uintptr_t)buf + size) {
2114                                pr_getprot_done(&tmp);
2115                                AS_LOCK_EXIT(as, &as->a_lock);

2117                                kmem_free(buf, size);

2119                                if (ISSIG(curthread, JUSTLOOKING))
2120                                        return (EINTR);

2122                                goto again;
2123                        }

2125                        php->pr_nmap++;
2126                        php->pr_npage += npage;
2127                        pmp->pr_vaddr = (caddr32_t)(uintptr_t)saddr;
2128                        pmp->pr_npage = (size32_t)npage;
2129                        pmp->pr_offset = segop_getoffset(seg, saddr);
2129                        pmp->pr_offset = SEGOP_GETOFFSET(seg, saddr);
2130                        pmp->pr_mflags = 0;
2131                        if (prot & PROT_READ)
2132                                pmp->pr_mflags |= MA_READ;
2133                        if (prot & PROT_WRITE)
2134                                pmp->pr_mflags |= MA_WRITE;
2135                        if (prot & PROT_EXEC)
2136                                pmp->pr_mflags |= MA_EXEC;
2137                        if (segop_gettype(seg, saddr) & MAP_SHARED)
2137                        if (SEGOP_GETTYPE(seg, saddr) & MAP_SHARED)
2138                                pmp->pr_mflags |= MA_SHARED;
2139                        if (segop_gettype(seg, saddr) & MAP_NORESERVE)
2139                        if (SEGOP_GETTYPE(seg, saddr) & MAP_NORESERVE)
2140                                pmp->pr_mflags |= MA_NORESERVE;
2141                        if (seg->s_ops == &segspt_shmops ||
2142                            (seg->s_ops == &segvn_ops &&
2143                            (segop_getvp(seg, saddr, &vp) != 0 || vp == NULL)))
2143                            (SEGOP_GETVP(seg, saddr, &vp) != 0 || vp == NULL)))
2144                                pmp->pr_mflags |= MA_ANON;
2145                        if (seg->s_ops == &segspt_shmops)
2146                                pmp->pr_mflags |= MA_ISM | MA_SHM;
2147                        pmp->pr_pagesize = PAGESIZE;
2148                        /*
2149                         * Manufacture a filename for the "object" directory.
2150                         */
```

```
2151                        vattr.va_mask = AT_FSID|AT_NODEID;
2152                        if (seg->s_ops == &segvn_ops &&
2153                            segop_getvp(seg, saddr, &vp) == 0 &&
2153                            SEGOP_GETVP(seg, saddr, &vp) == 0 &&
2154                            vp != NULL && vp->v_type == VREG &&
2155                            VOP_GETATTR(vp, &vattr, 0, CRED(), NULL) == 0) {
2156                                if (vp == p->p_exec)
2157                                        (void) strcpy(pmp->pr_mapname, "a.out");
2158                                else
2159                                        pr_object_name(pmp->pr_mapname,
2160                                            vp, &vattr);
2161                        }

2163                        /*
2164                         * Get the SysV shared memory id, if any.
2165                         */
2166                        if ((pmp->pr_mflags & MA_SHARED) && p->p_segacct &&
2167                            (pmp->pr_shmid = shmgetid(p, seg->s_base)) !=
2168                            SHMID_NONE) {
2169                                if (pmp->pr_shmid == SHMID_FREE)
2170                                        pmp->pr_shmid = -1;

2172                                pmp->pr_mflags |= MA_SHM;
2173                        } else {
2174                                pmp->pr_shmid = -1;
2175                        }

2177                        hat_getstat(as, saddr, len, hatid,
2178                            (char *)(pmp + 1), HAT_SYNC_ZERORM);
2179                        pmp = (prasmap32_t *)next;
2180                }
2181                ASSERT(tmp == NULL);
2182        } while ((seg = AS_SEGNEXT(as, seg)) != NULL);

2184        AS_LOCK_EXIT(as, &as->a_lock);

2186        ASSERT((uintptr_t)pmp <= (uintptr_t)buf + size);
2187        error = uiomove(buf, (caddr_t)pmp - buf, UIO_READ, uiop);
2188        kmem_free(buf, size);

2190        return (error);
2191 }
_____unchanged_portion_omitted_

3299 /*
3300  * This one is called by the traced process to unwatch all the
3301  * pages while deallocating the list of watched_page structs.
3302  */
3303 void
3304 pr_free_watched_pages(proc_t *p)
3305 {
3306        struct as *as = p->p_as;
3307        struct watched_page *pwp;
3308        uint_t prot;
3309        int    retrycnt, err;
3310        void *cookie;

3312        if (as == NULL || avl_numnodes(&as->a_wpage) == 0)
3313                return;

3315        ASSERT(MUTEX_NOT_HELD(&curproc->p_lock));
3316        AS_LOCK_ENTER(as, &as->a_lock, RW_WRITER);

3318        pwp = avl_first(&as->a_wpage);

3320        cookie = NULL;
```



```
3321        while ((pwp = avl_destroy_nodes(&as->a_wpage, &cookie)) != NULL) {
3322                retrycnt = 0;
3323                if ((prot = pwp->wp_oprot) != 0) {
3324                        caddr_t addr = pwp->wp_vaddr;
3325                        struct seg *seg;
3326                retry:

3328                        if ((pwp->wp_prot != prot ||
3329                            (pwp->wp_flags & WP_NOWATCH)) &&
3330                            (seg = as_segat(as, addr)) != NULL) {
3331                                err = segop_setprot(seg, addr, PAGESIZE, prot);
3331                                err = SEGOP_SETPROT(seg, addr, PAGESIZE, prot);
3332                                if (err == IE_RETRY) {
3333                                        ASSERT(retrycnt == 0);
3334                                        retrycnt++;
3335                                        goto retry;
3336                                }
3337                        }
3338                }
3339                kmem_free(pwp, sizeof (struct watched_page));
3340        }

3342        avl_destroy(&as->a_wpage);
3343        p->p_wprot = NULL;

3345        AS_LOCK_EXIT(as, &as->a_lock);
3346 }

3348 /*
3349  * Insert a watched area into the list of watched pages.
3350  * If oflags is zero then we are adding a new watched area.
3351  * Otherwise we are changing the flags of an existing watched area.
3352  */
3353 static int
3354 set_watched_page(proc_t *p, caddr_t vaddr, caddr_t eaddr,
3355        ulong_t flags, ulong_t oflags)
3356 {
3357        struct as *as = p->p_as;
3358        avl_tree_t *pwp_tree;
3359        struct watched_page *pwp, *newpwp;
3360        struct watched_page tpw;
3361        avl_index_t where;
3362        struct seg *seg;
3363        uint_t prot;
3364        caddr_t addr;

3366        /*
3367         * We need to pre-allocate a list of structures before we grab the
3368         * address space lock to avoid calling kmem_alloc(KM_SLEEP) with locks
3369         * held.
3370         */
3371        newpwp = NULL;
3372        for (addr = (caddr_t)((uintptr_t)vaddr & (uintptr_t)PAGEMASK);
3373            addr < eaddr; addr += PAGESIZE) {
3374                pwp = kmem_zalloc(sizeof (struct watched_page), KM_SLEEP);
3375                pwp->wp_list = newpwp;
3376                newpwp = pwp;
3377        }

3379        AS_LOCK_ENTER(as, &as->a_lock, RW_WRITER);

3381        /*
3382         * Search for an existing watched page to contain the watched area.
3383         * If none is found, grab a new one from the available list
3384         * and insert it in the active list, keeping the list sorted
3385         * by user-level virtual address.
```

```
3386          */
3387         if (p->p_flag & SVFWAIT)
3388                 pwp_tree = &p->p_wpage;
3389         else
3390                 pwp_tree = &as->a_wpage;

3392 again:
3393         if (avl_numnodes(pwp_tree) > prnwatch) {
3394                 AS_LOCK_EXIT(as, &as->a_lock);
3395                 while (newpwp != NULL) {
3396                         pwp = newpwp->wp_list;
3397                         kmem_free(newpwp, sizeof (struct watched_page));
3398                         newpwp = pwp;
3399                 }
3400                 return (E2BIG);
3401         }

3403         tpw.wp_vaddr = (caddr_t)((uintptr_t)vaddr & (uintptr_t)PAGEMASK);
3404         if ((pwp = avl_find(pwp_tree, &tpw, &where)) == NULL) {
3405                 pwp = newpwp;
3406                 newpwp = newpwp->wp_list;
3407                 pwp->wp_list = NULL;
3408                 pwp->wp_vaddr = (caddr_t)((uintptr_t)vaddr &
3409                     (uintptr_t)PAGEMASK);
3410                 avl_insert(pwp_tree, pwp, where);
3411         }

3413         ASSERT(vaddr >= pwp->wp_vaddr && vaddr < pwp->wp_vaddr + PAGESIZE);

3415         if (oflags & WA_READ)
3416                 pwp->wp_read--;
3417         if (oflags & WA_WRITE)
3418                 pwp->wp_write--;
3419         if (oflags & WA_EXEC)
3420                 pwp->wp_exec--;

3422         ASSERT(pwp->wp_read >= 0);
3423         ASSERT(pwp->wp_write >= 0);
3424         ASSERT(pwp->wp_exec >= 0);

3426         if (flags & WA_READ)
3427                 pwp->wp_read++;
3428         if (flags & WA_WRITE)
3429                 pwp->wp_write++;
3430         if (flags & WA_EXEC)
3431                 pwp->wp_exec++;

3433         if (!(p->p_flag & SVFWAIT)) {
3434                 vaddr = pwp->wp_vaddr;
3435                 if (pwp->wp_oprot == 0 &&
3436                     (seg = as_segat(as, vaddr)) != NULL) {
3437                         segop_getprot(seg, vaddr, 0, &prot);
3437                         SEGOP_GETPROT(seg, vaddr, 0, &prot);
3438                         pwp->wp_oprot = (uchar_t)prot;
3439                         pwp->wp_prot = (uchar_t)prot;
3440                 }
3441                 if (pwp->wp_oprot != 0) {
3442                         prot = pwp->wp_oprot;
3443                         if (pwp->wp_read)
3444                                 prot &= ~(PROT_READ|PROT_WRITE|PROT_EXEC);
3445                         if (pwp->wp_write)
3446                                 prot &= ~PROT_WRITE;
3447                         if (pwp->wp_exec)
3448                                 prot &= ~(PROT_READ|PROT_WRITE|PROT_EXEC);
3449                         if (!(pwp->wp_flags & WP_NOWATCH) &&
3450                             pwp->wp_prot != prot &&
```



```
3451                             (pwp->wp_flags & WP_SETPROT) == 0) {
3452                                 pwp->wp_flags |= WP_SETPROT;
3453                                 pwp->wp_list = p->p_wprot;
3454                                 p->p_wprot = pwp;
3455                         }
3456                         pwp->wp_prot = (uchar_t)prot;
3457                 }
3458         }

3460         /*
3461          * If the watched area extends into the next page then do
3462          * it over again with the virtual address of the next page.
3463          */
3464         if ((vaddr = pwp->wp_vaddr + PAGESIZE) < eaddr)
3465                 goto again;

3467         AS_LOCK_EXIT(as, &as->a_lock);

3469         /*
3470          * Free any pages we may have over-allocated
3471          */
3472         while (newpwp != NULL) {
3473                 pwp = newpwp->wp_list;
3474                 kmem_free(newpwp, sizeof (struct watched_page));
3475                 newpwp = pwp;
3476         }

3478         return (0);
3479 }
```

_____unchanged_portion_omitted_

```
3614 static caddr_t
3615 pr_pagev_fill(prpagev_t *pagev, struct seg *seg, caddr_t addr, caddr_t eaddr)
3616 {
3617         ulong_t lastpg = seg_page(seg, eaddr - 1);
3618         ulong_t pn, pnlim;
3619         caddr_t saddr;
3620         size_t len;

3622         ASSERT(addr >= seg->s_base && addr <= eaddr);

3624         if (addr == eaddr)
3625                 return (eaddr);

3627 refill:
3628         ASSERT(addr < eaddr);
3629         pagev->pg_pnbase = seg_page(seg, addr);
3630         pnlim = pagev->pg_pnbase + pagev->pg_npages;
3631         saddr = addr;

3633         if (lastpg < pnlim)
3634                 len = (size_t)(eaddr - addr);
3635         else
3636                 len = pagev->pg_npages * PAGESIZE;

3638         if (pagev->pg_incore != NULL) {
3639                 /*
3640                  * INCORE cleverly has different semantics than GETPROT:
3641                  * it returns info on pages up to but NOT including addr + len.
3642                  */
3643                 segop_incore(seg, addr, len, pagev->pg_incore);
3643                 SEGOP_INCORE(seg, addr, len, pagev->pg_incore);
3644                 pn = pagev->pg_pnbase;

3646                 do {
3647                         /*
```

```
3648                         * Guilty knowledge here:  We know that segvn_incore
3649                         * returns more than just the low-order bit that
3650                         * indicates the page is actually in memory.  If any
3651                         * bits are set, then the page has backing store.
3652                         */
3653                        if (pagev->pg_incore[pn++ - pagev->pg_pnbase])
3654                                goto out;

3656                } while ((addr += PAGESIZE) < eaddr && pn < pnlim);

3658                /*
3659                 * If we examined all the pages in the vector but we're not
3660                 * at the end of the segment, take another lap.
3661                 */
3662                if (addr < eaddr)
3663                        goto refill;
3664        }

3666        /*
3667         * Need to take len - 1 because addr + len is the address of the
3668         * first byte of the page just past the end of what we want.
3669         */
3670 out:
3671        segop_getprot(seg, saddr, len - 1, pagev->pg_protv);
3671        SEGOP_GETPROT(seg, saddr, len - 1, pagev->pg_protv);
3672        return (addr);
3673 }
______unchanged_portion_omitted_

3768 size_t
3769 pr_getsegsize(struct seg *seg, int reserved)
3770 {
3771        size_t size = seg->s_size;

3773        /*
3774         * If we're interested in the reserved space, return the size of the
3775         * segment itself.  Everything else in this function is a special case
3776         * to determine the actual underlying size of various segment types.
3777         */
3778        if (reserved)
3779                return (size);

3781        /*
3782         * If this is a segvn mapping of a regular file, return the smaller
3783         * of the segment size and the remaining size of the file beyond
3784         * the file offset corresponding to seg->s_base.
3785         */
3786        if (seg->s_ops == &segvn_ops) {
3787                vattr_t vattr;
3788                vnode_t *vp;

3790                vattr.va_mask = AT_SIZE;

3792                if (segop_getvp(seg, seg->s_base, &vp) == 0 &&
3792                if (SEGOP_GETVP(seg, seg->s_base, &vp) == 0 &&
3793                    vp != NULL && vp->v_type == VREG &&
3794                    VOP_GETATTR(vp, &vattr, 0, CRED(), NULL) == 0) {

3796                        u_offset_t fsize = vattr.va_size;
3797                        u_offset_t offset = segop_getoffset(seg, seg->s_base);
3797                        u_offset_t offset = SEGOP_GETOFFSET(seg, seg->s_base);

3799                        if (fsize < offset)
3800                                fsize = 0;
3801                        else
3802                                fsize -= offset;
```



```
3804                        fsize = roundup(fsize, (u_offset_t)PAGESIZE);

3806                        if (fsize < (u_offset_t)size)
3807                                size = (size_t)fsize;
3808                }

3810                return (size);
3811        }

3813        /*
3814         * If this is an ISM shared segment, don't include pages that are
3815         * beyond the real size of the spt segment that backs it.
3816         */
3817        if (seg->s_ops == &segspt_shmops)
3818                return (MIN(spt_realsize(seg), size));

3820        /*
3821         * If this is segment is a mapping from /dev/null, then this is a
3822         * reservation of virtual address space and has no actual size.
3823         * Such segments are backed by segdev and have type set to neither
3824         * MAP_SHARED nor MAP_PRIVATE.
3825         */
3826        if (seg->s_ops == &segdev_ops &&
3827            ((segop_gettype(seg, seg->s_base) &
3827            ((SEGOP_GETTYPE(seg, seg->s_base) &
3828            (MAP_SHARED | MAP_PRIVATE)) == 0))
3829                return (0);

3831        /*
3832         * If this segment doesn't match one of the special types we handle,
3833         * just return the size of the segment itself.
3834         */
3835        return (size);
3836 }
______unchanged_portion_omitted_

3997 /*
3998  * Return an array of structures with extended memory map information.
3999  * We allocate here; the caller must deallocate.
4000  */
4001 int
4002 prgetxmap(proc_t *p, list_t *iolhead)
4003 {
4004        struct as *as = p->p_as;
4005        prxmap_t *mp;
4006        struct seg *seg;
4007        struct seg *brkseg, *stkseg;
4008        struct vnode *vp;
4009        struct vattr vattr;
4010        uint_t prot;

4012        ASSERT(as != &kas && AS_WRITE_HELD(as, &as->a_lock));

4014        /*
4015         * Request an initial buffer size that doesn't waste memory
4016         * if the address space has only a small number of segments.
4017         */
4018        pr_iol_initlist(iolhead, sizeof (*mp), avl_numnodes(&as->a_segtree));

4020        if ((seg = AS_SEGFIRST(as)) == NULL)
4021                return (0);

4023        brkseg = break_seg(p);
4024        stkseg = as_segat(as, prgetstackbase(p));
```

```
4026         do {
4027                 caddr_t eaddr = seg->s_base + pr_getsegsize(seg, 0);
4028                 caddr_t saddr, naddr, baddr;
4029                 void *tmp = NULL;
4030                 ssize_t psz;
4031                 char *parr;
4032                 uint64_t npages;
4033                 uint64_t pagenum;

4035                 /*
4036                  * Segment loop part one: iterate from the base of the segment
4037                  * to its end, pausing at each address boundary (baddr) between
4038                  * ranges that have different virtual memory protections.
4039                  */
4040                 for (saddr = seg->s_base; saddr < eaddr; saddr = baddr) {
4041                         prot = pr_getprot(seg, 0, &tmp, &saddr, &baddr, eaddr);
4042                         ASSERT(baddr >= saddr && baddr <= eaddr);

4044                         /*
4045                          * Segment loop part two: iterate from the current
4046                          * position to the end of the protection boundary,
4047                          * pausing at each address boundary (naddr) between
4048                          * ranges that have different underlying page sizes.
4049                          */
4050                         for (; saddr < baddr; saddr = naddr) {
4051                                 psz = pr_getpagesize(seg, saddr, &naddr, baddr);
4052                                 ASSERT(naddr >= saddr && naddr <= baddr);

4054                                 mp = pr_iol_newbuf(iolhead, sizeof (*mp));

4056                                 mp->pr_vaddr = (uintptr_t)saddr;
4057                                 mp->pr_size = naddr - saddr;
4058                                 mp->pr_offset = segop_getoffset(seg, saddr);
4058                                 mp->pr_offset = SEGOP_GETOFFSET(seg, saddr);
4059                                 mp->pr_mflags = 0;
4060                                 if (prot & PROT_READ)
4061                                         mp->pr_mflags |= MA_READ;
4062                                 if (prot & PROT_WRITE)
4063                                         mp->pr_mflags |= MA_WRITE;
4064                                 if (prot & PROT_EXEC)
4065                                         mp->pr_mflags |= MA_EXEC;
4066                                 if (segop_gettype(seg, saddr) & MAP_SHARED)
4066                                 if (SEGOP_GETTYPE(seg, saddr) & MAP_SHARED)
4067                                         mp->pr_mflags |= MA_SHARED;
4068                                 if (segop_gettype(seg, saddr) & MAP_NORESERVE)
4068                                 if (SEGOP_GETTYPE(seg, saddr) & MAP_NORESERVE)
4069                                         mp->pr_mflags |= MA_NORESERVE;
4070                                 if (seg->s_ops == &segspt_shmops ||
4071                                     (seg->s_ops == &segvn_ops &&
4072                                     (segop_getvp(seg, saddr, &vp) != 0 ||
4072                                     (SEGOP_GETVP(seg, saddr, &vp) != 0 ||
4073                                     vp == NULL)))
4074                                         mp->pr_mflags |= MA_ANON;
4075                                 if (seg == brkseg)
4076                                         mp->pr_mflags |= MA_BREAK;
4077                                 else if (seg == stkseg)
4078                                         mp->pr_mflags |= MA_STACK;
4079                                 if (seg->s_ops == &segspt_shmops)
4080                                         mp->pr_mflags |= MA_ISM | MA_SHM;

4082                                 mp->pr_pagesize = PAGESIZE;
4083                                 if (psz == -1) {
4084                                         mp->pr_hatpagesize = 0;
4085                                 } else {
4086                                         mp->pr_hatpagesize = psz;
4087                                 }
```



```
4089                                 /*
4090                                  * Manufacture a filename for the "object" dir.
4091                                  */
4092                                 mp->pr_dev = PRNODEV;
4093                                 vattr.va_mask = AT_FSID|AT_NODEID;
4094                                 if (seg->s_ops == &segvn_ops &&
4095                                     segop_getvp(seg, saddr, &vp) == 0 &&
4095                                     SEGOP_GETVP(seg, saddr, &vp) == 0 &&
4096                                     vp != NULL && vp->v_type == VREG &&
4097                                     VOP_GETATTR(vp, &vattr, 0, CRED(),
4098                                     NULL) == 0) {
4099                                         mp->pr_dev = vattr.va_fsid;
4100                                         mp->pr_ino = vattr.va_nodeid;
4101                                         if (vp == p->p_exec)
4102                                                 (void) strcpy(mp->pr_mapname,
4103                                                     "a.out");
4104                                         else
4105                                                 pr_object_name(mp->pr_mapname,
4106                                                     vp, &vattr);
4107                                 }

4109                                 /*
4110                                  * Get the SysV shared memory id, if any.
4111                                  */
4112                                 if ((mp->pr_mflags & MA_SHARED) &&
4113                                     p->p_segacct && (mp->pr_shmid = shmgetid(p,
4114                                     seg->s_base)) != SHMID_NONE) {
4115                                         if (mp->pr_shmid == SHMID_FREE)
4116                                                 mp->pr_shmid = -1;

4118                                         mp->pr_mflags |= MA_SHM;
4119                                 } else {
4120                                         mp->pr_shmid = -1;
4121                                 }

4123                                 npages = ((uintptr_t)(naddr - saddr)) >>
4124                                     PAGESHIFT;
4125                                 parr = kmem_zalloc(npages, KM_SLEEP);

4127                                 segop_incore(seg, saddr, naddr - saddr, parr);
4127                                 SEGOP_INCORE(seg, saddr, naddr - saddr, parr);

4129                                 for (pagenum = 0; pagenum < npages; pagenum++) {
4130                                         if (parr[pagenum] & SEG_PAGE_INCORE)
4131                                                 mp->pr_rss++;
4132                                         if (parr[pagenum] & SEG_PAGE_ANON)
4133                                                 mp->pr_anon++;
4134                                         if (parr[pagenum] & SEG_PAGE_LOCKED)
4135                                                 mp->pr_locked++;
4136                                 }
4137                                 kmem_free(parr, npages);
4138                         }
4139                 }
4140                 ASSERT(tmp == NULL);
4141         } while ((seg = AS_SEGNEXT(as, seg)) != NULL);

4143         return (0);
4144 }
```

*_____unchanged_portion_omitted_*

```
4180 #ifdef _SYSCALL32_IMPL
4181 /*
4182  * Return an array of structures with HAT memory map information.
4183  * We allocate here; the caller must deallocate.
4184  */
```

```
4185 int
4186 prgetxmap32(proc_t *p, list_t *iolhead)
4187 {
4188         struct as *as = p->p_as;
4189         prxmap32_t *mp;
4190         struct seg *seg;
4191         struct seg *brkseg, *stkseg;
4192         struct vnode *vp;
4193         struct vattr vattr;
4194         uint_t prot;

4196         ASSERT(as != &kas && AS_WRITE_HELD(as, &as->a_lock));

4198         /*
4199          * Request an initial buffer size that doesn't waste memory
4200          * if the address space has only a small number of segments.
4201          */
4202         pr_iol_initlist(iolhead, sizeof (*mp), avl_numnodes(&as->a_segtree));

4204         if ((seg = AS_SEGFIRST(as)) == NULL)
4205                 return (0);

4207         brkseg = break_seg(p);
4208         stkseg = as_segat(as, prgetstackbase(p));

4210         do {
4211                 caddr_t eaddr = seg->s_base + pr_getsegsize(seg, 0);
4212                 caddr_t saddr, naddr, baddr;
4213                 void *tmp = NULL;
4214                 ssize_t psz;
4215                 char *parr;
4216                 uint64_t npages;
4217                 uint64_t pagenum;

4219                 /*
4220                  * Segment loop part one: iterate from the base of the segment
4221                  * to its end, pausing at each address boundary (baddr) between
4222                  * ranges that have different virtual memory protections.
4223                  */
4224                 for (saddr = seg->s_base; saddr < eaddr; saddr = baddr) {
4225                         prot = pr_getprot(seg, 0, &tmp, &saddr, &baddr, eaddr);
4226                         ASSERT(baddr >= saddr && baddr <= eaddr);

4228                         /*
4229                          * Segment loop part two: iterate from the current
4230                          * position to the end of the protection boundary,
4231                          * pausing at each address boundary (naddr) between
4232                          * ranges that have different underlying page sizes.
4233                          */
4234                         for (; saddr < baddr; saddr = naddr) {
4235                                 psz = pr_getpagesize(seg, saddr, &naddr, baddr);
4236                                 ASSERT(naddr >= saddr && naddr <= baddr);

4238                                 mp = pr_iol_newbuf(iolhead, sizeof (*mp));

4240                                 mp->pr_vaddr = (caddr32_t)(uintptr_t)saddr;
4241                                 mp->pr_size = (size32_t)(naddr - saddr);
4242                                 mp->pr_offset = segop_getoffset(seg, saddr);
4242                                 mp->pr_offset = SEGOP_GETOFFSET(seg, saddr);
4243                                 mp->pr_mflags = 0;
4244                                 if (prot & PROT_READ)
4245                                         mp->pr_mflags |= MA_READ;
4246                                 if (prot & PROT_WRITE)
4247                                         mp->pr_mflags |= MA_WRITE;
4248                                 if (prot & PROT_EXEC)
4249                                         mp->pr_mflags |= MA_EXEC;
```



```
4250                                 if (segop_gettype(seg, saddr) & MAP_SHARED)
4250                                 if (SEGOP_GETTYPE(seg, saddr) & MAP_SHARED)
4251                                         mp->pr_mflags |= MA_SHARED;
4252                                 if (segop_gettype(seg, saddr) & MAP_NORESERVE)
4252                                 if (SEGOP_GETTYPE(seg, saddr) & MAP_NORESERVE)
4253                                         mp->pr_mflags |= MA_NORESERVE;
4254                                 if (seg->s_ops == &segspt_shmops ||
4255                                     (seg->s_ops == &segvn_ops &&
4256                                         (segop_getvp(seg, saddr, &vp) != 0 ||
4256                                         (SEGOP_GETVP(seg, saddr, &vp) != 0 ||
4257                                         vp == NULL)))
4258                                         mp->pr_mflags |= MA_ANON;
4259                                 if (seg == brkseg)
4260                                         mp->pr_mflags |= MA_BREAK;
4261                                 else if (seg == stkseg)
4262                                         mp->pr_mflags |= MA_STACK;
4263                                 if (seg->s_ops == &segspt_shmops)
4264                                         mp->pr_mflags |= MA_ISM | MA_SHM;

4266                                 mp->pr_pagesize = PAGESIZE;
4267                                 if (psz == -1) {
4268                                         mp->pr_hatpagesize = 0;
4269                                 } else {
4270                                         mp->pr_hatpagesize = psz;
4271                                 }

4273                                 /*
4274                                  * Manufacture a filename for the "object" dir.
4275                                  */
4276                                 mp->pr_dev = PRNODEV32;
4277                                 vattr.va_mask = AT_FSID|AT_NODEID;
4278                                 if (seg->s_ops == &segvn_ops &&
4279                                     segop_getvp(seg, saddr, &vp) == 0 &&
4279                                     SEGOP_GETVP(seg, saddr, &vp) == 0 &&
4280                                     vp != NULL && vp->v_type == VREG &&
4281                                     VOP_GETATTR(vp, &vattr, 0, CRED(),
4282                                     NULL) == 0) {
4283                                         (void) cmpldev(&mp->pr_dev,
4284                                             vattr.va_fsid);
4285                                         mp->pr_ino = vattr.va_nodeid;
4286                                         if (vp == p->p_exec)
4287                                                 (void) strcpy(mp->pr_mapname,
4288                                                     "a.out");
4289                                         else
4290                                                 pr_object_name(mp->pr_mapname,
4291                                                     vp, &vattr);
4292                                 }

4294                                 /*
4295                                  * Get the SysV shared memory id, if any.
4296                                  */
4297                                 if ((mp->pr_mflags & MA_SHARED) &&
4298                                     p->p_segacct && (mp->pr_shmid = shmgetid(p,
4299                                     seg->s_base)) != SHMID_NONE) {
4300                                         if (mp->pr_shmid == SHMID_FREE)
4301                                                 mp->pr_shmid = -1;

4303                                         mp->pr_mflags |= MA_SHM;
4304                                 } else {
4305                                         mp->pr_shmid = -1;
4306                                 }

4308                                 npages = ((uintptr_t)(naddr - saddr)) >>
4309                                     PAGESHIFT;
4310                                 parr = kmem_zalloc(npages, KM_SLEEP);
```

```
4312                                    segop_incore(seg, saddr, naddr - saddr, parr);
4312                                    SEGOP_INCORE(seg, saddr, naddr - saddr, parr);

4314                                    for (pagenum = 0; pagenum < npages; pagenum++) {
4315                                            if (parr[pagenum] & SEG_PAGE_INCORE)
4316                                                    mp->pr_rss++;
4317                                            if (parr[pagenum] & SEG_PAGE_ANON)
4318                                                    mp->pr_anon++;
4319                                            if (parr[pagenum] & SEG_PAGE_LOCKED)
4320                                                    mp->pr_locked++;
4321                                    }
4322                                    kmem_free(parr, npages);
4323                            }
4324                    }
4325                    ASSERT(tmp == NULL);
4326            } while ((seg = AS_SEGNEXT(as, seg)) != NULL);

4328            return (0);
4329 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
  142900 Fri May  8 18:04:05 2015
new/usr/src/uts/common/fs/proc/prvnops.c
patch lower-case-segops
**********************************************************
______unchanged_portion_omitted_

3663 static vnode_t *
3664 pr_lookup_objectdir(vnode_t *dp, char *comp)
3665 {
3666         prnode_t *dpnp = VTOP(dp);
3667         prnode_t *pnp;
3668         proc_t *p;
3669         struct seg *seg;
3670         struct as *as;
3671         vnode_t *vp;
3672         vattr_t vattr;

3674         ASSERT(dpnp->pr_type == PR_OBJECTDIR);

3676         pnp = prgetnode(dp, PR_OBJECT);

3678         if (prlock(dpnp, ZNO) != 0) {
3679                 prfreenode(pnp);
3680                 return (NULL);
3681         }
3682         p = dpnp->pr_common->prc_proc;
3683         if ((p->p_flag & SSYS) || (as = p->p_as) == &kas) {
3684                 prunlock(dpnp);
3685                 prfreenode(pnp);
3686                 return (NULL);
3687         }

3689         /*
3690          * We drop p_lock before grabbing the address space lock
3691          * in order to avoid a deadlock with the clock thread.
3692          * The process will not disappear and its address space
3693          * will not change because it is marked P_PR_LOCK.
3694          */
3695         mutex_exit(&p->p_lock);
3696         AS_LOCK_ENTER(as, &as->a_lock, RW_READER);
3697         if ((seg = AS_SEGFIRST(as)) == NULL) {
3698                 vp = NULL;
3699                 goto out;
3700         }
3701         if (strcmp(comp, "a.out") == 0) {
3702                 vp = p->p_exec;
3703                 goto out;
3704         }
3705         do {
3706                 /*
3707                  * Manufacture a filename for the "object" directory.
3708                  */
3709                 vattr.va_mask = AT_FSID|AT_NODEID;
3710                 if (seg->s_ops == &segvn_ops &&
3711                     segop_getvp(seg, seg->s_base, &vp) == 0 &&
3711                     SEGOP_GETVP(seg, seg->s_base, &vp) == 0 &&
3712                     vp != NULL && vp->v_type == VREG &&
3713                     VOP_GETATTR(vp, &vattr, 0, CRED(), NULL) == 0) {
3714                         char name[64];

3716                         if (vp == p->p_exec)    /* "a.out" */
3717                                 continue;
3718                         pr_object_name(name, vp, &vattr);
3719                         if (strcmp(name, comp) == 0)
3720                                 goto out;
```



```
3721                 }
3722         } while ((seg = AS_SEGNEXT(as, seg)) != NULL);

3724         vp = NULL;
3725 out:
3726         if (vp != NULL) {
3727                 VN_HOLD(vp);
3728         }
3729         AS_LOCK_EXIT(as, &as->a_lock);
3730         mutex_enter(&p->p_lock);
3731         prunlock(dpnp);

3733         if (vp == NULL)
3734                 prfreenode(pnp);
3735         else {
3736                 /*
3737                  * Fill in the prnode so future references will
3738                  * be able to find the underlying object's vnode.
3739                  * Don't link this prnode into the list of all
3740                  * prnodes for the process; this is a one-use node.
3741                  * Its use is entirely to catch and fail opens for writing.
3742                  */
3743                 pnp->pr_realvp = vp;
3744                 vp = PTOV(pnp);
3745         }

3747         return (vp);
3748 }
______unchanged_portion_omitted_

4051 static vnode_t *
4052 pr_lookup_pathdir(vnode_t *dp, char *comp)
4053 {
4054         prnode_t *dpnp = VTOP(dp);
4055         prnode_t *pnp;
4056         vnode_t *vp = NULL;
4057         proc_t *p;
4058         uint_t fd, flags = 0;
4059         int c;
4060         uf_entry_t *ufp;
4061         uf_info_t *fip;
4062         enum { NAME_FD, NAME_OBJECT, NAME_ROOT, NAME_CWD, NAME_UNKNOWN } type;
4063         char *tmp;
4064         int idx;
4065         struct seg *seg;
4066         struct as *as = NULL;
4067         vattr_t vattr;

4069         ASSERT(dpnp->pr_type == PR_PATHDIR);

4071         /*
4072          * First, check if this is a numeric entry, in which case we have a
4073          * file descriptor.
4074          */
4075         fd = 0;
4076         type = NAME_FD;
4077         tmp = comp;
4078         while ((c = *tmp++) != '\0') {
4079                 int ofd;
4080                 if (c < '0' || c > '9') {
4081                         type = NAME_UNKNOWN;
4082                         break;
4083                 }
4084                 ofd = fd;
4085                 fd = 10*fd + c - '0';
4086                 if (fd/10 != ofd) {     /* integer overflow */
```

```
4087                        type = NAME_UNKNOWN;
4088                        break;
4089                }
4090        }

4092        /*
4093         * Next, see if it is one of the special values {root, cwd}.
4094         */
4095        if (type == NAME_UNKNOWN) {
4096                if (strcmp(comp, "root") == 0)
4097                        type = NAME_ROOT;
4098                else if (strcmp(comp, "cwd") == 0)
4099                        type = NAME_CWD;
4100        }

4102        /*
4103         * Grab the necessary data from the process
4104         */
4105        if (prlock(dpnp, ZNO) != 0)
4106                return (NULL);
4107        p = dpnp->pr_common->prc_proc;

4109        fip = P_FINFO(p);

4111        switch (type) {
4112        case NAME_ROOT:
4113                if ((vp = PTOU(p)->u_rdir) == NULL)
4114                        vp = p->p_zone->zone_rootvp;
4115                VN_HOLD(vp);
4116                break;
4117        case NAME_CWD:
4118                vp = PTOU(p)->u_cdir;
4119                VN_HOLD(vp);
4120                break;
4121        default:
4122                if ((p->p_flag & SSYS) || (as = p->p_as) == &kas) {
4123                        prunlock(dpnp);
4124                        return (NULL);
4125                }
4126        }
4127        mutex_exit(&p->p_lock);

4129        /*
4130         * Determine if this is an object entry
4131         */
4132        if (type == NAME_UNKNOWN) {
4133                /*
4134                 * Start with the inode index immediately after the number of
4135                 * files.
4136                 */
4137                mutex_enter(&fip->fi_lock);
4138                idx = fip->fi_nfiles + 4;
4139                mutex_exit(&fip->fi_lock);

4141                if (strcmp(comp, "a.out") == 0) {
4142                        if (p->p_execdir != NULL) {
4143                                vp = p->p_execdir;
4144                                VN_HOLD(vp);
4145                                type = NAME_OBJECT;
4146                                flags |= PR_AOUT;
4147                        } else {
4148                                vp = p->p_exec;
4149                                VN_HOLD(vp);
4150                                type = NAME_OBJECT;
4151                        }
4152                } else {
```



```
4153                        AS_LOCK_ENTER(as, &as->a_lock, RW_READER);
4154                        if ((seg = AS_SEGFIRST(as)) != NULL) {
4155                                do {
4156                                        /*
4157                                         * Manufacture a filename for the
4158                                         * "object" directory.
4159                                         */
4160                                        vattr.va_mask = AT_FSID|AT_NODEID;
4161                                        if (seg->s_ops == &segvn_ops &&
4162                                            segop_getvp(seg, seg->s_base, &vp)
4162                                            SEGOP_GETVP(seg, seg->s_base, &vp)
4163                                            == 0 &&
4164                                            vp != NULL && vp->v_type == VREG &&
4165                                            VOP_GETATTR(vp, &vattr, 0, CRED(),
4166                                            NULL) == 0) {
4167                                                char name[64];

4169                                                if (vp == p->p_exec)
4170                                                        continue;
4171                                                idx++;
4172                                                pr_object_name(name, vp,
4173                                                    &vattr);
4174                                                if (strcmp(name, comp) == 0)
4175                                                        break;
4176                                        }
4177                                } while ((seg = AS_SEGNEXT(as, seg)) != NULL);
4178                        }

4180                        if (seg == NULL) {
4181                                vp = NULL;
4182                        } else {
4183                                VN_HOLD(vp);
4184                                type = NAME_OBJECT;
4185                        }

4187                        AS_LOCK_EXIT(as, &as->a_lock);
4188                }
4189        }


4192        switch (type) {
4193        case NAME_FD:
4194                mutex_enter(&fip->fi_lock);
4195                if (fd < fip->fi_nfiles) {
4196                        UF_ENTER(ufp, fip, fd);
4197                        if (ufp->uf_file != NULL) {
4198                                vp = ufp->uf_file->f_vnode;
4199                                VN_HOLD(vp);
4200                        }
4201                        UF_EXIT(ufp);
4202                }
4203                mutex_exit(&fip->fi_lock);
4204                idx = fd + 4;
4205                break;
4206        case NAME_ROOT:
4207                idx = 2;
4208                break;
4209        case NAME_CWD:
4210                idx = 3;
4211                break;
4212        case NAME_OBJECT:
4213        case NAME_UNKNOWN:
4214                /* Nothing to do */
4215                break;
4216        }
```

```
4218         mutex_enter(&p->p_lock);
4219         prunlock(dpnp);

4221         if (vp != NULL) {
4222                 pnp = prgetnode(dp, PR_PATH);

4224                 pnp->pr_flags |= flags;
4225                 pnp->pr_common = dpnp->pr_common;
4226                 pnp->pr_pcommon = dpnp->pr_pcommon;
4227                 pnp->pr_realvp = vp;
4228                 pnp->pr_parent = dp;            /* needed for prlookup */
4229                 pnp->pr_ino = pmkino(idx, dpnp->pr_common->prc_slot, PR_PATH);
4230                 VN_HOLD(dp);
4231                 vp = PTOV(pnp);
4232                 vp->v_type = VLNK;
4233         }

4235         return (vp);
4236 }
_____unchanged_portion_omitted_

4829 static void
4830 rebuild_objdir(struct as *as)
4831 {
4832         struct seg *seg;
4833         vnode_t *vp;
4834         vattr_t vattr;
4835         vnode_t **dir;
4836         ulong_t nalloc;
4837         ulong_t nentries;
4838         int i, j;
4839         ulong_t nold, nnew;

4841         ASSERT(AS_WRITE_HELD(as, &as->a_lock));

4843         if (as->a_updatedir == 0 && as->a_objectdir != NULL)
4844                 return;
4845         as->a_updatedir = 0;

4847         if ((nalloc = avl_numnodes(&as->a_segtree)) == 0 ||
4848             (seg = AS_SEGFIRST(as)) == NULL)    /* can't happen? */
4849                 return;

4851         /*
4852          * Allocate space for the new object directory.
4853          * (This is usually about two times too many entries.)
4854          */
4855         nalloc = (nalloc + 0xf) & ~0xf;         /* multiple of 16 */
4856         dir = kmem_zalloc(nalloc * sizeof (vnode_t *), KM_SLEEP);

4858         /* fill in the new directory with desired entries */
4859         nentries = 0;
4860         do {
4861                 vattr.va_mask = AT_FSID|AT_NODEID;
4862                 if (seg->s_ops == &segvn_ops &&
4863                     segop_getvp(seg, seg->s_base, &vp) == 0 &&
4863                     SEGOP_GETVP(seg, seg->s_base, &vp) == 0 &&
4864                     vp != NULL && vp->v_type == VREG &&
4865                     VOP_GETATTR(vp, &vattr, 0, CRED(), NULL) == 0) {
4866                         for (i = 0; i < nentries; i++)
4867                                 if (vp == dir[i])
4868                                         break;
4869                         if (i == nentries) {
4870                                 ASSERT(nentries < nalloc);
4871                                 dir[nentries++] = vp;
4872                         }
```



```
4873                 }
4874         } while ((seg = AS_SEGNEXT(as, seg)) != NULL);

4876         if (as->a_objectdir == NULL) {  /* first time */
4877                 as->a_objectdir = dir;
4878                 as->a_sizedir = nalloc;
4879                 return;
4880         }

4882         /*
4883          * Null out all of the defunct entries in the old directory.
4884          */
4885         nold = 0;
4886         nnew = nentries;
4887         for (i = 0; i < as->a_sizedir; i++) {
4888                 if ((vp = as->a_objectdir[i]) != NULL) {
4889                         for (j = 0; j < nentries; j++) {
4890                                 if (vp == dir[j]) {
4891                                         dir[j] = NULL;
4892                                         nnew--;
4893                                         break;
4894                                 }
4895                         }
4896                         if (j == nentries)
4897                                 as->a_objectdir[i] = NULL;
4898                         else
4899                                 nold++;
4900                 }
4901         }

4903         if (nold + nnew > as->a_sizedir) {
4904                 /*
4905                  * Reallocate the old directory to have enough
4906                  * space for the old and new entries combined.
4907                  * Round up to the next multiple of 16.
4908                  */
4909                 ulong_t newsize = (nold + nnew + 0xf) & ~0xf;
4910                 vnode_t **newdir = kmem_zalloc(newsize * sizeof (vnode_t *),
4911                     KM_SLEEP);
4912                 bcopy(as->a_objectdir, newdir,
4913                     as->a_sizedir * sizeof (vnode_t *));
4914                 kmem_free(as->a_objectdir, as->a_sizedir * sizeof (vnode_t *));
4915                 as->a_objectdir = newdir;
4916                 as->a_sizedir = newsize;
4917         }

4919         /*
4920          * Move all new entries to the old directory and
4921          * deallocate the space used by the new directory.
4922          */
4923         if (nnew) {
4924                 for (i = 0, j = 0; i < nentries; i++) {
4925                         if ((vp = dir[i]) == NULL)
4926                                 continue;
4927                         for (; j < as->a_sizedir; j++) {
4928                                 if (as->a_objectdir[j] != NULL)
4929                                         continue;
4930                                 as->a_objectdir[j++] = vp;
4931                                 break;
4932                         }
4933                 }
4934         }
4935         kmem_free(dir, nalloc * sizeof (vnode_t *));
4936 }
______unchanged_portion_omitted_
```

```
*********************************************************
   23668 Fri May  8 18:04:05 2015
new/usr/src/uts/common/io/mem.c
patch lower-case-segops
*********************************************************
______unchanged_portion_omitted_

 285 static int
 286 mmpagelock(struct as *as, caddr_t va)
 287 {
 288         struct seg *seg;
 289         int i;

 291         AS_LOCK_ENTER(as, &as->a_lock, RW_READER);
 292         seg = as_segat(as, va);
 293         i = (seg != NULL)? segop_capable(seg, S_CAPABILITY_NOMINFLT) : 0;
 293         i = (seg != NULL)? SEGOP_CAPABLE(seg, S_CAPABILITY_NOMINFLT) : 0;
 294         AS_LOCK_EXIT(as, &as->a_lock);

 296         return (i);
 297 }
______unchanged_portion_omitted_
```

```
**********************************************************
   35777 Fri May  8 18:04:05 2015
new/usr/src/uts/common/os/dumpsubr.c
patch lower-case-segops
**********************************************************
______unchanged_portion_omitted_

 661 /*
 662  * Dump the <as, va, pfn> information for a given address space.
 663  * segop_dump() will call dump_addpage() for each page in the segment.
 663  * SEGOP_DUMP() will call dump_addpage() for each page in the segment.
 664  */
 665 static void
 666 dump_as(struct as *as)
 667 {
 668         struct seg *seg;

 670         AS_LOCK_ENTER(as, &as->a_lock, RW_READER);
 671         for (seg = AS_SEGFIRST(as); seg; seg = AS_SEGNEXT(as, seg)) {
 672                 if (seg->s_as != as)
 673                         break;
 674                 if (seg->s_ops == NULL)
 675                         continue;
 676                 segop_dump(seg);
 676                 SEGOP_DUMP(seg);
 677         }
 678         AS_LOCK_EXIT(as, &as->a_lock);

 680         if (seg != NULL)
 681                 cmn_err(CE_WARN, "invalid segment %p in address space %p",
 682                     (void *)seg, (void *)as);
 683 }
______unchanged_portion_omitted_
```

```
**********************************************************
   52326 Fri May  8 18:04:05 2015
new/usr/src/uts/common/os/exec.c
patch lower-case-segops
**********************************************************
______unchanged_portion_omitted_

1144 /*
1145  * Map a section of an executable file into the user's
1146  * address space.
1147  */
1148 int
1149 execmap(struct vnode *vp, caddr_t addr, size_t len, size_t zfodlen,
1150     off_t offset, int prot, int page, uint_t szc)
1151 {
1152         int error = 0;
1153         off_t oldoffset;
1154         caddr_t zfodbase, oldaddr;
1155         size_t end, oldlen;
1156         size_t zfoddiff;
1157         label_t ljb;
1158         proc_t *p = ttoproc(curthread);

1160         oldaddr = addr;
1161         addr = (caddr_t)((uintptr_t)addr & (uintptr_t)PAGEMASK);
1162         if (len) {
1163                 oldlen = len;
1164                 len += ((size_t)oldaddr - (size_t)addr);
1165                 oldoffset = offset;
1166                 offset = (off_t)((uintptr_t)offset & PAGEMASK);
1167                 if (page) {
1168                         spgcnt_t  prefltmem, availm, npages;
1169                         int preread;
1170                         uint_t mflag = MAP_PRIVATE | MAP_FIXED;

1172                         if ((prot & (PROT_WRITE | PROT_EXEC)) == PROT_EXEC) {
1173                                 mflag |= MAP_TEXT;
1174                         } else {
1175                                 mflag |= MAP_INITDATA;
1176                         }

1178                         if (valid_usr_range(addr, len, prot, p->p_as,
1179                             p->p_as->a_userlimit) != RANGE_OKAY) {
1180                                 error = ENOMEM;
1181                                 goto bad;
1182                         }
1183                         if (error = VOP_MAP(vp, (offset_t)offset,
1184                             p->p_as, &addr, len, prot, PROT_ALL,
1185                             mflag, CRED(), NULL))
1186                                 goto bad;

1188                         /*
1189                          * If the segment can fit, then we prefault
1190                          * the entire segment in.  This is based on the
1191                          * model that says the best working set of a
1192                          * small program is all of its pages.
1193                          */
1194                         npages = (spgcnt_t)btopr(len);
1195                         prefltmem = freemem - desfree;
1196                         preread =
1197                             (npages < prefltmem && len < PGTHRESH) ? 1 : 0;

1199                         /*
1200                          * If we aren't prefaulting the segment,
1201                          * increment "deficit", if necessary to ensure
1202                          * that pages will become available when this
```



```
1203                          * process starts executing.
1204                          */
1205                         availm = freemem - lotsfree;
1206                         if (preread == 0 && npages > availm &&
1207                             deficit < lotsfree) {
1208                                 deficit += MIN((pgcnt_t)(npages - availm),
1209                                     lotsfree - deficit);
1210                         }

1212                         if (preread) {
1213                                 TRACE_2(TR_FAC_PROC, TR_EXECMAP_PREREAD,
1214                                     "execmap preread:freemem %d size %lu",
1215                                     freemem, len);
1216                                 (void) as_fault(p->p_as->a_hat, p->p_as,
1217                                     (caddr_t)addr, len, F_INVAL, S_READ);
1218                         }
1219                 } else {
1220                         if (valid_usr_range(addr, len, prot, p->p_as,
1221                             p->p_as->a_userlimit) != RANGE_OKAY) {
1222                                 error = ENOMEM;
1223                                 goto bad;
1224                         }

1226                         if (error = as_map(p->p_as, addr, len,
1227                             segvn_create, zfod_argsp))
1228                                 goto bad;
1229                         /*
1230                          * Read in the segment in one big chunk.
1231                          */
1232                         if (error = vn_rdwr(UIO_READ, vp, (caddr_t)oldaddr,
1233                             oldlen, (offset_t)oldoffset, UIO_USERSPACE, 0,
1234                             (rlim64_t)0, CRED(), (ssize_t *)0))
1235                                 goto bad;
1236                         /*
1237                          * Now set protections.
1238                          */
1239                         if (prot != PROT_ZFOD) {
1240                                 (void) as_setprot(p->p_as, (caddr_t)addr,
1241                                     len, prot);
1242                         }
1243                 }
1244         }

1246         if (zfodlen) {
1247                 struct as *as = curproc->p_as;
1248                 struct seg *seg;
1249                 uint_t zprot = 0;

1251                 end = (size_t)addr + len;
1252                 zfodbase = (caddr_t)roundup(end, PAGESIZE);
1253                 zfoddiff = (uintptr_t)zfodbase - end;
1254                 if (zfoddiff) {
1255                         /*
1256                          * Before we go to zero the remaining space on the last
1257                          * page, make sure we have write permission.
1258                          *
1259                          * Normal illumos binaries don't even hit the case
1260                          * where we have to change permission on the last page
1261                          * since their protection is typically either
1262                          *    PROT_USER | PROT_WRITE | PROT_READ
1263                          * or
1264                          *    PROT_ZFOD (same as PROT_ALL).
1265                          *
1266                          * We need to be careful how we zero-fill the last page
1267                          * if the segment protection does not include
1268                          * PROT_WRITE. Using as_setprot() can cause the VM
```

```
1269                         * segment code to call segvn_vpage(), which must
1270                         * allocate a page struct for each page in the segment.
1271                         * If we have a very large segment, this may fail, so
1272                         * we have to check for that, even though we ignore
1273                         * other return values from as_setprot.
1274                         */
1276                        AS_LOCK_ENTER(as, &as->a_lock, RW_READER);
1277                        seg = as_segat(curproc->p_as, (caddr_t)end);
1278                        if (seg != NULL)
1279                                segop_getprot(seg, (caddr_t)end, zfoddiff - 1,
1279                                SEGOP_GETPROT(seg, (caddr_t)end, zfoddiff - 1,
1280                                    &zprot);
1281                        AS_LOCK_EXIT(as, &as->a_lock);
1283                        if (seg != NULL && (zprot & PROT_WRITE) == 0) {
1284                                if (as_setprot(as, (caddr_t)end, zfoddiff - 1,
1285                                    zprot | PROT_WRITE) == ENOMEM) {
1286                                        error = ENOMEM;
1287                                        goto bad;
1288                                }
1289                        }
1291                        if (on_fault(&ljb)) {
1292                                no_fault();
1293                                if (seg != NULL && (zprot & PROT_WRITE) == 0)
1294                                        (void) as_setprot(as, (caddr_t)end,
1295                                            zfoddiff - 1, zprot);
1296                                error = EFAULT;
1297                                goto bad;
1298                        }
1299                        uzero((void *)end, zfoddiff);
1300                        no_fault();
1301                        if (seg != NULL && (zprot & PROT_WRITE) == 0)
1302                                (void) as_setprot(as, (caddr_t)end,
1303                                    zfoddiff - 1, zprot);
1304                }
1305                if (zfodlen > zfoddiff) {
1306                        struct segvn_crargs crargs =
1307                            SEGVN_ZFOD_ARGS(PROT_ZFOD, PROT_ALL);
1309                        zfodlen -= zfoddiff;
1310                        if (valid_usr_range(zfodbase, zfodlen, prot, p->p_as,
1311                            p->p_as->a_userlimit) != RANGE_OKAY) {
1312                                error = ENOMEM;
1313                                goto bad;
1314                        }
1315                        if (szc > 0) {
1316                                /*
1317                                 * ASSERT alignment because the mapelfexec()
1318                                 * caller for the szc > 0 case extended zfod
1319                                 * so it's end is pgsz aligned.
1320                                 */
1321                                size_t pgsz = page_get_pagesize(szc);
1322                                ASSERT(IS_P2ALIGNED(zfodbase + zfodlen, pgsz));
1324                                if (IS_P2ALIGNED(zfodbase, pgsz)) {
1325                                        crargs.szc = szc;
1326                                } else {
1327                                        crargs.szc = AS_MAP_HEAP;
1328                                }
1329                        } else {
1330                                crargs.szc = AS_MAP_NO_LPOOB;
1331                        }
1332                        if (error = as_map(p->p_as, (caddr_t)zfodbase,
1333                            zfodlen, segvn_create, &crargs))
```



```
1334                                        goto bad;
1335                                if (prot != PROT_ZFOD) {
1336                                        (void) as_setprot(p->p_as, (caddr_t)zfodbase,
1337                                            zfodlen, prot);
1338                                }
1339                        }
1340                }
1341                return (0);
1342 bad:
1343                return (error);
1344 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
  119448 Fri May  8 18:04:06 2015
new/usr/src/uts/common/os/lgrp.c
patch lower-case-segops
**********************************************************
______unchanged_portion_omitted_

3498 /*
3499  * Get memory allocation policy for this segment
3500  */
3501 lgrp_mem_policy_info_t *
3502 lgrp_mem_policy_get(struct seg *seg, caddr_t vaddr)
3503 {
3504         lgrp_mem_policy_info_t  *policy_info;
3505         extern struct seg_ops   segspt_ops;
3506         extern struct seg_ops   segspt_shmops;

3508         /*
3509          * This is for binary compatibility to protect against third party
3510          * segment drivers which haven't recompiled to allow for
3511          * segop_getpolicy()
3511          * SEGOP_GETPOLICY()
3512          */
3513         if (seg->s_ops != &segvn_ops && seg->s_ops != &segspt_ops &&
3514             seg->s_ops != &segspt_shmops)
3515                 return (NULL);

3517         policy_info = NULL;
3518         if (seg->s_ops->getpolicy != NULL)
3519                 policy_info = segop_getpolicy(seg, vaddr);
3519                 policy_info = SEGOP_GETPOLICY(seg, vaddr);

3521         return (policy_info);
3522 }
______unchanged_portion_omitted_
```

```
**********************************************************
   69060 Fri May  8 18:04:06 2015
new/usr/src/uts/common/os/mmapobj.c
patch lower-case-segops
**********************************************************
______unchanged_portion_omitted_

1434 /*
1435  * Check the address space to see if the virtual addresses to be used are
1436  * available.  If they are not, return errno for failure.  On success, 0
1437  * will be returned, and the virtual addresses for each mmapobj_result_t
1438  * will be reserved.  Note that a reservation could have earlier been made
1439  * for a given segment via a /dev/null mapping.  If that is the case, then
1440  * we can use that VA space for our mappings.
1441  * Note: this function will only be used for ET_EXEC binaries.
1442  */
1443 int
1444 check_exec_addrs(int loadable, mmapobj_result_t *mrp, caddr_t start_addr)
1445 {
1446         int i;
1447         struct as *as = curproc->p_as;
1448         struct segvn_crargs crargs = SEGVN_ZFOD_ARGS(PROT_ZFOD, PROT_ALL);
1449         int ret;
1450         caddr_t myaddr;
1451         size_t mylen;
1452         struct seg *seg;

1454         /* No need to reserve swap space now since it will be reserved later */
1455         crargs.flags |= MAP_NORESERVE;
1456         as_rangelock(as);
1457         for (i = 0; i < loadable; i++) {

1459                 myaddr = start_addr + (size_t)mrp[i].mr_addr;
1460                 mylen = mrp[i].mr_msize;

1462                 /* See if there is a hole in the as for this range */
1463                 if (as_gap(as, mylen, &myaddr, &mylen, 0, NULL) == 0) {
1464                         ASSERT(myaddr == start_addr + (size_t)mrp[i].mr_addr);
1465                         ASSERT(mylen == mrp[i].mr_msize);

1467 #ifdef DEBUG
1468                         if (MR_GET_TYPE(mrp[i].mr_flags) == MR_PADDING) {
1469                                 MOBJ_STAT_ADD(exec_padding);
1470                         }
1471 #endif
1472                         ret = as_map(as, myaddr, mylen, segvn_create, &crargs);
1473                         if (ret) {
1474                                 as_rangeunlock(as);
1475                                 mmapobj_unmap_exec(mrp, i, start_addr);
1476                                 return (ret);
1477                         }
1478                 } else {
1479                         /*
1480                          * There is a mapping that exists in the range
1481                          * so check to see if it was a "reservation"
1482                          * from /dev/null.  The mapping is from
1483                          * /dev/null if the mapping comes from
1484                          * segdev and the type is neither MAP_SHARED
1485                          * nor MAP_PRIVATE.
1486                          */
1487                         AS_LOCK_ENTER(as, &as->a_lock, RW_READER);
1488                         seg = as_findseg(as, myaddr, 0);
1489                         MOBJ_STAT_ADD(exec_addr_mapped);
1490                         if (seg && seg->s_ops == &segdev_ops &&
1491                             ((segop_gettype(seg, myaddr) &
1491                             ((SEGOP_GETTYPE(seg, myaddr) &
```



```
1492                             (MAP_SHARED | MAP_PRIVATE)) == 0) &&
1493                             myaddr >= seg->s_base &&
1494                             myaddr + mylen <=
1495                             seg->s_base + seg->s_size) {
1496                                 MOBJ_STAT_ADD(exec_addr_devnull);
1497                                 AS_LOCK_EXIT(as, &as->a_lock);
1498                                 (void) as_unmap(as, myaddr, mylen);
1499                                 ret = as_map(as, myaddr, mylen, segvn_create,
1500                                     &crargs);
1501                                 mrp[i].mr_flags |= MR_RESV;
1502                                 if (ret) {
1503                                         as_rangeunlock(as);
1504                                         /* Need to remap what we unmapped */
1505                                         mmapobj_unmap_exec(mrp, i + 1,
1506                                             start_addr);
1507                                         return (ret);
1508                                 }
1509                         } else {
1510                                 AS_LOCK_EXIT(as, &as->a_lock);
1511                                 as_rangeunlock(as);
1512                                 mmapobj_unmap_exec(mrp, i, start_addr);
1513                                 MOBJ_STAT_ADD(exec_addr_in_use);
1514                                 return (EADDRINUSE);
1515                         }
1516                 }
1517         }
1518         as_rangeunlock(as);
1519         return (0);
1520 }
______unchanged_portion_omitted_
```

```
**********************************************************
  248841 Fri May  8 18:04:06 2015
new/usr/src/uts/common/os/sunddi.c
patch lower-case-segops
**********************************************************
______unchanged_portion_omitted_

8219 /*
8220  * A consolidation private function which is essentially equivalent to
8221  * ddi_umem_lock but with the addition of arguments ops_vector and procp.
8222  * A call to as_add_callback is done if DDI_UMEMLOCK_LONGTERM is set, and
8223  * the ops_vector is valid.
8224  *
8225  * Lock the virtual address range in the current process and create a
8226  * ddi_umem_cookie (of type UMEM_LOCKED). This can be used to pass to
8227  * ddi_umem_iosetup to create a buf or do devmap_umem_setup/remap to export
8228  * to user space.
8229  *
8230  * Note: The resource control accounting currently uses a full charge model
8231  * in other words attempts to lock the same/overlapping areas of memory
8232  * will deduct the full size of the buffer from the projects running
8233  * counter for the device locked memory.
8234  *
8235  * addr, size should be PAGESIZE aligned
8236  *
8237  * flags - DDI_UMEMLOCK_READ, DDI_UMEMLOCK_WRITE or both
8238  *      identifies whether the locked memory will be read or written or both
8239  *      DDI_UMEMLOCK_LONGTERM  must be set when the locking will
8240  * be maintained for an indefinitely long period (essentially permanent),
8241  * rather than for what would be required for a typical I/O completion.
8242  * When DDI_UMEMLOCK_LONGTERM is set, umem_lockmemory will return EFAULT
8243  * if the memory pertains to a regular file which is mapped MAP_SHARED.
8244  * This is to prevent a deadlock if a file truncation is attempted after
8245  * after the locking is done.
8246  *
8247  * Returns 0 on success
8248  *      EINVAL - for invalid parameters
8249  *      EPERM, ENOMEM and other error codes returned by as_pagelock
8250  *      ENOMEM - is returned if the current request to lock memory exceeds
8251  *               *.max-locked-memory resource control value.
8252  *      EFAULT - memory pertains to a regular file mapped shared and
8253  *               and DDI_UMEMLOCK_LONGTERM flag is set
8254  *      EAGAIN - could not start the ddi_umem_unlock list processing thread
8255  */
8256 int
8257 umem_lockmemory(caddr_t addr, size_t len, int flags, ddi_umem_cookie_t *cookie,
8258                 struct umem_callback_ops *ops_vector,
8259                 proc_t *procp)
8260 {
8261         int     error;
8262         struct ddi_umem_cookie *p;
8263         void    (*driver_callback)() = NULL;
8264         struct as *as;
8265         struct seg              *seg;
8266         vnode_t                 *vp;

8268         /* Allow device drivers to not have to reference "curproc" */
8269         if (procp == NULL)
8270                 procp = curproc;
8271         as = procp->p_as;
8272         *cookie = NULL;         /* in case of any error return */

8274         /* These are the only three valid flags */
8275         if ((flags & ~(DDI_UMEMLOCK_READ | DDI_UMEMLOCK_WRITE |
8276             DDI_UMEMLOCK_LONGTERM)) != 0)
8277                 return (EINVAL);
```



```
8279         /* At least one (can be both) of the two access flags must be set */
8280         if ((flags & (DDI_UMEMLOCK_READ | DDI_UMEMLOCK_WRITE)) == 0)
8281                 return (EINVAL);

8283         /* addr and len must be page-aligned */
8284         if (((uintptr_t)addr & PAGEOFFSET) != 0)
8285                 return (EINVAL);

8287         if ((len & PAGEOFFSET) != 0)
8288                 return (EINVAL);

8290         /*
8291          * For longterm locking a driver callback must be specified; if
8292          * not longterm then a callback is optional.
8293          */
8294         if (ops_vector != NULL) {
8295                 if (ops_vector->cbo_umem_callback_version !=
8296                     UMEM_CALLBACK_VERSION)
8297                         return (EINVAL);
8298                 else
8299                         driver_callback = ops_vector->cbo_umem_lock_cleanup;
8300         }
8301         if ((driver_callback == NULL) && (flags & DDI_UMEMLOCK_LONGTERM))
8302                 return (EINVAL);

8304         /*
8305          * Call i_ddi_umem_unlock_thread_start if necessary.  It will
8306          * be called on first ddi_umem_lock or umem_lockmemory call.
8307          */
8308         if (ddi_umem_unlock_thread == NULL)
8309                 i_ddi_umem_unlock_thread_start();

8311         /* Allocate memory for the cookie */
8312         p = kmem_zalloc(sizeof (struct ddi_umem_cookie), KM_SLEEP);

8314         /* Convert the flags to seg_rw type */
8315         if (flags & DDI_UMEMLOCK_WRITE) {
8316                 p->s_flags = S_WRITE;
8317         } else {
8318                 p->s_flags = S_READ;
8319         }

8321         /* Store procp in cookie for later iosetup/unlock */
8322         p->procp = (void *)procp;

8324         /*
8325          * Store the struct as pointer in cookie for later use by
8326          * ddi_umem_unlock.  The proc->p_as will be stale if ddi_umem_unlock
8327          * is called after relvm is called.
8328          */
8329         p->asp = as;

8331         /*
8332          * The size field is needed for lockmem accounting.
8333          */
8334         p->size = len;
8335         init_lockedmem_rctl_flag(p);

8337         if (umem_incr_devlockmem(p) != 0) {
8338                 /*
8339                  * The requested memory cannot be locked
8340                  */
8341                 kmem_free(p, sizeof (struct ddi_umem_cookie));
8342                 *cookie = (ddi_umem_cookie_t)NULL;
8343                 return (ENOMEM);
```

```
8344         }

8346         /* Lock the pages corresponding to addr, len in memory */
8347         error = as_pagelock(as, &(p->pparray), addr, len, p->s_flags);
8348         if (error != 0) {
8349                 umem_decr_devlockmem(p);
8350                 kmem_free(p, sizeof (struct ddi_umem_cookie));
8351                 *cookie = (ddi_umem_cookie_t)NULL;
8352                 return (error);
8353         }

8355         /*
8356          * For longterm locking the addr must pertain to a seg_vn segment or
8357          * or a seg_spt segment.
8358          * If the segment pertains to a regular file, it cannot be
8359          * mapped MAP_SHARED.
8360          * This is to prevent a deadlock if a file truncation is attempted
8361          * after the locking is done.
8362          * Doing this after as_pagelock guarantees persistence of the as; if
8363          * an unacceptable segment is found, the cleanup includes calling
8364          * as_pageunlock before returning EFAULT.
8365          *
8366          * segdev is allowed here as it is already locked.  This allows
8367          * for memory exported by drivers through mmap() (which is already
8368          * locked) to be allowed for LONGTERM.
8369          */
8370         if (flags & DDI_UMEMLOCK_LONGTERM) {
8371                 extern  struct seg_ops segspt_shmops;
8372                 extern  struct seg_ops segdev_ops;
8373                 AS_LOCK_ENTER(as, &as->a_lock, RW_READER);
8374                 for (seg = as_segat(as, addr); ; seg = AS_SEGNEXT(as, seg)) {
8375                         if (seg == NULL || seg->s_base > addr + len)
8376                                 break;
8377                         if (seg->s_ops == &segdev_ops)
8378                                 continue;
8379                         if (((seg->s_ops != &segvn_ops) &&
8380                             (seg->s_ops != &segspt_shmops)) ||
8381                             ((segop_getvp(seg, addr, &vp) == 0 &&
8381                             ((SEGOP_GETVP(seg, addr, &vp) == 0 &&
8382                             vp != NULL && vp->v_type == VREG) &&
8383                             (segop_gettype(seg, addr) & MAP_SHARED))) {
8383                             (SEGOP_GETTYPE(seg, addr) & MAP_SHARED))) {
8384                                 as_pageunlock(as, p->pparray,
8385                                     addr, len, p->s_flags);
8386                                 AS_LOCK_EXIT(as, &as->a_lock);
8387                                 umem_decr_devlockmem(p);
8388                                 kmem_free(p, sizeof (struct ddi_umem_cookie));
8389                                 *cookie = (ddi_umem_cookie_t)NULL;
8390                                 return (EFAULT);
8391                         }
8392                 }
8393                 AS_LOCK_EXIT(as, &as->a_lock);
8394         }


8397         /* Initialize the fields in the ddi_umem_cookie */
8398         p->cvaddr = addr;
8399         p->type = UMEM_LOCKED;
8400         if (driver_callback != NULL) {
8401                 /* i_ddi_umem_unlock and umem_lock_undo may need the cookie */
8402                 p->cook_refcnt = 2;
8403                 p->callbacks = *ops_vector;
8404         } else {
8405                 /* only i_ddi_umme_unlock needs the cookie */
8406                 p->cook_refcnt = 1;
8407         }
```



```
8409         *cookie = (ddi_umem_cookie_t)p;

8411         /*
8412          * If a driver callback was specified, add an entry to the
8413          * as struct callback list. The as_pagelock above guarantees
8414          * the persistence of as.
8415          */
8416         if (driver_callback) {
8417                 error = as_add_callback(as, umem_lock_undo, p, AS_ALL_EVENT,
8418                     addr, len, KM_SLEEP);
8419                 if (error != 0) {
8420                         as_pageunlock(as, p->pparray,
8421                             addr, len, p->s_flags);
8422                         umem_decr_devlockmem(p);
8423                         kmem_free(p, sizeof (struct ddi_umem_cookie));
8424                         *cookie = (ddi_umem_cookie_t)NULL;
8425                 }
8426         }
8427         return (error);
8428 }
```

*_____unchanged_portion_omitted_*

```
**********************************************************
    8569 Fri May  8 18:04:07 2015
new/usr/src/uts/common/os/urw.c
patch lower-case-segops
**********************************************************
```

```
/*
 * CDDL HEADER START
 *
 * The contents of this file are subject to the terms of the
 * Common Development and Distribution License (the "License").
 * You may not use this file except in compliance with the License.
 *
 * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
 * or http://www.opensolaris.org/os/licensing.
 * See the License for the specific language governing permissions
 * and limitations under the License.
 *
 * When distributing Covered Code, include this CDDL HEADER in each
 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
 * If applicable, add the following below this CDDL HEADER, with the
 * fields enclosed by brackets "[]" replaced with your own identifying
 * information: Portions Copyright [yyyy] [name of copyright owner]
 *
 * CDDL HEADER END
 */
/*
 * Copyright 2007 Sun Microsystems, Inc.  All rights reserved.
 * Use is subject to license terms.
 */

/*      Copyright (c) 1984, 1986, 1987, 1988, 1989 AT&T */
/*        All Rights Reserved   */

#pragma ident   "%Z%%M% %I%     %E% SMI"

#include <sys/atomic.h>
#include <sys/errno.h>
#include <sys/stat.h>
#include <sys/modctl.h>
#include <sys/conf.h>
#include <sys/systm.h>
#include <sys/ddi.h>
#include <sys/sunddi.h>
#include <sys/cpuvar.h>
#include <sys/kmem.h>
#include <sys/strsubr.h>
#include <sys/sysmacros.h>
#include <sys/frame.h>
#include <sys/stack.h>
#include <sys/proc.h>
#include <sys/priv.h>
#include <sys/policy.h>
#include <sys/ontrap.h>
#include <sys/vmsystm.h>
#include <sys/prsystm.h>

#include <vm/as.h>
#include <vm/seg.h>
#include <vm/seg_dev.h>
#include <vm/seg_vn.h>
#include <vm/seg_spt.h>
#include <vm/seg_kmem.h>

extern struct seg_ops segdev_ops;       /* needs a header file */
extern struct seg_ops segspt_shmops;    /* needs a header file */
```



```
static int
page_valid(struct seg *seg, caddr_t addr)
{
        struct segvn_data *svd;
        vnode_t *vp;
        vattr_t vattr;

        /*
         * Fail if the page doesn't map to a page in the underlying
         * mapped file, if an underlying mapped file exists.
         */
        vattr.va_mask = AT_SIZE;
        if (seg->s_ops == &segvn_ops &&
            segop_getvp(seg, addr, &vp) == 0 &&
            SEGOP_GETVP(seg, addr, &vp) == 0 &&
            vp != NULL && vp->v_type == VREG &&
            VOP_GETATTR(vp, &vattr, 0, CRED(), NULL) == 0) {
                u_offset_t size = roundup(vattr.va_size, (u_offset_t)PAGESIZE);
                u_offset_t offset = segop_getoffset(seg, addr);
                u_offset_t offset = SEGOP_GETOFFSET(seg, addr);

                if (offset >= size)
                        return (0);
        }

        /*
         * Fail if this is an ISM shared segment and the address is
         * not within the real size of the spt segment that backs it.
         */
        if (seg->s_ops == &segspt_shmops &&
            addr >= seg->s_base + spt_realsize(seg))
                return (0);

        /*
         * Fail if the segment is mapped from /dev/null.
         * The key is that the mapping comes from segdev and the
         * type is neither MAP_SHARED nor MAP_PRIVATE.
         */
        if (seg->s_ops == &segdev_ops &&
            ((segop_gettype(seg, addr) & (MAP_SHARED | MAP_PRIVATE)) == 0))
            ((SEGOP_GETTYPE(seg, addr) & (MAP_SHARED | MAP_PRIVATE)) == 0))
                return (0);

        /*
         * Fail if the page is a MAP_NORESERVE page that has
         * not actually materialized.
         * We cheat by knowing that segvn is the only segment
         * driver that supports MAP_NORESERVE.
         */
        if (seg->s_ops == &segvn_ops &&
            (svd = (struct segvn_data *)seg->s_data) != NULL &&
            (svd->vp == NULL || svd->vp->v_type != VREG) &&
            (svd->flags & MAP_NORESERVE)) {
                /*
                 * Guilty knowledge here.  We know that
                 * segvn_incore returns more than just the
                 * low-order bit that indicates the page is
                 * actually in memory.  If any bits are set,
                 * then there is backing store for the page.
                 */
                char incore = 0;
                (void) segop_incore(seg, addr, PAGESIZE, &incore);
                (void) SEGOP_INCORE(seg, addr, PAGESIZE, &incore);
                if (incore == 0)
                        return (0);
        }
```

```
124         return (1);
125 }
_____unchanged_portion_omitted_

178 /*
179  * Perform I/O to a given process. This will return EIO if we detect
180  * corrupt memory and ENXIO if there is no such mapped address in the
181  * user process's address space.
182  */
183 static int
184 urw(proc_t *p, int writing, void *buf, size_t len, uintptr_t a)
185 {
186         caddr_t addr = (caddr_t)a;
187         caddr_t page;
188         caddr_t vaddr;
189         struct seg *seg;
190         int error = 0;
191         int err = 0;
192         uint_t prot;
193         uint_t prot_rw = writing ? PROT_WRITE : PROT_READ;
194         int protchanged;
195         on_trap_data_t otd;
196         int retrycnt;
197         struct as *as = p->p_as;
198         enum seg_rw rw;

200         /*
201          * Locate segment containing address of interest.
202          */
203         page = (caddr_t)(uintptr_t)((uintptr_t)addr & PAGEMASK);
204         retrycnt = 0;
205         AS_LOCK_ENTER(as, &as->a_lock, RW_WRITER);
206 retry:
207         if ((seg = as_segat(as, page)) == NULL ||
208             !page_valid(seg, page)) {
209                 AS_LOCK_EXIT(as, &as->a_lock);
210                 return (ENXIO);
211         }
212         segop_getprot(seg, page, 0, &prot);
212         SEGOP_GETPROT(seg, page, 0, &prot);

214         protchanged = 0;
215         if ((prot & prot_rw) == 0) {
216                 protchanged = 1;
217                 err = segop_setprot(seg, page, PAGESIZE, prot | prot_rw);
217                 err = SEGOP_SETPROT(seg, page, PAGESIZE, prot | prot_rw);

219                 if (err == IE_RETRY) {
220                         protchanged = 0;
221                         ASSERT(retrycnt == 0);
222                         retrycnt++;
223                         goto retry;
224                 }

226                 if (err != 0) {
227                         AS_LOCK_EXIT(as, &as->a_lock);
228                         return (ENXIO);
229                 }
230         }

232         /*
233          * segvn may do a copy-on-write for F_SOFTLOCK/S_READ case to break
234          * sharing to avoid a copy on write of a softlocked page by another
235          * thread. But since we locked the address space as a writer no other
236          * thread can cause a copy on write. S_READ_NOCOW is passed as the
237          * access type to tell segvn that it's ok not to do a copy-on-write
```



```
238          * for this SOFTLOCK fault.
239          */
240         if (writing)
241                 rw = S_WRITE;
242         else if (seg->s_ops == &segvn_ops)
243                 rw = S_READ_NOCOW;
244         else
245                 rw = S_READ;

247         if (segop_fault(as->a_hat, seg, page, PAGESIZE, F_SOFTLOCK, rw)) {
247         if (SEGOP_FAULT(as->a_hat, seg, page, PAGESIZE, F_SOFTLOCK, rw)) {
248                 if (protchanged)
249                         (void) segop_setprot(seg, page, PAGESIZE, prot);
249                         (void) SEGOP_SETPROT(seg, page, PAGESIZE, prot);
250                 AS_LOCK_EXIT(as, &as->a_lock);
251                 return (ENXIO);
252         }
253         CPU_STATS_ADD_K(vm, softlock, 1);

255         /*
256          * Make sure we're not trying to read or write off the end of the page.
257          */
258         ASSERT(len <= page + PAGESIZE - addr);

260         /*
261          * Map in the locked page, copy to our local buffer,
262          * then map the page out and unlock it.
263          */
264         vaddr = mapin(as, addr, writing);

266         /*
267          * Since we are copying memory on behalf of the user process,
268          * protect against memory error correction faults.
269          */
270         if (!on_trap(&otd, OT_DATA_EC)) {
271                 if (seg->s_ops == &segdev_ops) {
272                         /*
273                          * Device memory can behave strangely; invoke
274                          * a segdev-specific copy operation instead.
275                          */
276                         if (writing) {
277                                 if (segdev_copyto(seg, addr, buf, vaddr, len))
278                                         error = ENXIO;
279                         } else {
280                                 if (segdev_copyfrom(seg, addr, vaddr, buf, len))
281                                         error = ENXIO;
282                         }
283                 } else {
284                         if (writing)
285                                 bcopy(buf, vaddr, len);
286                         else
287                                 bcopy(vaddr, buf, len);
288                 }
289         } else {
290                 error = EIO;
291         }
292         no_trap();

294         /*
295          * If we're writing to an executable page, we may need to sychronize
296          * the I$ with the modifications we made through the D$.
297          */
298         if (writing && (prot & PROT_EXEC))
299                 sync_icache(vaddr, (uint_t)len);

301         mapout(as, addr, vaddr, writing);
```

```
303         if (rw == S_READ_NOCOW)
304                 rw = S_READ;

306         (void) segop_fault(as->a_hat, seg, page, PAGESIZE, F_SOFTUNLOCK, rw);
306         (void) SEGOP_FAULT(as->a_hat, seg, page, PAGESIZE, F_SOFTUNLOCK, rw);

308         if (protchanged)
309                 (void) segop_setprot(seg, page, PAGESIZE, prot);
309                 (void) SEGOP_SETPROT(seg, page, PAGESIZE, prot);

311         AS_LOCK_EXIT(as, &as->a_lock);

313         return (error);
314 }
```

unchanged_portion_omitted_

```
**********************************************************
   14034 Fri May  8 18:04:07 2015
new/usr/src/uts/common/os/vm_subr.c
patch lower-case-segops
**********************************************************
______unchanged_portion_omitted_

 318 #define MAX_MAPIN_PAGES 8

 320 /*
 321  * This function temporarily "borrows" user pages for kernel use. If
 322  * "cow" is on, it also sets up copy-on-write protection (only feasible
 323  * on MAP_PRIVATE segment) on the user mappings, to protect the borrowed
 324  * pages from any changes by the user. The caller is responsible for
 325  * unlocking and tearing down cow settings when it's done with the pages.
 326  * For an example, see kcfree().
 327  *
 328  * Pages behind [uaddr..uaddr+*lenp] under address space "as" are locked
 329  * (shared), and mapped into kernel address range [kaddr..kaddr+*lenp] if
 330  * kaddr != -1. On entering this function, cached_ppp contains a list
 331  * of pages that are mapped into [kaddr..kaddr+*lenp] already (from a
 332  * previous call). Thus if same pages remain behind [uaddr..uaddr+*lenp],
 333  * the kernel map won't need to be reloaded again.
 334  *
 335  * For cow == 1, if the pages are anonymous pages, it also bumps the anon
 336  * reference count, and change the user-mapping to read-only. This
 337  * scheme should work on all types of segment drivers. But to be safe,
 338  * we check against segvn here.
 339  *
 340  * Since this function is used to emulate copyin() semantic, it checks
 341  * to make sure the user-mappings allow "user-read".
 342  *
 343  * On exit "lenp" contains the number of bytes successfully locked and
 344  * mapped in. For the unsuccessful ones, the caller can fall back to
 345  * copyin().
 346  *
 347  * Error return:
 348  * ENOTSUP - operation like this is not supported either on this segment
 349  * type, or on this platform type.
 350  */
 351 int
 352 cow_mapin(struct as *as, caddr_t uaddr, caddr_t kaddr, struct page **cached_ppp,
 353     struct anon **app, size_t *lenp, int cow)
 354 {
 355         struct          hat *hat;
 356         struct seg      *seg;
 357         caddr_t         base;
 358         page_t          *pp, *ppp[MAX_MAPIN_PAGES];
 359         long            i;
 360         int             flags;
 361         size_t          size, total = *lenp;
 362         char            first = 1;
 363         faultcode_t     res;

 365         *lenp = 0;
 366         if (cow) {
 367                 AS_LOCK_ENTER(as, &as->a_lock, RW_WRITER);
 368                 seg = as_findseg(as, uaddr, 0);
 369                 if ((seg == NULL) || ((base = seg->s_base) > uaddr) ||
 370                     (uaddr + total) > base + seg->s_size) {
 371                         AS_LOCK_EXIT(as, &as->a_lock);
 372                         return (EINVAL);
 373                 }
 374                 /*
 375                  * The COW scheme should work for all segment types.
 376                  * But to be safe, we check against segvn.
```



```
 377                  */
 378                 if (seg->s_ops != &segvn_ops) {
 379                         AS_LOCK_EXIT(as, &as->a_lock);
 380                         return (ENOTSUP);
 381                 } else if ((segop_gettype(seg, uaddr) & MAP_PRIVATE) == 0) {
 381                 } else if ((SEGOP_GETTYPE(seg, uaddr) & MAP_PRIVATE) == 0) {
 382                         AS_LOCK_EXIT(as, &as->a_lock);
 383                         return (ENOTSUP);
 384                 }
 385         }
 386         hat = as->a_hat;
 387         size = total;
 388 tryagain:
 389         /*
 390          * If (cow), hat_softlock will also change the usr protection to RO.
 391          * This is the first step toward setting up cow. Before we
 392          * bump up an_refcnt, we can't allow any cow-fault on this
 393          * address. Otherwise segvn_fault will change the protection back
 394          * to RW upon seeing an_refcnt == 1.
 395          * The solution is to hold the writer lock on "as".
 396          */
 397         res = hat_softlock(hat, uaddr, &size, &ppp[0], cow ? HAT_COW : 0);
 398         size = total - size;
 399         *lenp += size;
 400         size = size >> PAGESHIFT;
 401         i = 0;
 402         while (i < size) {
 403                 pp = ppp[i];
 404                 if (cow) {
 405                         kmutex_t *ahm;
 406                         /*
 407                          * Another solution is to hold SE_EXCL on pp, and
 408                          * disable PROT_WRITE. This also works for MAP_SHARED
 409                          * segment. The disadvantage is that it locks the
 410                          * page from being used by anybody else.
 411                          */
 412                         ahm = AH_MUTEX(pp->p_vnode, pp->p_offset);
 413                         mutex_enter(ahm);
 414                         *app = swap_anon(pp->p_vnode, pp->p_offset);
 415                         /*
 416                          * Since we are holding the as lock, this avoids a
 417                          * potential race with anon_decref. (segvn_unmap and
 418                          * segvn_free needs the as writer lock to do anon_free.)
 419                          */
 420                         if (*app != NULL) {
 421 #if 0
 422                                 if ((*app)->an_refcnt == 0)
 423                                 /*
 424                                  * Consider the following senario (unlikey
 425                                  * though):
 426                                  * 1. an_refcnt == 2
 427                                  * 2. we solftlock the page.
 428                                  * 3. cow ocurrs on this addr. So a new ap,
 429                                  * page and mapping is established on addr.
 430                                  * 4. an_refcnt drops to 1 (segvn_faultpage
 431                                  * -> anon_decref(oldap))
 432                                  * 5. the last ref to ap also drops (from
 433                                  * another as). It ends up blocked inside
 434                                  * anon_decref trying to get page's excl lock.
 435                                  * 6. Later kcfree unlocks the page, call
 436                                  * anon_decref -> oops, ap is gone already.
 437                                  *
 438                                  * Holding as writer lock solves all problems.
 439                                  */
 440                                         *app = NULL;
 441                                 else
```

```
#endif
                                        (*app)->an_refcnt++;
                        }
                        mutex_exit(ahm);
                } else {
                        *app = NULL;
                }
                if (kaddr != (caddr_t)-1) {
                        if (pp != *cached_ppp) {
                                if (*cached_ppp == NULL)
                                        flags = HAT_LOAD_LOCK | HAT_NOSYNC |
                                            HAT_LOAD_NOCONSIST;
                                else
                                        flags = HAT_LOAD_REMAP |
                                            HAT_LOAD_NOCONSIST;
                                /*
                                 * In order to cache the kernel mapping after
                                 * the user page is unlocked, we call
                                 * hat_devload instead of hat_memload so
                                 * that the kernel mapping we set up here is
                                 * "invisible" to the rest of the world. This
                                 * is not very pretty. But as long as the
                                 * caller bears the responsibility of keeping
                                 * cache consistency, we should be ok -
                                 * HAT_NOCONSIST will get us a uncached
                                 * mapping on VAC. hat_softlock will flush
                                 * a VAC_WRITEBACK cache. Therefore the kaddr
                                 * doesn't have to be of the same vcolor as
                                 * uaddr.
                                 * The alternative is - change hat_devload
                                 * to get a cached mapping. Allocate a kaddr
                                 * with the same vcolor as uaddr. Then
                                 * hat_softlock won't need to flush the VAC.
                                 */
                                hat_devload(kas.a_hat, kaddr, PAGESIZE,
                                    page_pptonum(pp), PROT_READ, flags);
                                *cached_ppp = pp;
                        }
                        kaddr += PAGESIZE;
                }
                cached_ppp++;
                app++;
                ++i;
        }
        if (cow) {
                AS_LOCK_EXIT(as, &as->a_lock);
        }
        if (first && res == FC_NOMAP) {
                /*
                 * If the address is not mapped yet, we call as_fault to
                 * fault the pages in. We could've fallen back to copy and
                 * let it fault in the pages. But for a mapped file, we
                 * normally reference each page only once. For zero-copy to
                 * be of any use, we'd better fall in the page now and try
                 * again.
                 */
                first = 0;
                size = size << PAGESHIFT;
                uaddr += size;
                total -= size;
                size = total;
                res = as_fault(as->a_hat, as, uaddr, size, F_INVAL, S_READ);
                if (cow)
                        AS_LOCK_ENTER(as, &as->a_lock, RW_WRITER);
                goto tryagain;
        }
```



```
        switch (res) {
        case FC_NOSUPPORT:
                return (ENOTSUP);
        case FC_PROT:   /* Pretend we don't know about it. This will be */
                        /* caught by the caller when uiomove fails. */
        case FC_NOMAP:
        case FC_OBJERR:
        default:
                return (0);
        }
}
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   38530 Fri May  8 18:04:07 2015
new/usr/src/uts/common/os/watchpoint.c
patch lower-case-segops
**********************************************************
______unchanged_portion_omitted_

153 #define X       0
154 #define W       1
155 #define R       2
156 #define sum(a)  (a[X] + a[W] + a[R])

158 /*
159  * Common code for pr_mappage() and pr_unmappage().
160  */
161 static int
162 pr_do_mappage(caddr_t addr, size_t size, int mapin, enum seg_rw rw, int kernel)
163 {
164         proc_t *p = curproc;
165         struct as *as = p->p_as;
166         char *eaddr = addr + size;
167         int prot_rw = rw_to_prot(rw);
168         int xrw = rw_to_index(rw);
169         int rv = 0;
170         struct watched_page *pwp;
171         struct watched_page tpw;
172         avl_index_t where;
173         uint_t prot;

175         ASSERT(as != &kas);

177 startover:
178         ASSERT(rv == 0);
179         if (avl_numnodes(&as->a_wpage) == 0)
180                 return (0);

182         /*
183          * as->a_wpage can only be changed while the process is totally stopped.
184          * Don't grab p_lock here.  Holding p_lock while grabbing the address
185          * space lock leads to deadlocks with the clock thread.
186          *
187          * p_maplock prevents simultaneous execution of this function.  Under
188          * normal circumstances, holdwatch() will stop all other threads, so the
189          * lock isn't really needed.  But there may be multiple threads within
190          * stop() when SWATCHOK is set, so we need to handle multiple threads
191          * at once.  See holdwatch() for the details of this dance.
192          */

194         mutex_enter(&p->p_maplock);
195         AS_LOCK_ENTER(as, &as->a_lock, RW_READER);

197         tpw.wp_vaddr = (caddr_t)((uintptr_t)addr & (uintptr_t)PAGEMASK);
198         if ((pwp = avl_find(&as->a_wpage, &tpw, &where)) == NULL)
199                 pwp = avl_nearest(&as->a_wpage, where, AVL_AFTER);

201         for (; pwp != NULL && pwp->wp_vaddr < eaddr;
202                 pwp = AVL_NEXT(&as->a_wpage, pwp)) {

204                 /*
205                  * If the requested protection has not been
206                  * removed, we need not remap this page.
207                  */
208                 prot = pwp->wp_prot;
209                 if (kernel || (prot & PROT_USER))
210                         if (prot & prot_rw)
211                                 continue;
```



```
212                 /*
213                  * If the requested access does not exist in the page's
214                  * original protections, we need not remap this page.
215                  * If the page does not exist yet, we can't test it.
216                  */
217                 if ((prot = pwp->wp_oprot) != 0) {
218                         if (!(kernel || (prot & PROT_USER)))
219                                 continue;
220                         if (!(prot & prot_rw))
221                                 continue;
222                 }

224                 if (mapin) {
225                         /*
226                          * Before mapping the page in, ensure that
227                          * all other lwps are held in the kernel.
228                          */
229                         if (p->p_mapcnt == 0) {
230                                 /*
231                                  * Release as lock while in holdwatch()
232                                  * in case other threads need to grab it.
233                                  */
234                                 AS_LOCK_EXIT(as, &as->a_lock);
235                                 mutex_exit(&p->p_maplock);
236                                 if (holdwatch() != 0) {
237                                         /*
238                                          * We stopped in holdwatch().
239                                          * Start all over again because the
240                                          * watched page list may have changed.
241                                          */
242                                         goto startover;
243                                 }
244                                 mutex_enter(&p->p_maplock);
245                                 AS_LOCK_ENTER(as, &as->a_lock, RW_READER);
246                         }
247                         p->p_mapcnt++;
248                 }

250                 addr = pwp->wp_vaddr;
251                 rv++;

253                 prot = pwp->wp_prot;
254                 if (mapin) {
255                         if (kernel)
256                                 pwp->wp_kmap[xrw]++;
257                         else
258                                 pwp->wp_umap[xrw]++;
259                         pwp->wp_flags |= WP_NOWATCH;
260                         if (pwp->wp_kmap[X] + pwp->wp_umap[X])
261                                 /* cannot have exec-only protection */
262                                 prot |= PROT_READ|PROT_EXEC;
263                         if (pwp->wp_kmap[R] + pwp->wp_umap[R])
264                                 prot |= PROT_READ;
265                         if (pwp->wp_kmap[W] + pwp->wp_umap[W])
266                                 /* cannot have write-only protection */
267                                 prot |= PROT_READ|PROT_WRITE;
268 #if 0   /* damned broken mmu feature! */
269                         if (sum(pwp->wp_umap) == 0)
270                                 prot &= ~PROT_USER;
271 #endif
272                 } else {
273                         ASSERT(pwp->wp_flags & WP_NOWATCH);
274                         if (kernel) {
275                                 ASSERT(pwp->wp_kmap[xrw] != 0);
276                                 --pwp->wp_kmap[xrw];
277                         } else {
```

```
278                                 ASSERT(pwp->wp_umap[xrw] != 0);
279                                 --pwp->wp_umap[xrw];
280                         }
281                         if (sum(pwp->wp_kmap) + sum(pwp->wp_umap) == 0)
282                                 pwp->wp_flags &= ~WP_NOWATCH;
283                         else {
284                                 if (pwp->wp_kmap[X] + pwp->wp_umap[X])
285                                         /* cannot have exec-only protection */
286                                         prot |= PROT_READ|PROT_EXEC;
287                                 if (pwp->wp_kmap[R] + pwp->wp_umap[R])
288                                         prot |= PROT_READ;
289                                 if (pwp->wp_kmap[W] + pwp->wp_umap[W])
290                                         /* cannot have write-only protection */
291                                         prot |= PROT_READ|PROT_WRITE;
292 #if 0   /* damned broken mmu feature! */
293                                 if (sum(pwp->wp_umap) == 0)
294                                         prot &= ~PROT_USER;
295 #endif
296                         }
297                 }


300                 if (pwp->wp_oprot != 0) {       /* if page exists */
301                         struct seg *seg;
302                         uint_t oprot;
303                         int err, retrycnt = 0;

305                         AS_LOCK_EXIT(as, &as->a_lock);
306                         AS_LOCK_ENTER(as, &as->a_lock, RW_WRITER);
307                 retry:
308                         seg = as_segat(as, addr);
309                         ASSERT(seg != NULL);
310                         segop_getprot(seg, addr, 0, &oprot);
310                         SEGOP_GETPROT(seg, addr, 0, &oprot);
311                         if (prot != oprot) {
312                                 err = segop_setprot(seg, addr, PAGESIZE, prot);
312                                 err = SEGOP_SETPROT(seg, addr, PAGESIZE, prot);
313                                 if (err == IE_RETRY) {
314                                         ASSERT(retrycnt == 0);
315                                         retrycnt++;
316                                         goto retry;
317                                 }
318                         }
319                         AS_LOCK_EXIT(as, &as->a_lock);
320                 } else
321                         AS_LOCK_EXIT(as, &as->a_lock);

323                 /*
324                  * When all pages are mapped back to their normal state,
325                  * continue the other lwps.
326                  */
327                 if (!mapin) {
328                         ASSERT(p->p_mapcnt > 0);
329                         p->p_mapcnt--;
330                         if (p->p_mapcnt == 0) {
331                                 mutex_exit(&p->p_maplock);
332                                 mutex_enter(&p->p_lock);
333                                 continuelwps(p);
334                                 mutex_exit(&p->p_lock);
335                                 mutex_enter(&p->p_maplock);
336                         }
337                 }

339                 AS_LOCK_ENTER(as, &as->a_lock, RW_READER);
340         }
```



```
342         AS_LOCK_EXIT(as, &as->a_lock);
343         mutex_exit(&p->p_maplock);

345         return (rv);
346 }
______unchanged_portion_omitted_

374 /*
375  * Function called by an lwp after it resumes from stop().
376  */
377 void
378 setallwatch(void)
379 {
380         proc_t *p = curproc;
381         struct as *as = curproc->p_as;
382         struct watched_page *pwp, *next;
383         struct seg *seg;
384         caddr_t vaddr;
385         uint_t prot;
386         int err, retrycnt;

388         if (p->p_wprot == NULL)
389                 return;

391         ASSERT(MUTEX_NOT_HELD(&curproc->p_lock));

393         AS_LOCK_ENTER(as, &as->a_lock, RW_WRITER);

395         pwp = p->p_wprot;
396         while (pwp != NULL) {

398                 vaddr = pwp->wp_vaddr;
399                 retrycnt = 0;
400         retry:
401                 ASSERT(pwp->wp_flags & WP_SETPROT);
402                 if ((seg = as_segat(as, vaddr)) != NULL &&
403                     !(pwp->wp_flags & WP_NOWATCH)) {
404                         prot = pwp->wp_prot;
405                         err = segop_setprot(seg, vaddr, PAGESIZE, prot);
405                         err = SEGOP_SETPROT(seg, vaddr, PAGESIZE, prot);
406                         if (err == IE_RETRY) {
407                                 ASSERT(retrycnt == 0);
408                                 retrycnt++;
409                                 goto retry;
410                         }
411                 }

413                 next = pwp->wp_list;

415                 if (pwp->wp_read + pwp->wp_write + pwp->wp_exec == 0) {
416                         /*
417                          * No watched areas remain in this page.
418                          * Free the watched_page structure.
419                          */
420                         avl_remove(&as->a_wpage, pwp);
421                         kmem_free(pwp, sizeof (struct watched_page));
422                 } else {
423                         pwp->wp_flags &= ~WP_SETPROT;
424                 }

426                 pwp = next;
427         }
428         p->p_wprot = NULL;

430         AS_LOCK_EXIT(as, &as->a_lock);
431 }
______unchanged_portion_omitted_
```

****
193225 Fri May 8 18:04:08 2015
new/usr/src/uts/common/os/zone.c
patch lower-case-segops
****
_____unchanged_portion_omitted_

```
5593 /*
5594  * Return zero if the process has at least one vnode mapped in to its
5595  * address space which shouldn't be allowed to change zones.
5596  *
5597  * Also return zero if the process has any shared mappings which reserve
5598  * swap.  This is because the counting for zone.max-swap does not allow swap
5599  * reservation to be shared between zones.  zone swap reservation is counted
5600  * on zone->zone_max_swap.
5601  */
5602 static int
5603 as_can_change_zones(void)
5604 {
5605         proc_t *pp = curproc;
5606         struct seg *seg;
5607         struct as *as = pp->p_as;
5608         vnode_t *vp;
5609         int allow = 1;

5611         ASSERT(pp->p_as != &kas);
5612         AS_LOCK_ENTER(as, &as->a_lock, RW_READER);
5613         for (seg = AS_SEGFIRST(as); seg != NULL; seg = AS_SEGNEXT(as, seg)) {

5615                 /*
5616                  * Cannot enter zone with shared anon memory which
5617                  * reserves swap.  See comment above.
5618                  */
5619                 if (seg_can_change_zones(seg) == B_FALSE) {
5620                         allow = 0;
5621                         break;
5622                 }
5623                 /*
5624                  * if we can't get a backing vnode for this segment then skip
5625                  * it.
5626                  */
5627                 vp = NULL;
5628                 if (segop_getvp(seg, seg->s_base, &vp) != 0 || vp == NULL)
5628                 if (SEGOP_GETVP(seg, seg->s_base, &vp) != 0 || vp == NULL)
5629                         continue;
5630                 if (!vn_can_change_zones(vp)) { /* bail on first match */
5631                         allow = 0;
5632                         break;
5633                 }
5634         }
5635         AS_LOCK_EXIT(as, &as->a_lock);
5636         return (allow);
5637 }
```

_____unchanged_portion_omitted_

```
**********************************************************
    4234 Fri May  8 18:04:08 2015
new/usr/src/uts/common/sys/watchpoint.h
patch lower-case-segops
**********************************************************
______unchanged_portion_omitted_

  75 /* wp_flags */
  76 #define WP_NOWATCH      0x01    /* protections temporarily restored */
  77 #define WP_SETPROT      0x02    /* segop_setprot() needed on this page */

  79 #ifdef  _KERNEL

  81 /*
  82  * These functions handle the necessary logic to perform the copy operation
  83  * while ignoring watchpoints.
  84  */
  85 extern int copyin_nowatch(const void *, void *, size_t);
  86 extern int copyout_nowatch(const void *, void *, size_t);
  87 extern int fuword32_nowatch(const void *, uint32_t *);
  88 extern int suword32_nowatch(void *, uint32_t);
  89 #ifdef _LP64
  90 extern int suword64_nowatch(void *, uint64_t);
  91 extern int fuword64_nowatch(const void *, uint64_t *);
  92 #endif

  94 /*
  95  * Disable watchpoints for a given region of memory.  When bracketed by these
  96  * calls, functions can use copyops and ignore watchpoints.
  97  */
  98 extern int watch_disable_addr(const void *, size_t, enum seg_rw);
  99 extern void watch_enable_addr(const void *, size_t, enum seg_rw);

 101 /*
 102  * Enable/Disable watchpoints for an entire thread.
 103  */
 104 extern  void    watch_enable(kthread_id_t);
 105 extern  void    watch_disable(kthread_id_t);

 107 struct as;
 108 struct proc;
 109 struct k_siginfo;
 110 extern  void    setallwatch(void);
 111 extern  int     pr_is_watchpage(caddr_t, enum seg_rw);
 112 extern  int     pr_is_watchpage_as(caddr_t, enum seg_rw, struct as *);
 113 extern  int     pr_is_watchpoint(caddr_t *, int *, size_t, size_t *,
 114                         enum seg_rw);
 115 extern  void    do_watch_step(caddr_t, size_t, enum seg_rw, int, greg_t);
 116 extern  int     undo_watch_step(struct k_siginfo *);
 117 extern  int     wp_compare(const void *, const void *);
 118 extern  int     wa_compare(const void *, const void *);

 120 extern  struct copyops watch_copyops;

 122 extern watched_area_t *pr_find_watched_area(struct proc *, watched_area_t *,
 123     avl_index_t *);

 125 #endif

 127 #ifdef  __cplusplus
 128 }
______unchanged_portion_omitted_
```

```
**********************************************************
   23749 Fri May  8 18:04:08 2015
new/usr/src/uts/common/syscall/utssys.c
patch lower-case-segops
**********************************************************
______unchanged_portion_omitted_

313 static fu_data_t *
314 dofusers(vnode_t *fvp, int flags)
315 {
316         fu_data_t       *fu_data;
317         proc_t          *prp;
318         vfs_t           *cvfsp;
319         pid_t           npids, pidx, *pidlist;
320         int             v_proc = v.v_proc;      /* max # of procs */
321         int             pcnt = 0;
322         int             contained = (flags & F_CONTAINED);
323         int             nbmandonly = (flags & F_NBMANDLIST);
324         int             dip_usage = (flags & F_DEVINFO);
325         int             fvp_isdev = vn_matchops(fvp, spec_getvnodeops());
326         zone_t *zone = curproc->p_zone;
327         int inglobal = INGLOBALZONE(curproc);

329         /* get a pointer to the file system containing this vnode */
330         cvfsp = fvp->v_vfsp;
331         ASSERT(cvfsp);

333         /* allocate the data structure to return our results in */
334         fu_data = kmem_alloc(fu_data_size(v_proc), KM_SLEEP);
335         fu_data->fud_user_max = v_proc;
336         fu_data->fud_user_count = 0;

338         /* get a snapshot of all the pids we're going to check out */
339         pidlist = kmem_alloc(v_proc * sizeof (pid_t), KM_SLEEP);
340         mutex_enter(&pidlock);
341         for (npids = 0, prp = practive; prp != NULL; prp = prp->p_next) {
342                 if (inglobal || prp->p_zone == zone)
343                         pidlist[npids++] = prp->p_pid;
344         }
345         mutex_exit(&pidlock);

347         /* grab each process and check its file usage */
348         for (pidx = 0; pidx < npids; pidx++) {
349                 locklist_t      *llp = NULL;
350                 uf_info_t       *fip;
351                 vnode_t         *vp;
352                 user_t          *up;
353                 sess_t          *sp;
354                 uid_t           uid;
355                 pid_t           pid = pidlist[pidx];
356                 int             i, use_flag = 0;

358                 /*
359                  * grab prp->p_lock using sprlock()
360                  * if sprlock() fails the process does not exists anymore
361                  */
362                 prp = sprlock(pid);
363                 if (prp == NULL)
364                         continue;

366                 /* get the processes credential info in case we need it */
367                 mutex_enter(&prp->p_crlock);
368                 uid = crgetruid(prp->p_cred);
369                 mutex_exit(&prp->p_crlock);

371                 /*
```



```
372                  * it's safe to drop p_lock here because we
373                  * called sprlock() before and it set the SPRLOCK
374                  * flag for the process so it won't go away.
375                  */
376                 mutex_exit(&prp->p_lock);

378                 /*
379                  * now we want to walk a processes open file descriptors
380                  * to do this we need to grab the fip->fi_lock.  (you
381                  * can't hold p_lock when grabbing the fip->fi_lock.)
382                  */
383                 fip = P_FINFO(prp);
384                 mutex_enter(&fip->fi_lock);

386                 /*
387                  * Snapshot nbmand locks for pid
388                  */
389                 llp = flk_active_nbmand_locks(prp->p_pid);
390                 for (i = 0; i < fip->fi_nfiles; i++) {
391                         uf_entry_t      *ufp;
392                         file_t          *fp;

394                         UF_ENTER(ufp, fip, i);
395                         if (((fp = ufp->uf_file) == NULL) ||
396                             ((vp = fp->f_vnode) == NULL)) {
397                                 UF_EXIT(ufp);
398                                 continue;
399                         }

401                         /*
402                          * if the target file (fvp) is not a device
403                          * and corrosponds to the root of a filesystem
404                          * (cvfsp), then check if it contains the file
405                          * is use by this process (vp).
406                          */
407                         if (contained && (vp->v_vfsp == cvfsp))
408                                 use_flag |= F_OPEN;

410                         /*
411                          * if the target file (fvp) is not a device,
412                          * then check if it matches the file in use
413                          * by this process (vp).
414                          */
415                         if (!fvp_isdev && VN_CMP(fvp, vp))
416                                 use_flag |= F_OPEN;

418                         /*
419                          * if the target file (fvp) is a device,
420                          * then check if the current file in use
421                          * by this process (vp) maps to the same device
422                          * minor node.
423                          */
424                         if (fvp_isdev &&
425                             vn_matchops(vp, spec_getvnodeops()) &&
426                             (fvp->v_rdev == vp->v_rdev))
427                                 use_flag |= F_OPEN;

429                         /*
430                          * if the target file (fvp) is a device,
431                          * and we're checking for device instance
432                          * usage, then check if the current file in use
433                          * by this process (vp) maps to the same device
434                          * instance.
435                          */
436                         if (dip_usage &&
437                             vn_matchops(vp, spec_getvnodeops()) &&
```

```
438                             (VTOCS(fvp)->s_dip == VTOCS(vp)->s_dip))
439                                 use_flag |= F_OPEN;
441                         /*
442                          * if the current file in use by this process (vp)
443                          * doesn't match what we're looking for, move on
444                          * to the next file in the process.
445                          */
446                         if ((use_flag & F_OPEN) == 0) {
447                                 UF_EXIT(ufp);
448                                 continue;
449                         }
451                         if (proc_has_nbmand_on_vp(vp, prp->p_pid, llp)) {
452                                 /* A nbmand found so we're done.  */
453                                 use_flag |= F_NBM;
454                                 UF_EXIT(ufp);
455                                 break;
456                         }
457                         UF_EXIT(ufp);
458                 }
459                 if (llp)
460                         flk_free_locklist(llp);
462                 mutex_exit(&fip->fi_lock);
464                 /*
465                  * If nbmand usage tracking is desired and no nbmand was
466                  * found for this process, then no need to do further
467                  * usage tracking for this process.
468                  */
469                 if (nbmandonly && (!(use_flag & F_NBM))) {
470                         /*
471                          * grab the process lock again, clear the SPRLOCK
472                          * flag, release the process, and continue.
473                          */
474                         mutex_enter(&prp->p_lock);
475                         sprunlock(prp);
476                         continue;
477                 }
479                 /*
480                  * All other types of usage.
481                  * For the next few checks we need to hold p_lock.
482                  */
483                 mutex_enter(&prp->p_lock);
484                 up = PTOU(prp);
485                 if (fvp_isdev) {
486                         /*
487                          * if the target file (fvp) is a device
488                          * then check if it matches the processes tty
489                          *
490                          * we grab s_lock to protect ourselves against
491                          * freectty() freeing the vnode out from under us.
492                          */
493                         sp = prp->p_sessp;
494                         mutex_enter(&sp->s_lock);
495                         vp = prp->p_sessp->s_vp;
496                         if (vp != NULL) {
497                                 if (fvp->v_rdev == vp->v_rdev)
498                                         use_flag |= F_TTY;
500                                 if (dip_usage &&
501                                     (VTOCS(fvp)->s_dip == VTOCS(vp)->s_dip))
502                                         use_flag |= F_TTY;
503                         }
```



```
504                         mutex_exit(&sp->s_lock);
505                 } else {
506                         /* check the processes current working directory */
507                         if (up->u_cdir &&
508                             (VN_CMP(fvp, up->u_cdir) ||
509                             (contained && (up->u_cdir->v_vfsp == cvfsp))))
510                                 use_flag |= F_CDIR;
512                         /* check the processes root directory */
513                         if (up->u_rdir &&
514                             (VN_CMP(fvp, up->u_rdir) ||
515                             (contained && (up->u_rdir->v_vfsp == cvfsp))))
516                                 use_flag |= F_RDIR;
518                         /* check the program text vnode */
519                         if (prp->p_exec &&
520                             (VN_CMP(fvp, prp->p_exec) ||
521                             (contained && (prp->p_exec->v_vfsp == cvfsp))))
522                                 use_flag |= F_TEXT;
523                 }
525                 /* Now we can drop p_lock again */
526                 mutex_exit(&prp->p_lock);
528                 /*
529                  * now we want to walk a processes memory mappings.
530                  * to do this we need to grab the prp->p_as lock.  (you
531                  * can't hold p_lock when grabbing the prp->p_as lock.)
532                  */
533                 if (prp->p_as != &kas) {
534                         struct seg      *seg;
535                         struct as       *as = prp->p_as;
537                         AS_LOCK_ENTER(as, &as->a_lock, RW_READER);
538                         for (seg = AS_SEGFIRST(as); seg;
539                             seg = AS_SEGNEXT(as, seg)) {
540                                 /*
541                                  * if we can't get a backing vnode for this
542                                  * segment then skip it
543                                  */
544                                 vp = NULL;
545                                 if ((segop_getvp(seg, seg->s_base, &vp)) ||
545                                 if ((SEGOP_GETVP(seg, seg->s_base, &vp)) ||
546                                     (vp == NULL))
547                                         continue;
549                                 /*
550                                  * if the target file (fvp) is not a device
551                                  * and corrosponds to the root of a filesystem
552                                  * (cvfsp), then check if it contains the
553                                  * vnode backing this segment (vp).
554                                  */
555                                 if (contained && (vp->v_vfsp == cvfsp)) {
556                                         use_flag |= F_MAP;
557                                         break;
558                                 }
560                                 /*
561                                  * if the target file (fvp) is not a device,
562                                  * check if it matches the the vnode backing
563                                  * this segment (vp).
564                                  */
565                                 if (!fvp_isdev && VN_CMP(fvp, vp)) {
566                                         use_flag |= F_MAP;
567                                         break;
568                                 }
```

```
570                                 /*
571                                  * if the target file (fvp) isn't a device,
572                                  * or the the vnode backing this segment (vp)
573                                  * isn't a device then continue.
574                                  */
575                                 if (!fvp_isdev ||
576                                     !vn_matchops(vp, spec_getvnodeops()))
577                                         continue;

579                                 /*
580                                  * check if the vnode backing this segment
581                                  * (vp) maps to the same device minor node
582                                  * as the target device (fvp)
583                                  */
584                                 if (fvp->v_rdev == vp->v_rdev) {
585                                         use_flag |= F_MAP;
586                                         break;
587                                 }

589                                 /*
590                                  * if we're checking for device instance
591                                  * usage, then check if the vnode backing
592                                  * this segment (vp) maps to the same device
593                                  * instance as the target device (fvp).
594                                  */
595                                 if (dip_usage &&
596                                     (VTOCS(fvp)->s_dip == VTOCS(vp)->s_dip)) {
597                                         use_flag |= F_MAP;
598                                         break;
599                                 }
600                         }
601                         AS_LOCK_EXIT(as, &as->a_lock);
602                 }

604                 if (use_flag) {
605                         ASSERT(pcnt < fu_data->fud_user_max);
606                         fu_data->fud_user[pcnt].fu_flags = use_flag;
607                         fu_data->fud_user[pcnt].fu_pid = pid;
608                         fu_data->fud_user[pcnt].fu_uid = uid;
609                         pcnt++;
610                 }

612                 /*
613                  * grab the process lock again, clear the SPRLOCK
614                  * flag, release the process, and continue.
615                  */
616                 mutex_enter(&prp->p_lock);
617                 sprunlock(prp);
618         }

620         kmem_free(pidlist, v_proc * sizeof (pid_t));

622         fu_data->fud_user_count = pcnt;
623         return (fu_data);
624 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   10007 Fri May  8 18:04:09 2015
new/usr/src/uts/common/vm/seg.h
patch lower-case-segops
**********************************************************
______unchanged_portion_omitted_

 146 #ifdef _KERNEL

 148 /*
 149  * Generic segment operations
 150  */
 151 extern  void    seg_init(void);
 152 extern  struct  seg *seg_alloc(struct as *as, caddr_t base, size_t size);
 153 extern  int     seg_attach(struct as *as, caddr_t base, size_t size,
 154                         struct seg *seg);
 155 extern  void    seg_unmap(struct seg *seg);
 156 extern  void    seg_free(struct seg *seg);

 158 /*
 159  * functions for pagelock cache support
 160  */
 161 typedef int (*seg_preclaim_cbfunc_t)(void *, caddr_t, size_t,
 162     struct page **, enum seg_rw, int);

 164 extern  struct  page **seg_plookup(struct seg *seg, struct anon_map *amp,
 165     caddr_t addr, size_t len, enum seg_rw rw, uint_t flags);
 166 extern  void    seg_pinactive(struct seg *seg, struct anon_map *amp,
 167     caddr_t addr, size_t len, struct page **pp, enum seg_rw rw,
 168     uint_t flags, seg_preclaim_cbfunc_t callback);

 170 extern  void    seg_ppurge(struct seg *seg, struct anon_map *amp,
 171     uint_t flags);
 172 extern  void    seg_ppurge_wiredpp(struct page **pp);

 174 extern  int     seg_pinsert_check(struct seg *seg, struct anon_map *amp,
 175     caddr_t addr, size_t len, uint_t flags);
 176 extern  int     seg_pinsert(struct seg *seg, struct anon_map *amp,
 177     caddr_t addr, size_t len, size_t wlen, struct page **pp, enum seg_rw rw,
 178     uint_t flags, seg_preclaim_cbfunc_t callback);

 180 extern  void    seg_pasync_thread(void);
 181 extern  void    seg_preap(void);
 182 extern  int     seg_p_disable(void);
 183 extern  void    seg_p_enable(void);

 185 extern  segadvstat_t    segadvstat;

 187 /*
 188  * Flags for pagelock cache support.
 189  * Flags argument is passed as uint_t to pcache routines.  upper 16 bits of
 190  * the flags argument are reserved for alignment page shift when SEGP_PSHIFT
 191  * is set.
 192  */
 193 #define SEGP_FORCE_WIRED        0x1     /* skip check against seg_pwindow */
 194 #define SEGP_AMP                0x2     /* anon map's pcache entry */
 195 #define SEGP_PSHIFT             0x4     /* addr pgsz shift for hash function */

 197 /*
 198  * Return values for seg_pinsert and seg_pinsert_check functions.
 199  */
 200 #define SEGP_SUCCESS            0       /* seg_pinsert() succeeded */
 201 #define SEGP_FAIL               1       /* seg_pinsert() failed */

 203 /* Page status bits for segop_incore */
 204 #define SEG_PAGE_INCORE         0x01    /* VA has a page backing it */
```



```
 205 #define SEG_PAGE_LOCKED         0x02    /* VA has a page that is locked */
 206 #define SEG_PAGE_HASCOW         0x04    /* VA has a page with a copy-on-write */
 207 #define SEG_PAGE_SOFTLOCK       0x08    /* VA has a page with softlock held */
 208 #define SEG_PAGE_VNODEBACKED    0x10    /* Segment is backed by a vnode */
 209 #define SEG_PAGE_ANON           0x20    /* VA has an anonymous page */
 210 #define SEG_PAGE_VNODE          0x40    /* VA has a vnode page backing it */

 212 #define seg_page(seg, addr) \
 213         (((uintptr_t)((addr) - (seg)->s_base)) >> PAGESHIFT)

 215 #define seg_pages(seg) \
 216         (((uintptr_t)((seg)->s_size + PAGEOFFSET)) >> PAGESHIFT)

 218 #define IE_NOMEM        -1      /* internal to seg layer */
 219 #define IE_RETRY        -2      /* internal to seg layer */
 220 #define IE_REATTACH     -3      /* internal to seg layer */

 222 /* Values for segop_inherit */
 222 /* Values for SEGOP_INHERIT */
 223 #define SEGP_INH_ZERO   0x01

 225 int seg_inherit_notsup(struct seg *, caddr_t, size_t, uint_t);

 227 /* Delay/retry factors for seg_p_mem_config_pre_del */
 228 #define SEGP_PREDEL_DELAY_FACTOR        4
 229 /*
 230  * As a workaround to being unable to purge the pagelock
 231  * cache during a DR delete memory operation, we use
 232  * a stall threshold that is twice the maximum seen
 233  * during testing.  This workaround will be removed
 234  * when a suitable fix is found.
 235  */
 236 #define SEGP_STALL_SECONDS      25
 237 #define SEGP_STALL_THRESHOLD \
 238         (SEGP_STALL_SECONDS * SEGP_PREDEL_DELAY_FACTOR)

 240 #ifdef VMDEBUG

 242 uint_t  seg_page(struct seg *, caddr_t);
 243 uint_t  seg_pages(struct seg *);

 245 #endif  /* VMDEBUG */

 247 boolean_t       seg_can_change_zones(struct seg *);
 248 size_t          seg_swresv(struct seg *);

 250 /* segop wrappers */
 251 int segop_dup(struct seg *, struct seg *);
 252 int segop_unmap(struct seg *, caddr_t, size_t);
 253 void segop_free(struct seg *);
 254 faultcode_t segop_fault(struct hat *, struct seg *, caddr_t, size_t, enum fault_
 255 faultcode_t segop_faulta(struct seg *, caddr_t);
 256 int segop_setprot(struct seg *, caddr_t, size_t, uint_t);
 257 int segop_checkprot(struct seg *, caddr_t, size_t, uint_t);
 258 int segop_kluster(struct seg *, caddr_t, ssize_t);
 259 int segop_sync(struct seg *, caddr_t, size_t, int, uint_t);
 260 size_t segop_incore(struct seg *, caddr_t, size_t, char *);
 261 int segop_lockop(struct seg *, caddr_t, size_t, int, int, ulong_t *, size_t );
 262 int segop_getprot(struct seg *, caddr_t, size_t, uint_t *);
 263 u_offset_t segop_getoffset(struct seg *, caddr_t);
 264 int segop_gettype(struct seg *, caddr_t);
 265 int segop_getvp(struct seg *, caddr_t, struct vnode **);
 266 int segop_advise(struct seg *, caddr_t, size_t, uint_t);
 267 void segop_dump(struct seg *);
 268 int segop_pagelock(struct seg *, caddr_t, size_t, struct page ***, enum lock_typ
 269 int segop_setpagesize(struct seg *, caddr_t, size_t, uint_t);
```

```
 270 int segop_getmemid(struct seg *, caddr_t, memid_t *);
 271 struct lgrp_mem_policy_info *segop_getpolicy(struct seg *, caddr_t);
 272 int segop_capable(struct seg *, segcapability_t);
 273 int segop_inherit(struct seg *, caddr_t, size_t, uint_t);

 275 #endif  /* _KERNEL */

 277 #ifdef  __cplusplus
 278 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
  114107 Fri May  8 18:04:09 2015
new/usr/src/uts/common/vm/seg_dev.c
patch lower-case-segops
**********************************************************
______unchanged_portion_omitted_

 355 /*
 356  * Create a device segment.
 357  */
 358 int
 359 segdev_create(struct seg *seg, void *argsp)
 360 {
 361         struct segdev_data *sdp;
 362         struct segdev_crargs *a = (struct segdev_crargs *)argsp;
 363         devmap_handle_t *dhp = (devmap_handle_t *)a->devmap_data;
 364         int error;

 366         /*
 367          * Since the address space is "write" locked, we
 368          * don't need the segment lock to protect "segdev" data.
 369          */
 370         ASSERT(seg->s_as && AS_WRITE_HELD(seg->s_as, &seg->s_as->a_lock));

 372         hat_map(seg->s_as->a_hat, seg->s_base, seg->s_size, HAT_MAP);

 374         sdp = sdp_alloc();

 376         sdp->mapfunc = a->mapfunc;
 377         sdp->offset = a->offset;
 378         sdp->prot = a->prot;
 379         sdp->maxprot = a->maxprot;
 380         sdp->type = a->type;
 381         sdp->pageprot = 0;
 382         sdp->softlockcnt = 0;
 383         sdp->vpage = NULL;

 385         if (sdp->mapfunc == NULL)
 386                 sdp->devmap_data = dhp;
 387         else
 388                 sdp->devmap_data = dhp = NULL;

 390         sdp->hat_flags = a->hat_flags;
 391         sdp->hat_attr = a->hat_attr;

 393         /*
 394          * Currently, hat_flags supports only HAT_LOAD_NOCONSIST
 395          */
 396         ASSERT(!(sdp->hat_flags & ~HAT_LOAD_NOCONSIST));

 398         /*
 399          * Hold shadow vnode -- segdev only deals with
 400          * character (VCHR) devices. We use the common
 401          * vp to hang pages on.
 402          */
 403         sdp->vp = specfind(a->dev, VCHR);
 404         ASSERT(sdp->vp != NULL);

 406         seg->s_ops = &segdev_ops;
 407         seg->s_data = sdp;

 409         while (dhp != NULL) {
 410                 dhp->dh_seg = seg;
 411                 dhp = dhp->dh_next;
 412         }
```



```
 414         /*
 415          * Inform the vnode of the new mapping.
 416          */
 417         /*
 418          * It is ok to use pass sdp->maxprot to ADDMAP rather than to use
 419          * dhp specific maxprot because spec_addmap does not use maxprot.
 420          */
 421         error = VOP_ADDMAP(VTOCVP(sdp->vp), sdp->offset,
 422             seg->s_as, seg->s_base, seg->s_size,
 423             sdp->prot, sdp->maxprot, sdp->type, CRED(), NULL);

 425         if (error != 0) {
 426                 sdp->devmap_data = NULL;
 427                 hat_unload(seg->s_as->a_hat, seg->s_base, seg->s_size,
 428                     HAT_UNLOAD_UNMAP);
 429         } else {
 430                 /*
 431                  * Mappings of /dev/null don't count towards the VSZ of a
 432                  * process.  Mappings of /dev/null have no mapping type.
 433                  */
 434                 if ((segop_gettype(seg, seg->s_base) & (MAP_SHARED |
 434                 if ((SEGOP_GETTYPE(seg, (seg)->s_base) & (MAP_SHARED |
 435                     MAP_PRIVATE)) == 0) {
 436                         seg->s_as->a_resvsize -= seg->s_size;
 437                 }
 438         }

 440         return (error);
 441 }
______unchanged_portion_omitted_
```

```
**********************************************************
    4464 Fri May  8 18:04:09 2015
new/usr/src/uts/common/vm/seg_dev.h
patch lower-case-segops
**********************************************************
______unchanged_portion_omitted_

 113 #ifdef _KERNEL

 115 /*
 116  * Mappings of /dev/null come from segdev and have no mapping type.
 117  */

 119 #define SEG_IS_DEVNULL_MAPPING(seg)     \
 120         ((seg)->s_ops == &segdev_ops && \
 121         ((segop_gettype((seg), (seg)->s_base) & (MAP_SHARED | MAP_PRIVATE)) == 0
 121         ((SEGOP_GETTYPE(seg, (seg)->s_base) & (MAP_SHARED | MAP_PRIVATE)) == 0))

 123 extern void segdev_init(void);

 125 extern int segdev_create(struct seg *, void *);

 127 extern int segdev_copyto(struct seg *, caddr_t, const void *, void *, size_t);
 128 extern int segdev_copyfrom(struct seg *, caddr_t, const void *, void *, size_t);
 129 extern struct seg_ops segdev_ops;

 131 #endif  /* _KERNEL */

 133 #ifdef  __cplusplus
 134 }
______unchanged_portion_omitted_
```

```
**********************************************************
   45428 Fri May  8 18:04:09 2015
new/usr/src/uts/common/vm/seg_kmem.c
patch lower-case-segops
**********************************************************
______unchanged_portion_omitted_

#define SEGKMEM_BADOP(t)        (t(*)())segkmem_badop

/*ARGSUSED*/
static faultcode_t
segkmem_fault(struct hat *hat, struct seg *seg, caddr_t addr, size_t size,
        enum fault_type type, enum seg_rw rw)
{
        pgcnt_t npages;
        spgcnt_t pg;
        page_t *pp;
        struct vnode *vp = seg->s_data;

        ASSERT(RW_READ_HELD(&seg->s_as->a_lock));

        if (seg->s_as != &kas || size > seg->s_size ||
            addr < seg->s_base || addr + size > seg->s_base + seg->s_size)
                panic("segkmem_fault: bad args");

        /*
         * If it is one of segkp pages, call segkp_fault.
         */
        if (segkp_bitmap && seg == &kvseg &&
            BT_TEST(segkp_bitmap, btop((uintptr_t)(addr - seg->s_base))))
                return (segop_fault(hat, segkp, addr, size, type, rw));
                return (SEGOP_FAULT(hat, segkp, addr, size, type, rw));

        if (rw != S_READ && rw != S_WRITE && rw != S_OTHER)
                return (FC_NOSUPPORT);

        npages = btopr(size);

        switch (type) {
        case F_SOFTLOCK:        /* lock down already-loaded translations */
                for (pg = 0; pg < npages; pg++) {
                        pp = page_lookup(vp, (u_offset_t)(uintptr_t)addr,
                            SE_SHARED);
                        if (pp == NULL) {
                                /*
                                 * Hmm, no page. Does a kernel mapping
                                 * exist for it?
                                 */
                                if (!hat_probe(kas.a_hat, addr)) {
                                        addr -= PAGESIZE;
                                        while (--pg >= 0) {
                                                pp = page_find(vp, (u_offset_t)
                                                    (uintptr_t)addr);
                                                if (pp)
                                                        page_unlock(pp);
                                                addr -= PAGESIZE;
                                        }
                                        return (FC_NOMAP);
                                }
                        }
                        addr += PAGESIZE;
                }
                if (rw == S_OTHER)
                        hat_reserve(seg->s_as, addr, size);
                return (0);
        case F_SOFTUNLOCK:
```



```
                while (npages--) {
                        pp = page_find(vp, (u_offset_t)(uintptr_t)addr);
                        if (pp)
                                page_unlock(pp);
                        addr += PAGESIZE;
                }
                return (0);
        default:
                return (FC_NOSUPPORT);
        }
        /*NOTREACHED*/
}

static int
segkmem_setprot(struct seg *seg, caddr_t addr, size_t size, uint_t prot)
{
        ASSERT(RW_LOCK_HELD(&seg->s_as->a_lock));

        if (seg->s_as != &kas || size > seg->s_size ||
            addr < seg->s_base || addr + size > seg->s_base + seg->s_size)
                panic("segkmem_setprot: bad args");

        /*
         * If it is one of segkp pages, call segkp.
         */
        if (segkp_bitmap && seg == &kvseg &&
            BT_TEST(segkp_bitmap, btop((uintptr_t)(addr - seg->s_base))))
                return (segop_setprot(segkp, addr, size, prot));
                return (SEGOP_SETPROT(segkp, addr, size, prot));

        if (prot == 0)
                hat_unload(kas.a_hat, addr, size, HAT_UNLOAD);
        else
                hat_chgprot(kas.a_hat, addr, size, prot);
        return (0);
}

/*
 * This is a dummy segkmem function overloaded to call segkp
 * when segkp is under the heap.
 */
/* ARGSUSED */
static int
segkmem_checkprot(struct seg *seg, caddr_t addr, size_t size, uint_t prot)
{
        ASSERT(RW_LOCK_HELD(&seg->s_as->a_lock));

        if (seg->s_as != &kas)
                segkmem_badop();

        /*
         * If it is one of segkp pages, call into segkp.
         */
        if (segkp_bitmap && seg == &kvseg &&
            BT_TEST(segkp_bitmap, btop((uintptr_t)(addr - seg->s_base))))
                return (segop_checkprot(segkp, addr, size, prot));
                return (SEGOP_CHECKPROT(segkp, addr, size, prot));

        segkmem_badop();
        return (0);
}

/*
 * This is a dummy segkmem function overloaded to call segkp
 * when segkp is under the heap.
 */
```

```
559 /* ARGSUSED */
560 static int
561 segkmem_kluster(struct seg *seg, caddr_t addr, ssize_t delta)
562 {
563         ASSERT(RW_LOCK_HELD(&seg->s_as->a_lock));

565         if (seg->s_as != &kas)
566                 segkmem_badop();

568         /*
569          * If it is one of segkp pages, call into segkp.
570          */
571         if (segkp_bitmap && seg == &kvseg &&
572             BT_TEST(segkp_bitmap, btop((uintptr_t)(addr - seg->s_base))))
573                 return (segop_kluster(segkp, addr, delta));
573                 return (SEGOP_KLUSTER(segkp, addr, delta));

575         segkmem_badop();
576         return (0);
577 }
______unchanged_portion_omitted_

669 /*
670  * lock/unlock kmem pages over a given range [addr, addr+len).
671  * Returns a shadow list of pages in ppp. If there are holes
672  * in the range (e.g. some of the kernel mappings do not have
673  * underlying page_ts) returns ENOTSUP so that as_pagelock()
674  * will handle the range via as_fault(F_SOFTLOCK).
675  */
676 /*ARGSUSED*/
677 static int
678 segkmem_pagelock(struct seg *seg, caddr_t addr, size_t len,
679         page_t ***ppp, enum lock_type type, enum seg_rw rw)
680 {
681         page_t **pplist, *pp;
682         pgcnt_t npages;
683         spgcnt_t pg;
684         size_t nb;
685         struct vnode *vp = seg->s_data;

687         ASSERT(ppp != NULL);

689         /*
690          * If it is one of segkp pages, call into segkp.
691          */
692         if (segkp_bitmap && seg == &kvseg &&
693             BT_TEST(segkp_bitmap, btop((uintptr_t)(addr - seg->s_base))))
694                 return (segop_pagelock(segkp, addr, len, ppp, type, rw));
694                 return (SEGOP_PAGELOCK(segkp, addr, len, ppp, type, rw));

696         npages = btopr(len);
697         nb = sizeof (page_t *) * npages;

699         if (type == L_PAGEUNLOCK) {
700                 pplist = *ppp;
701                 ASSERT(pplist != NULL);

703                 for (pg = 0; pg < npages; pg++) {
704                         pp = pplist[pg];
705                         page_unlock(pp);
706                 }
707                 kmem_free(pplist, nb);
708                 return (0);
709         }

711         ASSERT(type == L_PAGELOCK);
```



```
713         pplist = kmem_alloc(nb, KM_NOSLEEP);
714         if (pplist == NULL) {
715                 *ppp = NULL;
716                 return (ENOTSUP);       /* take the slow path */
717         }

719         for (pg = 0; pg < npages; pg++) {
720                 pp = page_lookup(vp, (u_offset_t)(uintptr_t)addr, SE_SHARED);
721                 if (pp == NULL) {
722                         while (--pg >= 0)
723                                 page_unlock(pplist[pg]);
724                         kmem_free(pplist, nb);
725                         *ppp = NULL;
726                         return (ENOTSUP);
727                 }
728                 pplist[pg] = pp;
729                 addr += PAGESIZE;
730         }

732         *ppp = pplist;
733         return (0);
734 }

736 /*
737  * This is a dummy segkmem function overloaded to call segkp
738  * when segkp is under the heap.
739  */
740 /* ARGSUSED */
741 static int
742 segkmem_getmemid(struct seg *seg, caddr_t addr, memid_t *memidp)
743 {
744         ASSERT(RW_LOCK_HELD(&seg->s_as->a_lock));

746         if (seg->s_as != &kas)
747                 segkmem_badop();

749         /*
750          * If it is one of segkp pages, call into segkp.
751          */
752         if (segkp_bitmap && seg == &kvseg &&
753             BT_TEST(segkp_bitmap, btop((uintptr_t)(addr - seg->s_base))))
754                 return (segop_getmemid(segkp, addr, memidp));
754                 return (SEGOP_GETMEMID(segkp, addr, memidp));

756         segkmem_badop();
757         return (0);
758 }
______unchanged_portion_omitted_
```

```
**********************************************************
  280654 Fri May  8 18:04:10 2015
new/usr/src/uts/common/vm/seg_vn.c
patch lower-case-segops
**********************************************************
______unchanged_portion_omitted_

6063 /*
6064  * segvn_setpagesize is called via segop_setpagesize from as_setpagesize,
6064  * segvn_setpagesize is called via SEGOP_SETPAGESIZE from as_setpagesize,
6065  * to determine if the seg is capable of mapping the requested szc.
6066  */
6067 static int
6068 segvn_setpagesize(struct seg *seg, caddr_t addr, size_t len, uint_t szc)
6069 {
6070         struct segvn_data *svd = (struct segvn_data *)seg->s_data;
6071         struct segvn_data *nsvd;
6072         struct anon_map *amp = svd->amp;
6073         struct seg *nseg;
6074         caddr_t eaddr = addr + len, a;
6075         size_t pgsz = page_get_pagesize(szc);
6076         pgcnt_t pgcnt = page_get_pagecnt(szc);
6077         int err;
6078         u_offset_t off = svd->offset + (uintptr_t)(addr - seg->s_base);

6080         ASSERT(seg->s_as && AS_WRITE_HELD(seg->s_as, &seg->s_as->a_lock));
6081         ASSERT(addr >= seg->s_base && eaddr <= seg->s_base + seg->s_size);

6083         if (seg->s_szc == szc || segvn_lpg_disable != 0) {
6084                 return (0);
6085         }

6087         /*
6088          * addr should always be pgsz aligned but eaddr may be misaligned if
6089          * it's at the end of the segment.
6090          *
6091          * XXX we should assert this condition since as_setpagesize() logic
6092          * guarantees it.
6093          */
6094         if (!IS_P2ALIGNED(addr, pgsz) ||
6095             (!IS_P2ALIGNED(eaddr, pgsz) &&
6096             eaddr != seg->s_base + seg->s_size)) {

6098                 segvn_setpgsz_align_err++;
6099                 return (EINVAL);
6100         }

6102         if (amp != NULL && svd->type == MAP_SHARED) {
6103                 ulong_t an_idx = svd->anon_index + seg_page(seg, addr);
6104                 if (!IS_P2ALIGNED(an_idx, pgcnt)) {

6106                         segvn_setpgsz_anon_align_err++;
6107                         return (EINVAL);
6108                 }
6109         }

6111         if ((svd->flags & MAP_NORESERVE) || seg->s_as == &kas ||
6112             szc > segvn_maxpgszc) {
6113                 return (EINVAL);
6114         }

6116         /* paranoid check */
6117         if (svd->vp != NULL &&
6118             (IS_SWAPFSVP(svd->vp) || VN_ISKAS(svd->vp))) {
6119                 return (EINVAL);
6120         }
```



```
6122         if (seg->s_szc == 0 && svd->vp != NULL &&
6123             map_addr_vacalign_check(addr, off)) {
6124                 return (EINVAL);
6125         }

6127         /*
6128          * Check that protections are the same within new page
6129          * size boundaries.
6130          */
6131         if (svd->pageprot) {
6132                 for (a = addr; a < eaddr; a += pgsz) {
6133                         if ((a + pgsz) > eaddr) {
6134                                 if (!sameprot(seg, a, eaddr - a)) {
6135                                         return (EINVAL);
6136                                 }
6137                         } else {
6138                                 if (!sameprot(seg, a, pgsz)) {
6139                                         return (EINVAL);
6140                                 }
6141                         }
6142                 }
6143         }

6145         /*
6146          * Since we are changing page size we first have to flush
6147          * the cache. This makes sure all the pagelock calls have
6148          * to recheck protections.
6149          */
6150         if (svd->softlockcnt > 0) {
6151                 ASSERT(svd->tr_state == SEGVN_TR_OFF);

6153                 /*
6154                  * If this is shared segment non 0 softlockcnt
6155                  * means locked pages are still in use.
6156                  */
6157                 if (svd->type == MAP_SHARED) {
6158                         return (EAGAIN);
6159                 }

6161                 /*
6162                  * Since we do have the segvn writers lock nobody can fill
6163                  * the cache with entries belonging to this seg during
6164                  * the purge. The flush either succeeds or we still have
6165                  * pending I/Os.
6166                  */
6167                 segvn_purge(seg);
6168                 if (svd->softlockcnt > 0) {
6169                         return (EAGAIN);
6170                 }
6171         }

6173         if (HAT_IS_REGION_COOKIE_VALID(svd->rcookie)) {
6174                 ASSERT(svd->amp == NULL);
6175                 ASSERT(svd->tr_state == SEGVN_TR_OFF);
6176                 hat_leave_region(seg->s_as->a_hat, svd->rcookie,
6177                     HAT_REGION_TEXT);
6178                 svd->rcookie = HAT_INVALID_REGION_COOKIE;
6179         } else if (svd->tr_state == SEGVN_TR_INIT) {
6180                 svd->tr_state = SEGVN_TR_OFF;
6181         } else if (svd->tr_state == SEGVN_TR_ON) {
6182                 ASSERT(svd->amp != NULL);
6183                 segvn_textunrepl(seg, 1);
6184                 ASSERT(svd->amp == NULL && svd->tr_state == SEGVN_TR_OFF);
6185                 amp = NULL;
6186         }
```

```
6188         /*
6189          * Operation for sub range of existing segment.
6190          */
6191         if (addr != seg->s_base || eaddr != (seg->s_base + seg->s_size)) {
6192                 if (szc < seg->s_szc) {
6193                         VM_STAT_ADD(segvnvmstats.demoterange[2]);
6194                         err = segvn_demote_range(seg, addr, len, SDR_RANGE, 0);
6195                         if (err == 0) {
6196                                 return (IE_RETRY);
6197                         }
6198                         if (err == ENOMEM) {
6199                                 return (IE_NOMEM);
6200                         }
6201                         return (err);
6202                 }
6203                 if (addr != seg->s_base) {
6204                         nseg = segvn_split_seg(seg, addr);
6205                         if (eaddr != (nseg->s_base + nseg->s_size)) {
6206                                 /* eaddr is szc aligned */
6207                                 (void) segvn_split_seg(nseg, eaddr);
6208                         }
6209                         return (IE_RETRY);
6210                 }
6211                 if (eaddr != (seg->s_base + seg->s_size)) {
6212                         /* eaddr is szc aligned */
6213                         (void) segvn_split_seg(seg, eaddr);
6214                 }
6215                 return (IE_RETRY);
6216         }

6218         /*
6219          * Break any low level sharing and reset seg->s_szc to 0.
6220          */
6221         if ((err = segvn_clrszc(seg)) != 0) {
6222                 if (err == ENOMEM) {
6223                         err = IE_NOMEM;
6224                 }
6225                 return (err);
6226         }
6227         ASSERT(seg->s_szc == 0);

6229         /*
6230          * If the end of the current segment is not pgsz aligned
6231          * then attempt to concatenate with the next segment.
6232          */
6233         if (!IS_P2ALIGNED(eaddr, pgsz)) {
6234                 nseg = AS_SEGNEXT(seg->s_as, seg);
6235                 if (nseg == NULL || nseg == seg || eaddr != nseg->s_base) {
6236                         return (ENOMEM);
6237                 }
6238                 if (nseg->s_ops != &segvn_ops) {
6239                         return (EINVAL);
6240                 }
6241                 nsvd = (struct segvn_data *)nseg->s_data;
6242                 if (nsvd->softlockcnt > 0) {
6243                         /*
6244                          * If this is shared segment non 0 softlockcnt
6245                          * means locked pages are still in use.
6246                          */
6247                         if (nsvd->type == MAP_SHARED) {
6248                                 return (EAGAIN);
6249                         }
6250                         segvn_purge(nseg);
6251                         if (nsvd->softlockcnt > 0) {
6252                                 return (EAGAIN);
```



```
6253                         }
6254                 }
6255                 err = segvn_clrszc(nseg);
6256                 if (err == ENOMEM) {
6257                         err = IE_NOMEM;
6258                 }
6259                 if (err != 0) {
6260                         return (err);
6261                 }
6262                 ASSERT(nsvd->rcookie == HAT_INVALID_REGION_COOKIE);
6263                 err = segvn_concat(seg, nseg, 1);
6264                 if (err == -1) {
6265                         return (EINVAL);
6266                 }
6267                 if (err == -2) {
6268                         return (IE_NOMEM);
6269                 }
6270                 return (IE_RETRY);
6271         }

6273         /*
6274          * May need to re-align anon array to
6275          * new szc.
6276          */
6277         if (amp != NULL) {
6278                 if (!IS_P2ALIGNED(svd->anon_index, pgcnt)) {
6279                         struct anon_hdr *nahp;

6281                         ASSERT(svd->type == MAP_PRIVATE);

6283                         ANON_LOCK_ENTER(&amp->a_rwlock, RW_WRITER);
6284                         ASSERT(amp->refcnt == 1);
6285                         nahp = anon_create(btop(amp->size), ANON_NOSLEEP);
6286                         if (nahp == NULL) {
6287                                 ANON_LOCK_EXIT(&amp->a_rwlock);
6288                                 return (IE_NOMEM);
6289                         }
6290                         if (anon_copy_ptr(amp->ahp, svd->anon_index,
6291                             nahp, 0, btop(seg->s_size), ANON_NOSLEEP)) {
6292                                 anon_release(nahp, btop(amp->size));
6293                                 ANON_LOCK_EXIT(&amp->a_rwlock);
6294                                 return (IE_NOMEM);
6295                         }
6296                         anon_release(amp->ahp, btop(amp->size));
6297                         amp->ahp = nahp;
6298                         svd->anon_index = 0;
6299                         ANON_LOCK_EXIT(&amp->a_rwlock);
6300                 }
6301         }
6302         if (svd->vp != NULL && szc != 0) {
6303                 struct vattr va;
6304                 u_offset_t eoffpage = svd->offset;
6305                 va.va_mask = AT_SIZE;
6306                 eoffpage += seg->s_size;
6307                 eoffpage = btopr(eoffpage);
6308                 if (VOP_GETATTR(svd->vp, &va, 0, svd->cred, NULL) != 0) {
6309                         segvn_setpgsz_getattr_err++;
6310                         return (EINVAL);
6311                 }
6312                 if (btopr(va.va_size) < eoffpage) {
6313                         segvn_setpgsz_eof_err++;
6314                         return (EINVAL);
6315                 }
6316                 if (amp != NULL) {
6317                         /*
6318                          * anon_fill_cow_holes() may call VOP_GETPAGE().
```

```
6319                          * don't take anon map lock here to avoid holding it
6320                          * across VOP_GETPAGE() calls that may call back into
6321                          * segvn for klsutering checks. We don't really need
6322                          * anon map lock here since it's a private segment and
6323                          * we hold as level lock as writers.
6324                          */
6325                         if ((err = anon_fill_cow_holes(seg, seg->s_base,
6326                             amp->ahp, svd->anon_index, svd->vp, svd->offset,
6327                             seg->s_size, szc, svd->prot, svd->vpage,
6328                             svd->cred)) != 0) {
6329                                 return (EINVAL);
6330                         }
6331                 }
6332                 segvn_setvnode_mpss(svd->vp);
6333         }

6335         if (amp != NULL) {
6336                 ANON_LOCK_ENTER(&amp->a_rwlock, RW_WRITER);
6337                 if (svd->type == MAP_PRIVATE) {
6338                         amp->a_szc = szc;
6339                 } else if (szc > amp->a_szc) {
6340                         amp->a_szc = szc;
6341                 }
6342                 ANON_LOCK_EXIT(&amp->a_rwlock);
6343         }

6345         seg->s_szc = szc;

6347         return (0);
6348 }
```

*_____unchanged_portion_omitted_*

```
**********************************************************
   90695 Fri May  8 18:04:10 2015
new/usr/src/uts/common/vm/vm_as.c
patch lower-case-segops
**********************************************************
______unchanged_portion_omitted_

 676 /*
 677  * Free an address space data structure.
 678  * Need to free the hat first and then
 679  * all the segments on this as and finally
 680  * the space for the as struct itself.
 681  */
 682 void
 683 as_free(struct as *as)
 684 {
 685         struct hat *hat = as->a_hat;
 686         struct seg *seg, *next;
 687         boolean_t free_started = B_FALSE;

 689 top:
 690         /*
 691          * Invoke ALL callbacks. as_do_callbacks will do one callback
 692          * per call, and not return (-1) until the callback has completed.
 693          * When as_do_callbacks returns zero, all callbacks have completed.
 694          */
 695         mutex_enter(&as->a_contents);
 696         while (as->a_callbacks && as_do_callbacks(as, AS_ALL_EVENT, 0, 0))
 697                 ;

 699         mutex_exit(&as->a_contents);
 700         AS_LOCK_ENTER(as, &as->a_lock, RW_WRITER);

 702         if (!free_started) {
 703                 free_started = B_TRUE;
 704                 hat_free_start(hat);
 705         }
 706         for (seg = AS_SEGFIRST(as); seg != NULL; seg = next) {
 707                 int err;

 709                 next = AS_SEGNEXT(as, seg);
 710 retry:
 711                 err = segop_unmap(seg, seg->s_base, seg->s_size);
 711                 err = SEGOP_UNMAP(seg, seg->s_base, seg->s_size);
 712                 if (err == EAGAIN) {
 713                         mutex_enter(&as->a_contents);
 714                         if (as->a_callbacks) {
 715                                 AS_LOCK_EXIT(as, &as->a_lock);
 716                         } else if (!AS_ISNOUNMAPWAIT(as)) {
 717                                 /*
 718                                  * Memory is currently locked. Wait for a
 719                                  * cv_signal that it has been unlocked, then
 720                                  * try the operation again.
 721                                  */
 722                                 if (AS_ISUNMAPWAIT(as) == 0)
 723                                         cv_broadcast(&as->a_cv);
 724                                 AS_SETUNMAPWAIT(as);
 725                                 AS_LOCK_EXIT(as, &as->a_lock);
 726                                 while (AS_ISUNMAPWAIT(as))
 727                                         cv_wait(&as->a_cv, &as->a_contents);
 728                         } else {
 729                                 /*
 730                                  * We may have raced with
 731                                  * segvn_reclaim()/segspt_reclaim(). In this
 732                                  * case clean nounmapwait flag and retry since
 733                                  * softlockcnt in this segment may be already
```



```
 734                                  * 0.  We don't drop as writer lock so our
 735                                  * number of retries without sleeping should
 736                                  * be very small. See segvn_reclaim() for
 737                                  * more comments.
 738                                  */
 739                                 AS_CLRNOUNMAPWAIT(as);
 740                                 mutex_exit(&as->a_contents);
 741                                 goto retry;
 742                         }
 743                         mutex_exit(&as->a_contents);
 744                         goto top;
 745                 } else {
 746                         /*
 747                          * We do not expect any other error return at this
 748                          * time. This is similar to an ASSERT in seg_unmap()
 749                          */
 750                         ASSERT(err == 0);
 751                 }
 752         }
 753         hat_free_end(hat);
 754         AS_LOCK_EXIT(as, &as->a_lock);

 756         /* /proc stuff */
 757         ASSERT(avl_numnodes(&as->a_wpage) == 0);
 758         if (as->a_objectdir) {
 759                 kmem_free(as->a_objectdir, as->a_sizedir * sizeof (vnode_t *));
 760                 as->a_objectdir = NULL;
 761                 as->a_sizedir = 0;
 762         }

 764         /*
 765          * Free the struct as back to kmem.  Assert it has no segments.
 766          */
 767         ASSERT(avl_numnodes(&as->a_segtree) == 0);
 768         kmem_cache_free(as_cache, as);
 769 }

 771 int
 772 as_dup(struct as *as, struct proc *forkedproc)
 773 {
 774         struct as *newas;
 775         struct seg *seg, *newseg;
 776         size_t  purgesize = 0;
 777         int error;

 779         AS_LOCK_ENTER(as, &as->a_lock, RW_WRITER);
 780         as_clearwatch(as);
 781         newas = as_alloc();
 782         newas->a_userlimit = as->a_userlimit;
 783         newas->a_proc = forkedproc;

 785         AS_LOCK_ENTER(newas, &newas->a_lock, RW_WRITER);

 787         (void) hat_dup(as->a_hat, newas->a_hat, NULL, 0, HAT_DUP_SRD);

 789         for (seg = AS_SEGFIRST(as); seg != NULL; seg = AS_SEGNEXT(as, seg)) {

 791                 if (seg->s_flags & S_PURGE) {
 792                         purgesize += seg->s_size;
 793                         continue;
 794                 }

 796                 newseg = seg_alloc(newas, seg->s_base, seg->s_size);
 797                 if (newseg == NULL) {
 798                         AS_LOCK_EXIT(newas, &newas->a_lock);
 799                         as_setwatch(as);
```

```
800                         AS_LOCK_EXIT(as, &as->a_lock);
801                         as_free(newas);
802                         return (-1);
803                 }
804                 if ((error = segop_dup(seg, newseg)) != 0) {
804                 if ((error = SEGOP_DUP(seg, newseg)) != 0) {
805                         /*
806                          * We call seg_free() on the new seg
807                          * because the segment is not set up
808                          * completely; i.e. it has no ops.
809                          */
810                         as_setwatch(as);
811                         AS_LOCK_EXIT(as, &as->a_lock);
812                         seg_free(newseg);
813                         AS_LOCK_EXIT(newas, &newas->a_lock);
814                         as_free(newas);
815                         return (error);
816                 }
817                 newas->a_size += seg->s_size;
818         }
819         newas->a_resvsize = as->a_resvsize - purgesize;

821         error = hat_dup(as->a_hat, newas->a_hat, NULL, 0, HAT_DUP_ALL);

823         AS_LOCK_EXIT(newas, &newas->a_lock);

825         as_setwatch(as);
826         AS_LOCK_EXIT(as, &as->a_lock);
827         if (error != 0) {
828                 as_free(newas);
829                 return (error);
830         }
831         forkedproc->p_as = newas;
832         return (0);
833 }

835 /*
836  * Handle a ‘‘fault’’ at addr for size bytes.
837  */
838 faultcode_t
839 as_fault(struct hat *hat, struct as *as, caddr_t addr, size_t size,
840         enum fault_type type, enum seg_rw rw)
841 {
842         struct seg *seg;
843         caddr_t raddr;                  /* rounded down addr */
844         size_t rsize;                   /* rounded up size */
845         size_t ssize;
846         faultcode_t res = 0;
847         caddr_t addrsav;
848         struct seg *segsav;
849         int as_lock_held;
850         klwp_t *lwp = ttolwp(curthread);
851         int holding_wpage = 0;



855 retry:
856         /*
857          * Indicate that the lwp is not to be stopped while waiting for a
858          * pagefault.  This is to avoid deadlock while debugging a process
859          * via /proc over NFS (in particular).
860          */
861         if (lwp != NULL)
862                 lwp->lwp_nostop++;

864         /*
```



```
865          * same length must be used when we softlock and softunlock.  We
866          * don’t support softunlocking lengths less than the original length
867          * when there is largepage support.  See seg_dev.c for more
868          * comments.
869          */
870         switch (type) {

872         case F_SOFTLOCK:
873                 CPU_STATS_ADD_K(vm, softlock, 1);
874                 break;

876         case F_SOFTUNLOCK:
877                 break;

879         case F_PROT:
880                 CPU_STATS_ADD_K(vm, prot_fault, 1);
881                 break;

883         case F_INVAL:
884                 CPU_STATS_ENTER_K();
885                 CPU_STATS_ADDQ(CPU, vm, as_fault, 1);
886                 if (as == &kas)
887                         CPU_STATS_ADDQ(CPU, vm, kernel_asflt, 1);
888                 CPU_STATS_EXIT_K();
889                 break;
890         }

892         /* Kernel probe */
893         TNF_PROBE_3(address_fault, "vm pagefault", /* CSTYLED */,
894             tnf_opaque, address,        addr,
895             tnf_fault_type,     fault_type,     type,
896             tnf_seg_access,     access,         rw);

898         raddr = (caddr_t)((uintptr_t)addr & (uintptr_t)PAGEMASK);
899         rsize = (((size_t)(addr + size) + PAGEOFFSET) & PAGEMASK) -
900             (size_t)raddr;

902         /*
903          * XXX -- Don’t grab the as lock for segkmap. We should grab it for
904          * correctness, but then we could be stuck holding this lock for
905          * a LONG time if the fault needs to be resolved on a slow
906          * filesystem, and then no-one will be able to exec new commands,
907          * as exec’ing requires the write lock on the as.
908          */
909         if (as == &kas && segkmap && segkmap->s_base <= raddr &&
910             raddr + size < segkmap->s_base + segkmap->s_size) {
911                 seg = segkmap;
912                 as_lock_held = 0;
913         } else {
914                 AS_LOCK_ENTER(as, &as->a_lock, RW_READER);

916                 seg = as_segat(as, raddr);
917                 if (seg == NULL) {
918                         AS_LOCK_EXIT(as, &as->a_lock);
919                         if (lwp != NULL)
920                                 lwp->lwp_nostop--;
921                         return (FC_NOMAP);
922                 }

924                 as_lock_held = 1;
925         }

927         addrsav = raddr;
928         segsav = seg;

930         for (; rsize != 0; rsize -= ssize, raddr += ssize) {
```

```
931                 if (raddr >= seg->s_base + seg->s_size) {
932                         seg = AS_SEGNEXT(as, seg);
933                         if (seg == NULL || raddr != seg->s_base) {
934                                 res = FC_NOMAP;
935                                 break;
936                         }
937                 }
938                 if (raddr + rsize > seg->s_base + seg->s_size)
939                         ssize = seg->s_base + seg->s_size - raddr;
940                 else
941                         ssize = rsize;

943                 res = segop_fault(hat, seg, raddr, ssize, type, rw);
943                 res = SEGOP_FAULT(hat, seg, raddr, ssize, type, rw);

945                 /* Restore watchpoints */
946                 if (holding_wpage) {
947                         as_setwatch(as);
948                         holding_wpage = 0;
949                 }

951                 if (res != 0)
952                         break;
953         }

955         /*
956          * If we were SOFTLOCKing and encountered a failure,
957          * we must SOFTUNLOCK the range we already did. (Maybe we
958          * should just panic if we are SOFTLOCKing or even SOFTUNLOCKing
959          * right here...)
960          */
961         if (res != 0 && type == F_SOFTLOCK) {
962                 for (seg = segsav; addrsav < raddr; addrsav += ssize) {
963                         if (addrsav >= seg->s_base + seg->s_size)
964                                 seg = AS_SEGNEXT(as, seg);
965                         ASSERT(seg != NULL);
966                         /*
967                          * Now call the fault routine again to perform the
968                          * unlock using S_OTHER instead of the rw variable
969                          * since we never got a chance to touch the pages.
970                          */
971                         if (raddr > seg->s_base + seg->s_size)
972                                 ssize = seg->s_base + seg->s_size - addrsav;
973                         else
974                                 ssize = raddr - addrsav;
975                         (void) segop_fault(hat, seg, addrsav, ssize,
975                         (void) SEGOP_FAULT(hat, seg, addrsav, ssize,
976                             F_SOFTUNLOCK, S_OTHER);
977                 }
978         }
979         if (as_lock_held)
980                 AS_LOCK_EXIT(as, &as->a_lock);
981         if (lwp != NULL)
982                 lwp->lwp_nostop--;

984         /*
985          * If the lower levels returned EDEADLK for a fault,
986          * It means that we should retry the fault.  Let's wait
987          * a bit also to let the deadlock causing condition clear.
988          * This is part of a gross hack to work around a design flaw
989          * in the ufs/sds logging code and should go away when the
990          * logging code is re-designed to fix the problem. See bug
991          * 4125102 for details of the problem.
992          */
993         if (FC_ERRNO(res) == EDEADLK) {
994                 delay(deadlk_wait);
```



```
995                 res = 0;
996                 goto retry;
997         }
998         return (res);
999 }



1003 /*
1004  * Asynchronous ''fault'' at addr for size bytes.
1005  */
1006 faultcode_t
1007 as_faulta(struct as *as, caddr_t addr, size_t size)
1008 {
1009         struct seg *seg;
1010         caddr_t raddr;                  /* rounded down addr */
1011         size_t rsize;                   /* rounded up size */
1012         faultcode_t res = 0;
1013         klwp_t *lwp = ttolwp(curthread);

1015 retry:
1016         /*
1017          * Indicate that the lwp is not to be stopped while waiting
1018          * for a pagefault.  This is to avoid deadlock while debugging
1019          * a process via /proc over NFS (in particular).
1020          */
1021         if (lwp != NULL)
1022                 lwp->lwp_nostop++;

1024         raddr = (caddr_t)((uintptr_t)addr & (uintptr_t)PAGEMASK);
1025         rsize = (((size_t)(addr + size) + PAGEOFFSET) & PAGEMASK) -
1026             (size_t)raddr;

1028         AS_LOCK_ENTER(as, &as->a_lock, RW_READER);
1029         seg = as_segat(as, raddr);
1030         if (seg == NULL) {
1031                 AS_LOCK_EXIT(as, &as->a_lock);
1032                 if (lwp != NULL)
1033                         lwp->lwp_nostop--;
1034                 return (FC_NOMAP);
1035         }

1037         for (; rsize != 0; rsize -= PAGESIZE, raddr += PAGESIZE) {
1038                 if (raddr >= seg->s_base + seg->s_size) {
1039                         seg = AS_SEGNEXT(as, seg);
1040                         if (seg == NULL || raddr != seg->s_base) {
1041                                 res = FC_NOMAP;
1042                                 break;
1043                         }
1044                 }
1045                 res = segop_faulta(seg, raddr);
1045                 res = SEGOP_FAULTA(seg, raddr);
1046                 if (res != 0)
1047                         break;
1048         }
1049         AS_LOCK_EXIT(as, &as->a_lock);
1050         if (lwp != NULL)
1051                 lwp->lwp_nostop--;
1052         /*
1053          * If the lower levels returned EDEADLK for a fault,
1054          * It means that we should retry the fault.  Let's wait
1055          * a bit also to let the deadlock causing condition clear.
1056          * This is part of a gross hack to work around a design flaw
1057          * in the ufs/sds logging code and should go away when the
1058          * logging code is re-designed to fix the problem. See bug
1059          * 4125102 for details of the problem.
```

```
1060          */
1061         if (FC_ERRNO(res) == EDEADLK) {
1062                 delay(deadlk_wait);
1063                 res = 0;
1064                 goto retry;
1065         }
1066         return (res);
1067 }

1069 /*
1070  * Set the virtual mapping for the interval from [addr : addr + size)
1071  * in address space 'as' to have the specified protection.
1072  * It is ok for the range to cross over several segments,
1073  * as long as they are contiguous.
1074  */
1075 int
1076 as_setprot(struct as *as, caddr_t addr, size_t size, uint_t prot)
1077 {
1078         struct seg *seg;
1079         struct as_callback *cb;
1080         size_t ssize;
1081         caddr_t raddr;                  /* rounded down addr */
1082         size_t rsize;                   /* rounded up size */
1083         int error = 0, writer = 0;
1084         caddr_t saveraddr;
1085         size_t saversize;

1087 setprot_top:
1088         raddr = (caddr_t)((uintptr_t)addr & (uintptr_t)PAGEMASK);
1089         rsize = (((size_t)(addr + size) + PAGEOFFSET) & PAGEMASK) -
1090             (size_t)raddr;

1092         if (raddr + rsize < raddr)              /* check for wraparound */
1093                 return (ENOMEM);

1095         saveraddr = raddr;
1096         saversize = rsize;

1098         /*
1099          * Normally we only lock the as as a reader. But
1100          * if due to setprot the segment driver needs to split
1101          * a segment it will return IE_RETRY. Therefore we re-acquire
1102          * the as lock as a writer so the segment driver can change
1103          * the seg list. Also the segment driver will return IE_RETRY
1104          * after it has changed the segment list so we therefore keep
1105          * locking as a writer. Since these opeartions should be rare
1106          * want to only lock as a writer when necessary.
1107          */
1108         if (writer || avl_numnodes(&as->a_wpage) != 0) {
1109                 AS_LOCK_ENTER(as, &as->a_lock, RW_WRITER);
1110         } else {
1111                 AS_LOCK_ENTER(as, &as->a_lock, RW_READER);
1112         }

1114         as_clearwatchprot(as, raddr, rsize);
1115         seg = as_segat(as, raddr);
1116         if (seg == NULL) {
1117                 as_setwatch(as);
1118                 AS_LOCK_EXIT(as, &as->a_lock);
1119                 return (ENOMEM);
1120         }

1122         for (; rsize != 0; rsize -= ssize, raddr += ssize) {
1123                 if (raddr >= seg->s_base + seg->s_size) {
1124                         seg = AS_SEGNEXT(as, seg);
1125                         if (seg == NULL || raddr != seg->s_base) {
```



```
1126                                 error = ENOMEM;
1127                                 break;
1128                         }
1129                 }
1130                 if ((raddr + rsize) > (seg->s_base + seg->s_size))
1131                         ssize = seg->s_base + seg->s_size - raddr;
1132                 else
1133                         ssize = rsize;
1134 retry:
1135                 error = segop_setprot(seg, raddr, ssize, prot);
1135                 error = SEGOP_SETPROT(seg, raddr, ssize, prot);

1137                 if (error == IE_NOMEM) {
1138                         error = EAGAIN;
1139                         break;
1140                 }

1142                 if (error == IE_RETRY) {
1143                         AS_LOCK_EXIT(as, &as->a_lock);
1144                         writer = 1;
1145                         goto setprot_top;
1146                 }

1148                 if (error == EAGAIN) {
1149                         /*
1150                          * Make sure we have a_lock as writer.
1151                          */
1152                         if (writer == 0) {
1153                                 AS_LOCK_EXIT(as, &as->a_lock);
1154                                 writer = 1;
1155                                 goto setprot_top;
1156                         }

1158                         /*
1159                          * Memory is currently locked.  It must be unlocked
1160                          * before this operation can succeed through a retry.
1161                          * The possible reasons for locked memory and
1162                          * corresponding strategies for unlocking are:
1163                          * (1) Normal I/O
1164                          *      wait for a signal that the I/O operation
1165                          *      has completed and the memory is unlocked.
1166                          * (2) Asynchronous I/O
1167                          *      The aio subsystem does not unlock pages when
1168                          *      the I/O is completed. Those pages are unlocked
1169                          *      when the application calls aiowait/aioerror.
1170                          *      So, to prevent blocking forever, cv_broadcast()
1171                          *      is done to wake up aio_cleanup_thread.
1172                          *      Subsequently, segvn_reclaim will be called, and
1173                          *      that will do AS_CLRUNMAPWAIT() and wake us up.
1174                          * (3) Long term page locking:
1175                          *      Drivers intending to have pages locked for a
1176                          *      period considerably longer than for normal I/O
1177                          *      (essentially forever) may have registered for a
1178                          *      callback so they may unlock these pages on
1179                          *      request. This is needed to allow this operation
1180                          *      to succeed. Each entry on the callback list is
1181                          *      examined. If the event or address range pertains
1182                          *      the callback is invoked (unless it already is in
1183                          *      progress). The a_contents lock must be dropped
1184                          *      before the callback, so only one callback can
1185                          *      be done at a time. Go to the top and do more
1186                          *      until zero is returned. If zero is returned,
1187                          *      either there were no callbacks for this event
1188                          *      or they were already in progress.
1189                          */
1190                         mutex_enter(&as->a_contents);
```

```
1191                        if (as->a_callbacks &&
1192                            (cb = as_find_callback(as, AS_SETPROT_EVENT,
1193                            seg->s_base, seg->s_size))) {
1194                                AS_LOCK_EXIT(as, &as->a_lock);
1195                                as_execute_callback(as, cb, AS_SETPROT_EVENT);
1196                        } else if (!AS_ISNOUNMAPWAIT(as)) {
1197                                if (AS_ISUNMAPWAIT(as) == 0)
1198                                        cv_broadcast(&as->a_cv);
1199                                AS_SETUNMAPWAIT(as);
1200                                AS_LOCK_EXIT(as, &as->a_lock);
1201                                while (AS_ISUNMAPWAIT(as))
1202                                        cv_wait(&as->a_cv, &as->a_contents);
1203                        } else {
1204                                /*
1205                                 * We may have raced with
1206                                 * segvn_reclaim()/segspt_reclaim(). In this
1207                                 * case clean nounmapwait flag and retry since
1208                                 * softlockcnt in this segment may be already
1209                                 * 0.  We don't drop as writer lock so our
1210                                 * number of retries without sleeping should
1211                                 * be very small. See segvn_reclaim() for
1212                                 * more comments.
1213                                 */
1214                                AS_CLRNOUNMAPWAIT(as);
1215                                mutex_exit(&as->a_contents);
1216                                goto retry;
1217                        }
1218                        mutex_exit(&as->a_contents);
1219                        goto setprot_top;
1220                } else if (error != 0)
1221                        break;
1222        }
1223        if (error != 0) {
1224                as_setwatch(as);
1225        } else {
1226                as_setwatchprot(as, saveraddr, saversize, prot);
1227        }
1228        AS_LOCK_EXIT(as, &as->a_lock);
1229        return (error);
1230 }

1232 /*
1233  * Check to make sure that the interval [addr, addr + size)
1234  * in address space 'as' has at least the specified protection.
1235  * It is ok for the range to cross over several segments, as long
1236  * as they are contiguous.
1237  */
1238 int
1239 as_checkprot(struct as *as, caddr_t addr, size_t size, uint_t prot)
1240 {
1241        struct seg *seg;
1242        size_t ssize;
1243        caddr_t raddr;                  /* rounded down addr */
1244        size_t rsize;                   /* rounded up size */
1245        int error = 0;

1247        raddr = (caddr_t)((uintptr_t)addr & (uintptr_t)PAGEMASK);
1248        rsize = (((size_t)(addr + size) + PAGEOFFSET) & PAGEMASK) -
1249            (size_t)raddr;

1251        if (raddr + rsize < raddr)              /* check for wraparound */
1252                return (ENOMEM);

1254        /*
1255         * This is ugly as sin...
1256         * Normally, we only acquire the address space readers lock.
```



```
1257         * However, if the address space has watchpoints present,
1258         * we must acquire the writer lock on the address space for
1259         * the benefit of as_clearwatchprot() and as_setwatchprot().
1260         */
1261        if (avl_numnodes(&as->a_wpage) != 0)
1262                AS_LOCK_ENTER(as, &as->a_lock, RW_WRITER);
1263        else
1264                AS_LOCK_ENTER(as, &as->a_lock, RW_READER);
1265        as_clearwatchprot(as, raddr, rsize);
1266        seg = as_segat(as, raddr);
1267        if (seg == NULL) {
1268                as_setwatch(as);
1269                AS_LOCK_EXIT(as, &as->a_lock);
1270                return (ENOMEM);
1271        }

1273        for (; rsize != 0; rsize -= ssize, raddr += ssize) {
1274                if (raddr >= seg->s_base + seg->s_size) {
1275                        seg = AS_SEGNEXT(as, seg);
1276                        if (seg == NULL || raddr != seg->s_base) {
1277                                error = ENOMEM;
1278                                break;
1279                        }
1280                }
1281                if ((raddr + rsize) > (seg->s_base + seg->s_size))
1282                        ssize = seg->s_base + seg->s_size - raddr;
1283                else
1284                        ssize = rsize;

1286                **error = segop_checkprot(seg, raddr, ssize, prot);**
1286                *error = SEGOP_CHECKPROT(seg, raddr, ssize, prot);*
1287                if (error != 0)
1288                        break;
1289        }
1290        as_setwatch(as);
1291        AS_LOCK_EXIT(as, &as->a_lock);
1292        return (error);
1293 }

1295 int
1296 as_unmap(struct as *as, caddr_t addr, size_t size)
1297 {
1298        struct seg *seg, *seg_next;
1299        struct as_callback *cb;
1300        caddr_t raddr, eaddr;
1301        size_t ssize, rsize = 0;
1302        int err;

1304 top:
1305        raddr = (caddr_t)((uintptr_t)addr & (uintptr_t)PAGEMASK);
1306        eaddr = (caddr_t)(((uintptr_t)(addr + size) + PAGEOFFSET) &
1307            (uintptr_t)PAGEMASK);

1309        AS_LOCK_ENTER(as, &as->a_lock, RW_WRITER);

1311        as->a_updatedir = 1;    /* inform /proc */
1312        gethrestime(&as->a_updatetime);

1314        /*
1315         * Use as_findseg to find the first segment in the range, then
1316         * step through the segments in order, following s_next.
1317         */
1318        as_clearwatchprot(as, raddr, eaddr - raddr);

1320        for (seg = as_findseg(as, raddr, 0); seg != NULL; seg = seg_next) {
1321                if (eaddr <= seg->s_base)
```

```
1322                            break;          /* eaddr was in a gap; all done */
1323
1324                    /* this is implied by the test above */
1325                    ASSERT(raddr < eaddr);
1326
1327                    if (raddr < seg->s_base)
1328                            raddr = seg->s_base;    /* raddr was in a gap */
1329
1330                    if (eaddr > (seg->s_base + seg->s_size))
1331                            ssize = seg->s_base + seg->s_size - raddr;
1332                    else
1333                            ssize = eaddr - raddr;
1334
1335                    /*
1336                     * Save next segment pointer since seg can be
1337                     * destroyed during the segment unmap operation.
1338                     */
1339                    seg_next = AS_SEGNEXT(as, seg);
1340
1341                    /*
1342                     * We didn't count /dev/null mappings, so ignore them here.
1343                     * We'll handle MAP_NORESERVE cases in segvn_unmap(). (Again,
1344                     * we have to do this check here while we have seg.)
1345                     */
1346                    rsize = 0;
1347                    if (!SEG_IS_DEVNULL_MAPPING(seg) &&
1348                        !SEG_IS_PARTIAL_RESV(seg))
1349                            rsize = ssize;
1350
1351 retry:
1352                    err = segop_unmap(seg, raddr, ssize);
1352                    err = SEGOP_UNMAP(seg, raddr, ssize);
1353                    if (err == EAGAIN) {
1354                            /*
1355                             * Memory is currently locked.  It must be unlocked
1356                             * before this operation can succeed through a retry.
1357                             * The possible reasons for locked memory and
1358                             * corresponding strategies for unlocking are:
1359                             * (1) Normal I/O
1360                             *      wait for a signal that the I/O operation
1361                             *      has completed and the memory is unlocked.
1362                             * (2) Asynchronous I/O
1363                             *      The aio subsystem does not unlock pages when
1364                             *      the I/O is completed. Those pages are unlocked
1365                             *      when the application calls aiowait/aioerror.
1366                             *      So, to prevent blocking forever, cv_broadcast()
1367                             *      is done to wake up aio_cleanup_thread.
1368                             *      Subsequently, segvn_reclaim will be called, and
1369                             *      that will do AS_CLRUNMAPWAIT() and wake us up.
1370                             * (3) Long term page locking:
1371                             *      Drivers intending to have pages locked for a
1372                             *      period considerably longer than for normal I/O
1373                             *      (essentially forever) may have registered for a
1374                             *      callback so they may unlock these pages on
1375                             *      request. This is needed to allow this operation
1376                             *      to succeed. Each entry on the callback list is
1377                             *      examined. If the event or address range pertains
1378                             *      the callback is invoked (unless it already is in
1379                             *      progress). The a_contents lock must be dropped
1380                             *      before the callback, so only one callback can
1381                             *      be done at a time. Go to the top and do more
1382                             *      until zero is returned. If zero is returned,
1383                             *      either there were no callbacks for this event
1384                             *      or they were already in progress.
1385                             */
1386                            mutex_enter(&as->a_contents);
```



```
1387                            if (as->a_callbacks &&
1388                                (cb = as_find_callback(as, AS_UNMAP_EVENT,
1389                                seg->s_base, seg->s_size))) {
1390                                    AS_LOCK_EXIT(as, &as->a_lock);
1391                                    as_execute_callback(as, cb, AS_UNMAP_EVENT);
1392                            } else if (!AS_ISNOUNMAPWAIT(as)) {
1393                                    if (AS_ISUNMAPWAIT(as) == 0)
1394                                            cv_broadcast(&as->a_cv);
1395                                    AS_SETUNMAPWAIT(as);
1396                                    AS_LOCK_EXIT(as, &as->a_lock);
1397                                    while (AS_ISUNMAPWAIT(as))
1398                                            cv_wait(&as->a_cv, &as->a_contents);
1399                            } else {
1400                                    /*
1401                                     * We may have raced with
1402                                     * segvn_reclaim()/segspt_reclaim(). In this
1403                                     * case clean nounmapwait flag and retry since
1404                                     * softlockcnt in this segment may be already
1405                                     * 0.  We don't drop as writer lock so our
1406                                     * number of retries without sleeping should
1407                                     * be very small. See segvn_reclaim() for
1408                                     * more comments.
1409                                     */
1410                                    AS_CLRNOUNMAPWAIT(as);
1411                                    mutex_exit(&as->a_contents);
1412                                    goto retry;
1413                            }
1414                            mutex_exit(&as->a_contents);
1415                            goto top;
1416                    } else if (err == IE_RETRY) {
1417                            AS_LOCK_EXIT(as, &as->a_lock);
1418                            goto top;
1419                    } else if (err) {
1420                            as_setwatch(as);
1421                            AS_LOCK_EXIT(as, &as->a_lock);
1422                            return (-1);
1423                    }
1424
1425                    as->a_size -= ssize;
1426                    if (rsize)
1427                            as->a_resvsize -= rsize;
1428                    raddr += ssize;
1429            }
1430            AS_LOCK_EXIT(as, &as->a_lock);
1431            return (0);
1432 }
```

_____***unchanged_portion_omitted_***

```
1765 /*
1766  * Delete all segments in the address space marked with S_PURGE.
1767  * This is currently used for Sparc V9 nofault ASI segments (seg_nf.c).
1768  * These segments are deleted as a first step before calls to as_gap(), so
1769  * that they don't affect mmap() or shmat().
1770  */
1771 void
1772 as_purge(struct as *as)
1773 {
1774            struct seg *seg;
1775            struct seg *next_seg;
1776
1777            /*
1778             * the setting of NEEDSPURGE is protect by as_rangelock(), so
1779             * no need to grab a_contents mutex for this check
1780             */
1781            if ((as->a_flags & AS_NEEDSPURGE) == 0)
```

```
1782                 return;

1784         AS_LOCK_ENTER(as, &as->a_lock, RW_WRITER);
1785         next_seg = NULL;
1786         seg = AS_SEGFIRST(as);
1787         while (seg != NULL) {
1788                 next_seg = AS_SEGNEXT(as, seg);
1789                 if (seg->s_flags & S_PURGE)
1790                         segop_unmap(seg, seg->s_base, seg->s_size);
1790                         SEGOP_UNMAP(seg, seg->s_base, seg->s_size);
1791                 seg = next_seg;
1792         }
1793         AS_LOCK_EXIT(as, &as->a_lock);

1795         mutex_enter(&as->a_contents);
1796         as->a_flags &= ~AS_NEEDSPURGE;
1797         mutex_exit(&as->a_contents);
1798 }
```

*unchanged_portion_omitted_*

```
2060 /*
2061  * Determine whether data from the mappings in interval [addr, addr + size)
2062  * are in the primary memory (core) cache.
2063  */
2064 int
2065 as_incore(struct as *as, caddr_t addr,
2066     size_t size, char *vec, size_t *sizep)
2067 {
2068         struct seg *seg;
2069         size_t ssize;
2070         caddr_t raddr;          /* rounded down addr */
2071         size_t rsize;           /* rounded up size */
2072         size_t isize;                   /* iteration size */
2073         int error = 0;          /* result, assume success */

2075         *sizep = 0;
2076         raddr = (caddr_t)((uintptr_t)addr & (uintptr_t)PAGEMASK);
2077         rsize = ((((size_t)addr + size) + PAGEOFFSET) & PAGEMASK) -
2078             (size_t)raddr;

2080         if (raddr + rsize < raddr)              /* check for wraparound */
2081                 return (ENOMEM);

2083         AS_LOCK_ENTER(as, &as->a_lock, RW_READER);
2084         seg = as_segat(as, raddr);
2085         if (seg == NULL) {
2086                 AS_LOCK_EXIT(as, &as->a_lock);
2087                 return (-1);
2088         }

2090         for (; rsize != 0; rsize -= ssize, raddr += ssize) {
2091                 if (raddr >= seg->s_base + seg->s_size) {
2092                         seg = AS_SEGNEXT(as, seg);
2093                         if (seg == NULL || raddr != seg->s_base) {
2094                                 error = -1;
2095                                 break;
2096                         }
2097                 }
2098                 if ((raddr + rsize) > (seg->s_base + seg->s_size))
2099                         ssize = seg->s_base + seg->s_size - raddr;
2100                 else
2101                         ssize = rsize;
2102                 *sizep += isize = segop_incore(seg, raddr, ssize, vec);
2102                 *sizep += isize = SEGOP_INCORE(seg, raddr, ssize, vec);
2103                 if (isize != ssize) {
2104                         error = -1;
```



```
2105                         break;
2106                 }
2107                 vec += btopr(ssize);
2108         }
2109         AS_LOCK_EXIT(as, &as->a_lock);
2110         return (error);
2111 }

2113 static void
2114 as_segunlock(struct seg *seg, caddr_t addr, int attr,
2115         ulong_t *bitmap, size_t position, size_t npages)
2116 {
2117         caddr_t range_start;
2118         size_t  pos1 = position;
2119         size_t  pos2;
2120         size_t  size;
2121         size_t  end_pos = npages + position;

2123         while (bt_range(bitmap, &pos1, &pos2, end_pos)) {
2124                 size = ptob((pos2 - pos1));
2125                 range_start = (caddr_t)((uintptr_t)addr +
2126                     ptob(pos1 - position));

2128                 (void) segop_lockop(seg, range_start, size, attr, MC_UNLOCK,
2128                 (void) SEGOP_LOCKOP(seg, range_start, size, attr, MC_UNLOCK,
2129                     (ulong_t *)NULL, (size_t)NULL);
2130                 pos1 = pos2;
2131         }
2132 }
```

*unchanged_portion_omitted_*

```
2157 /*
2158  * Cache control operations over the interval [addr, addr + size) in
2159  * address space "as".
2160  */
2161 /*ARGSUSED*/
2162 int
2163 as_ctl(struct as *as, caddr_t addr, size_t size, int func, int attr,
2164     uintptr_t arg, ulong_t *lock_map, size_t pos)
2165 {
2166         struct seg *seg;        /* working segment */
2167         caddr_t raddr;          /* rounded down addr */
2168         caddr_t initraddr;      /* saved initial rounded down addr */
2169         size_t rsize;           /* rounded up size */
2170         size_t initrsize;       /* saved initial rounded up size */
2171         size_t ssize;           /* size of seg */
2172         int error = 0;                  /* result */
2173         size_t mlock_size;      /* size of bitmap */
2174         ulong_t *mlock_map;     /* pointer to bitmap used */
2175                                 /* to represent the locked */
2176                                 /* pages. */
2177 retry:
2178         if (error == IE_RETRY)
2179                 AS_LOCK_ENTER(as, &as->a_lock, RW_WRITER);
2180         else
2181                 AS_LOCK_ENTER(as, &as->a_lock, RW_READER);

2183         /*
2184          * If these are address space lock/unlock operations, loop over
2185          * all segments in the address space, as appropriate.
2186          */
2187         if (func == MC_LOCKAS) {
2188                 size_t npages, idx;
2189                 size_t rlen = 0;        /* rounded as length */

2191                 idx = pos;
```

```
2193                if (arg & MCL_FUTURE) {
2194                        mutex_enter(&as->a_contents);
2195                        AS_SETPGLCK(as);
2196                        mutex_exit(&as->a_contents);
2197                }
2198                if ((arg & MCL_CURRENT) == 0) {
2199                        AS_LOCK_EXIT(as, &as->a_lock);
2200                        return (0);
2201                }

2203                seg = AS_SEGFIRST(as);
2204                if (seg == NULL) {
2205                        AS_LOCK_EXIT(as, &as->a_lock);
2206                        return (0);
2207                }

2209                do {
2210                        raddr = (caddr_t)((uintptr_t)seg->s_base &
2211                            (uintptr_t)PAGEMASK);
2212                        rlen += (((uintptr_t)(seg->s_base + seg->s_size) +
2213                            PAGEOFFSET) & PAGEMASK) - (uintptr_t)raddr;
2214                } while ((seg = AS_SEGNEXT(as, seg)) != NULL);

2216                mlock_size = BT_BITOUL(btopr(rlen));
2217                if ((mlock_map = (ulong_t *)kmem_zalloc(mlock_size *
2218                    sizeof (ulong_t), KM_NOSLEEP)) == NULL) {
2219                                AS_LOCK_EXIT(as, &as->a_lock);
2220                                return (EAGAIN);
2221                }

2223                for (seg = AS_SEGFIRST(as); seg; seg = AS_SEGNEXT(as, seg)) {
2224                        error = segop_lockop(seg, seg->s_base,
2224                        error = SEGOP_LOCKOP(seg, seg->s_base,
2225                            seg->s_size, attr, MC_LOCK, mlock_map, pos);
2226                        if (error != 0)
2227                                break;
2228                        pos += seg_pages(seg);
2229                }

2231                if (error) {
2232                        for (seg = AS_SEGFIRST(as); seg != NULL;
2233                            seg = AS_SEGNEXT(as, seg)) {

2235                                raddr = (caddr_t)((uintptr_t)seg->s_base &
2236                                    (uintptr_t)PAGEMASK);
2237                                npages = seg_pages(seg);
2238                                as_segunlock(seg, raddr, attr, mlock_map,
2239                                    idx, npages);
2240                                idx += npages;
2241                        }
2242                }

2244                kmem_free(mlock_map, mlock_size * sizeof (ulong_t));
2245                AS_LOCK_EXIT(as, &as->a_lock);
2246                goto lockerr;
2247        } else if (func == MC_UNLOCKAS) {
2248                mutex_enter(&as->a_contents);
2249                AS_CLRPGLCK(as);
2250                mutex_exit(&as->a_contents);

2252                for (seg = AS_SEGFIRST(as); seg; seg = AS_SEGNEXT(as, seg)) {
2253                        error = segop_lockop(seg, seg->s_base,
2253                        error = SEGOP_LOCKOP(seg, seg->s_base,
2254                            seg->s_size, attr, MC_UNLOCK, NULL, 0);
2255                        if (error != 0)
```



```
2256                                break;
2257                }

2259                AS_LOCK_EXIT(as, &as->a_lock);
2260                goto lockerr;
2261        }

2263        /*
2264         * Normalize addresses and sizes.
2265         */
2266        initraddr = raddr = (caddr_t)((uintptr_t)addr & (uintptr_t)PAGEMASK);
2267        initrsize = rsize = (((size_t)(addr + size) + PAGEOFFSET) & PAGEMASK) -
2268            (size_t)raddr;

2270        if (raddr + rsize < raddr) {            /* check for wraparound */
2271                AS_LOCK_EXIT(as, &as->a_lock);
2272                return (ENOMEM);
2273        }

2275        /*
2276         * Get initial segment.
2277         */
2278        if ((seg = as_segat(as, raddr)) == NULL) {
2279                AS_LOCK_EXIT(as, &as->a_lock);
2280                return (ENOMEM);
2281        }

2283        if (func == MC_LOCK) {
2284                mlock_size = BT_BITOUL(btopr(rsize));
2285                if ((mlock_map = (ulong_t *)kmem_zalloc(mlock_size *
2286                    sizeof (ulong_t), KM_NOSLEEP)) == NULL) {
2287                                AS_LOCK_EXIT(as, &as->a_lock);
2288                                return (EAGAIN);
2289                }
2290        }

2292        /*
2293         * Loop over all segments.  If a hole in the address range is
2294         * discovered, then fail.  For each segment, perform the appropriate
2295         * control operation.
2296         */
2297        while (rsize != 0) {

2299                /*
2300                 * Make sure there's no hole, calculate the portion
2301                 * of the next segment to be operated over.
2302                 */
2303                if (raddr >= seg->s_base + seg->s_size) {
2304                        seg = AS_SEGNEXT(as, seg);
2305                        if (seg == NULL || raddr != seg->s_base) {
2306                                if (func == MC_LOCK) {
2307                                        as_unlockerr(as, attr, mlock_map,
2308                                            initraddr, initrsize - rsize);
2309                                        kmem_free(mlock_map,
2310                                            mlock_size * sizeof (ulong_t));
2311                                }
2312                                AS_LOCK_EXIT(as, &as->a_lock);
2313                                return (ENOMEM);
2314                        }
2315                }
2316                if ((raddr + rsize) > (seg->s_base + seg->s_size))
2317                        ssize = seg->s_base + seg->s_size - raddr;
2318                else
2319                        ssize = rsize;

2321                /*
```

```
2322                  * Dispatch on specific function.
2323                  */
2324                 switch (func) {

2326                 /*
2327                  * Synchronize cached data from mappings with backing
2328                  * objects.
2329                  */
2330                 case MC_SYNC:
2331                         if (error = segop_sync(seg, raddr, ssize,
2331                         if (error = SEGOP_SYNC(seg, raddr, ssize,
2332                             attr, (uint_t)arg)) {
2333                                 AS_LOCK_EXIT(as, &as->a_lock);
2334                                 return (error);
2335                         }
2336                         break;

2338                 /*
2339                  * Lock pages in memory.
2340                  */
2341                 case MC_LOCK:
2342                         if (error = segop_lockop(seg, raddr, ssize,
2342                         if (error = SEGOP_LOCKOP(seg, raddr, ssize,
2343                             attr, func, mlock_map, pos)) {
2344                                 as_unlockerr(as, attr, mlock_map, initraddr,
2345                                     initrsize - rsize + ssize);
2346                                 kmem_free(mlock_map, mlock_size *
2347                                     sizeof (ulong_t));
2348                                 AS_LOCK_EXIT(as, &as->a_lock);
2349                                 goto lockerr;
2350                         }
2351                         break;

2353                 /*
2354                  * Unlock mapped pages.
2355                  */
2356                 case MC_UNLOCK:
2357                         (void) segop_lockop(seg, raddr, ssize, attr, func,
2357                         (void) SEGOP_LOCKOP(seg, raddr, ssize, attr, func,
2358                             (ulong_t *)NULL, (size_t)NULL);
2359                         break;

2361                 /*
2362                  * Store VM advise for mapped pages in segment layer.
2363                  */
2364                 case MC_ADVISE:
2365                         error = segop_advise(seg, raddr, ssize, (uint_t)arg);
2365                         error = SEGOP_ADVISE(seg, raddr, ssize, (uint_t)arg);

2367                         /*
2368                          * Check for regular errors and special retry error
2369                          */
2370                         if (error) {
2371                                 if (error == IE_RETRY) {
2372                                         /*
2373                                          * Need to acquire writers lock, so
2374                                          * have to drop readers lock and start
2375                                          * all over again
2376                                          */
2377                                         AS_LOCK_EXIT(as, &as->a_lock);
2378                                         goto retry;
2379                                 } else if (error == IE_REATTACH) {
2380                                         /*
2381                                          * Find segment for current address
2382                                          * because current segment just got
2383                                          * split or concatenated
```



```
2384                                          */
2385                                         seg = as_segat(as, raddr);
2386                                         if (seg == NULL) {
2387                                                 AS_LOCK_EXIT(as, &as->a_lock);
2388                                                 return (ENOMEM);
2389                                         }
2390                                 } else {
2391                                         /*
2392                                          * Regular error
2393                                          */
2394                                         AS_LOCK_EXIT(as, &as->a_lock);
2395                                         return (error);
2396                                 }
2397                         }
2398                         break;

2400                 case MC_INHERIT_ZERO:
2401                         if (seg->s_ops->inherit == NULL) {
2402                                 error = ENOTSUP;
2403                         } else {
2404                                 error = segop_inherit(seg, raddr, ssize,
2404                                 error = SEGOP_INHERIT(seg, raddr, ssize,
2405                                     SEGP_INH_ZERO);
2406                         }
2407                         if (error != 0) {
2408                                 AS_LOCK_EXIT(as, &as->a_lock);
2409                                 return (error);
2410                         }
2411                         break;

2413                 /*
2414                  * Can't happen.
2415                  */
2416                 default:
2417                         panic("as_ctl: bad operation %d", func);
2418                         /*NOTREACHED*/
2419                 }

2421                 rsize -= ssize;
2422                 raddr += ssize;
2423         }

2425         if (func == MC_LOCK)
2426                 kmem_free(mlock_map, mlock_size * sizeof (ulong_t));
2427         AS_LOCK_EXIT(as, &as->a_lock);
2428         return (0);
2429 lockerr:

2431         /*
2432          * If the lower levels returned EDEADLK for a segment lockop,
2433          * it means that we should retry the operation.  Let's wait
2434          * a bit also to let the deadlock causing condition clear.
2435          * This is part of a gross hack to work around a design flaw
2436          * in the ufs/sds logging code and should go away when the
2437          * logging code is re-designed to fix the problem. See bug
2438          * 4125102 for details of the problem.
2439          */
2440         if (error == EDEADLK) {
2441                 delay(deadlk_wait);
2442                 error = 0;
2443                 goto retry;
2444         }
2445         return (error);
2446 }
```

_____***unchanged_portion_omitted_***

```
2467 /*
2468  * Pagelock pages from a range that spans more than 1 segment.  Obtain shadow
2469  * lists from each segment and copy them to one contiguous shadow list (plist)
2470  * as expected by the caller.  Save pointers to per segment shadow lists at
2471  * the tail of plist so that they can be used during as_pageunlock().
2472  */
2473 static int
2474 as_pagelock_segs(struct as *as, struct seg *seg, struct page ***ppp,
2475     caddr_t addr, size_t size, enum seg_rw rw)
2476 {
2477         caddr_t sv_addr = addr;
2478         size_t sv_size = size;
2479         struct seg *sv_seg = seg;
2480         ulong_t segcnt = 1;
2481         ulong_t cnt;
2482         size_t ssize;
2483         pgcnt_t npages = btop(size);
2484         page_t **plist;
2485         page_t **pl;
2486         int error;
2487         caddr_t eaddr;
2488         faultcode_t fault_err = 0;
2489         pgcnt_t pl_off;
2490         extern struct seg_ops segspt_shmops;

2492         ASSERT(AS_LOCK_HELD(as, &as->a_lock));
2493         ASSERT(seg != NULL);
2494         ASSERT(addr >= seg->s_base && addr < seg->s_base + seg->s_size);
2495         ASSERT(addr + size > seg->s_base + seg->s_size);
2496         ASSERT(IS_P2ALIGNED(size, PAGESIZE));
2497         ASSERT(IS_P2ALIGNED(addr, PAGESIZE));

2499         /*
2500          * Count the number of segments covered by the range we are about to
2501          * lock. The segment count is used to size the shadow list we return
2502          * back to the caller.
2503          */
2504         for (; size != 0; size -= ssize, addr += ssize) {
2505                 if (addr >= seg->s_base + seg->s_size) {

2507                         seg = AS_SEGNEXT(as, seg);
2508                         if (seg == NULL || addr != seg->s_base) {
2509                                 AS_LOCK_EXIT(as, &as->a_lock);
2510                                 return (EFAULT);
2511                         }
2512                         /*
2513                          * Do a quick check if subsequent segments
2514                          * will most likely support pagelock.
2515                          */
2516                         if (seg->s_ops == &segvn_ops) {
2517                                 vnode_t *vp;

2519                                 if (segop_getvp(seg, addr, &vp) != 0 ||
2519                                 if (SEGOP_GETVP(seg, addr, &vp) != 0 ||
2520                                     vp != NULL) {
2521                                         AS_LOCK_EXIT(as, &as->a_lock);
2522                                         goto slow;
2523                                 }
2524                         } else if (seg->s_ops != &segspt_shmops) {
2525                                 AS_LOCK_EXIT(as, &as->a_lock);
2526                                 goto slow;
2527                         }
2528                         segcnt++;
2529                 }
2530                 if (addr + size > seg->s_base + seg->s_size) {
2531                         ssize = seg->s_base + seg->s_size - addr;
```



```
2532                 } else {
2533                         ssize = size;
2534                 }
2535         }
2536         ASSERT(segcnt > 1);

2538         plist = kmem_zalloc((npages + segcnt) * sizeof (page_t *), KM_SLEEP);

2540         addr = sv_addr;
2541         size = sv_size;
2542         seg = sv_seg;

2544         for (cnt = 0, pl_off = 0; size != 0; size -= ssize, addr += ssize) {
2545                 if (addr >= seg->s_base + seg->s_size) {
2546                         seg = AS_SEGNEXT(as, seg);
2547                         ASSERT(seg != NULL && addr == seg->s_base);
2548                         cnt++;
2549                         ASSERT(cnt < segcnt);
2550                 }
2551                 if (addr + size > seg->s_base + seg->s_size) {
2552                         ssize = seg->s_base + seg->s_size - addr;
2553                 } else {
2554                         ssize = size;
2555                 }
2556                 pl = &plist[npages + cnt];
2557                 error = segop_pagelock(seg, addr, ssize, (page_t ***)pl,
2557                 error = SEGOP_PAGELOCK(seg, addr, ssize, (page_t ***)pl,
2558                     L_PAGELOCK, rw);
2559                 if (error) {
2560                         break;
2561                 }
2562                 ASSERT(plist[npages + cnt] != NULL);
2563                 ASSERT(pl_off + btop(ssize) <= npages);
2564                 bcopy(plist[npages + cnt], &plist[pl_off],
2565                     btop(ssize) * sizeof (page_t *));
2566                 pl_off += btop(ssize);
2567         }

2569         if (size == 0) {
2570                 AS_LOCK_EXIT(as, &as->a_lock);
2571                 ASSERT(cnt == segcnt - 1);
2572                 *ppp = plist;
2573                 return (0);
2574         }

2576         /*
2577          * one of pagelock calls failed. The error type is in error variable.
2578          * Unlock what we've locked so far and retry with F_SOFTLOCK if error
2579          * type is either EFAULT or ENOTSUP. Otherwise just return the error
2580          * back to the caller.
2581          */

2583         eaddr = addr;
2584         seg = sv_seg;

2586         for (cnt = 0, addr = sv_addr; addr < eaddr; addr += ssize) {
2587                 if (addr >= seg->s_base + seg->s_size) {
2588                         seg = AS_SEGNEXT(as, seg);
2589                         ASSERT(seg != NULL && addr == seg->s_base);
2590                         cnt++;
2591                         ASSERT(cnt < segcnt);
2592                 }
2593                 if (eaddr > seg->s_base + seg->s_size) {
2594                         ssize = seg->s_base + seg->s_size - addr;
2595                 } else {
2596                         ssize = eaddr - addr;
```

```
2597                }
2598                pl = &plist[npages + cnt];
2599                ASSERT(*pl != NULL);
2600                (void) segop_pagelock(seg, addr, ssize, (page_t ***)pl,
2600                (void) SEGOP_PAGELOCK(seg, addr, ssize, (page_t ***)pl,
2601                    L_PAGEUNLOCK, rw);
2602        }

2604        AS_LOCK_EXIT(as, &as->a_lock);

2606        kmem_free(plist, (npages + segcnt) * sizeof (page_t *));

2608        if (error != ENOTSUP && error != EFAULT) {
2609                return (error);
2610        }

2612 slow:
2613        /*
2614         * If we are here because pagelock failed due to the need to cow fault
2615         * in the pages we want to lock F_SOFTLOCK will do this job and in
2616         * next as_pagelock() call for this address range pagelock will
2617         * hopefully succeed.
2618         */
2619        fault_err = as_fault(as->a_hat, as, sv_addr, sv_size, F_SOFTLOCK, rw);
2620        if (fault_err != 0) {
2621                return (fc_decode(fault_err));
2622        }
2623        *ppp = NULL;

2625        return (0);
2626 }

2628 /*
2629  * lock pages in a given address space. Return shadow list. If
2630  * the list is NULL, the MMU mapping is also locked.
2631  */
2632 int
2633 as_pagelock(struct as *as, struct page ***ppp, caddr_t addr,
2634     size_t size, enum seg_rw rw)
2635 {
2636        size_t rsize;
2637        caddr_t raddr;
2638        faultcode_t fault_err;
2639        struct seg *seg;
2640        int err;

2642        TRACE_2(TR_FAC_PHYSIO, TR_PHYSIO_AS_LOCK_START,
2643            "as_pagelock_start: addr %p size %ld", addr, size);

2645        raddr = (caddr_t)((uintptr_t)addr & (uintptr_t)PAGEMASK);
2646        rsize = (((size_t)(addr + size) + PAGEOFFSET) & PAGEMASK) -
2647            (size_t)raddr;

2649        /*
2650         * if the request crosses two segments let
2651         * as_fault handle it.
2652         */
2653        AS_LOCK_ENTER(as, &as->a_lock, RW_READER);

2655        seg = as_segat(as, raddr);
2656        if (seg == NULL) {
2657                AS_LOCK_EXIT(as, &as->a_lock);
2658                return (EFAULT);
2659        }
2660        ASSERT(raddr >= seg->s_base && raddr < seg->s_base + seg->s_size);
2661        if (raddr + rsize > seg->s_base + seg->s_size) {
```



```
2662                return (as_pagelock_segs(as, seg, ppp, raddr, rsize, rw));
2663        }
2664        if (raddr + rsize <= raddr) {
2665                AS_LOCK_EXIT(as, &as->a_lock);
2666                return (EFAULT);
2667        }

2669        TRACE_2(TR_FAC_PHYSIO, TR_PHYSIO_SEG_LOCK_START,
2670            "seg_lock_1_start: raddr %p rsize %ld", raddr, rsize);

2672        /*
2673         * try to lock pages and pass back shadow list
2674         */
2675        err = segop_pagelock(seg, raddr, rsize, ppp, L_PAGELOCK, rw);
2675        err = SEGOP_PAGELOCK(seg, raddr, rsize, ppp, L_PAGELOCK, rw);

2677        TRACE_0(TR_FAC_PHYSIO, TR_PHYSIO_SEG_LOCK_END, "seg_lock_1_end");

2679        AS_LOCK_EXIT(as, &as->a_lock);

2681        if (err == 0 || (err != ENOTSUP && err != EFAULT)) {
2682                return (err);
2683        }

2685        /*
2686         * Use F_SOFTLOCK to lock the pages because pagelock failed either due
2687         * to no pagelock support for this segment or pages need to be cow
2688         * faulted in. If fault is needed F_SOFTLOCK will do this job for
2689         * this as_pagelock() call and in the next as_pagelock() call for the
2690         * same address range pagelock call will hopefull succeed.
2691         */
2692        fault_err = as_fault(as->a_hat, as, addr, size, F_SOFTLOCK, rw);
2693        if (fault_err != 0) {
2694                return (fc_decode(fault_err));
2695        }
2696        *ppp = NULL;

2698        TRACE_0(TR_FAC_PHYSIO, TR_PHYSIO_AS_LOCK_END, "as_pagelock_end");
2699        return (0);
2700 }

2702 /*
2703  * unlock pages locked by as_pagelock_segs().  Retrieve per segment shadow
2704  * lists from the end of plist and call pageunlock interface for each segment.
2705  * Drop as lock and free plist.
2706  */
2707 static void
2708 as_pageunlock_segs(struct as *as, struct seg *seg, caddr_t addr, size_t size,
2709     struct page **plist, enum seg_rw rw)
2710 {
2711        ulong_t cnt;
2712        caddr_t eaddr = addr + size;
2713        pgcnt_t npages = btop(size);
2714        size_t ssize;
2715        page_t **pl;

2717        ASSERT(AS_LOCK_HELD(as, &as->a_lock));
2718        ASSERT(seg != NULL);
2719        ASSERT(addr >= seg->s_base && addr < seg->s_base + seg->s_size);
2720        ASSERT(addr + size > seg->s_base + seg->s_size);
2721        ASSERT(IS_P2ALIGNED(size, PAGESIZE));
2722        ASSERT(IS_P2ALIGNED(addr, PAGESIZE));
2723        ASSERT(plist != NULL);

2725        for (cnt = 0; addr < eaddr; addr += ssize) {
2726                if (addr >= seg->s_base + seg->s_size) {
```

```
2727                        seg = AS_SEGNEXT(as, seg);
2728                        ASSERT(seg != NULL && addr == seg->s_base);
2729                        cnt++;
2730                }
2731                if (eaddr > seg->s_base + seg->s_size) {
2732                        ssize = seg->s_base + seg->s_size - addr;
2733                } else {
2734                        ssize = eaddr - addr;
2735                }
2736                pl = &plist[npages + cnt];
2737                ASSERT(*pl != NULL);
2738                (void) segop_pagelock(seg, addr, ssize, (page_t ***)pl,
2738                (void) SEGOP_PAGELOCK(seg, addr, ssize, (page_t ***)pl,
2739                    L_PAGEUNLOCK, rw);
2740        }
2741        ASSERT(cnt > 0);
2742        AS_LOCK_EXIT(as, &as->a_lock);

2744        cnt++;
2745        kmem_free(plist, (npages + cnt) * sizeof (page_t *));
2746 }

2748 /*
2749  * unlock pages in a given address range
2750  */
2751 void
2752 as_pageunlock(struct as *as, struct page **pp, caddr_t addr, size_t size,
2753     enum seg_rw rw)
2754 {
2755        struct seg *seg;
2756        size_t rsize;
2757        caddr_t raddr;

2759        TRACE_2(TR_FAC_PHYSIO, TR_PHYSIO_AS_UNLOCK_START,
2760            "as_pageunlock_start: addr %p size %ld", addr, size);

2762        /*
2763         * if the shadow list is NULL, as_pagelock was
2764         * falling back to as_fault
2765         */
2766        if (pp == NULL) {
2767                (void) as_fault(as->a_hat, as, addr, size, F_SOFTUNLOCK, rw);
2768                return;
2769        }

2771        raddr = (caddr_t)((uintptr_t)addr & (uintptr_t)PAGEMASK);
2772        rsize = (((size_t)(addr + size) + PAGEOFFSET) & PAGEMASK) -
2773            (size_t)raddr;

2775        AS_LOCK_ENTER(as, &as->a_lock, RW_READER);
2776        seg = as_segat(as, raddr);
2777        ASSERT(seg != NULL);

2779        TRACE_2(TR_FAC_PHYSIO, TR_PHYSIO_SEG_UNLOCK_START,
2780            "seg_unlock_start: raddr %p rsize %ld", raddr, rsize);

2782        ASSERT(raddr >= seg->s_base && raddr < seg->s_base + seg->s_size);
2783        if (raddr + rsize <= seg->s_base + seg->s_size) {
2784                segop_pagelock(seg, raddr, rsize, &pp, L_PAGEUNLOCK, rw);
2784                SEGOP_PAGELOCK(seg, raddr, rsize, &pp, L_PAGEUNLOCK, rw);
2785        } else {
2786                as_pageunlock_segs(as, seg, raddr, rsize, pp, rw);
2787                return;
2788        }
2789        AS_LOCK_EXIT(as, &as->a_lock);
2790        TRACE_0(TR_FAC_PHYSIO, TR_PHYSIO_AS_UNLOCK_END, "as_pageunlock_end");
```



```
2791 }

2793 int
2794 as_setpagesize(struct as *as, caddr_t addr, size_t size, uint_t szc,
2795     boolean_t wait)
2796 {
2797        struct seg *seg;
2798        size_t ssize;
2799        caddr_t raddr;                  /* rounded down addr */
2800        size_t rsize;                   /* rounded up size */
2801        int error = 0;
2802        size_t pgsz = page_get_pagesize(szc);

2804 setpgsz_top:
2805        if (!IS_P2ALIGNED(addr, pgsz) || !IS_P2ALIGNED(size, pgsz)) {
2806                return (EINVAL);
2807        }

2809        raddr = addr;
2810        rsize = size;

2812        if (raddr + rsize < raddr)              /* check for wraparound */
2813                return (ENOMEM);

2815        AS_LOCK_ENTER(as, &as->a_lock, RW_WRITER);
2816        as_clearwatchprot(as, raddr, rsize);
2817        seg = as_segat(as, raddr);
2818        if (seg == NULL) {
2819                as_setwatch(as);
2820                AS_LOCK_EXIT(as, &as->a_lock);
2821                return (ENOMEM);
2822        }

2824        for (; rsize != 0; rsize -= ssize, raddr += ssize) {
2825                if (raddr >= seg->s_base + seg->s_size) {
2826                        seg = AS_SEGNEXT(as, seg);
2827                        if (seg == NULL || raddr != seg->s_base) {
2828                                error = ENOMEM;
2829                                break;
2830                        }
2831                }
2832                if ((raddr + rsize) > (seg->s_base + seg->s_size)) {
2833                        ssize = seg->s_base + seg->s_size - raddr;
2834                } else {
2835                        ssize = rsize;
2836                }

2838 retry:
2839                error = segop_setpagesize(seg, raddr, ssize, szc);
2839                error = SEGOP_SETPAGESIZE(seg, raddr, ssize, szc);

2841                if (error == IE_NOMEM) {
2842                        error = EAGAIN;
2843                        break;
2844                }

2846                if (error == IE_RETRY) {
2847                        AS_LOCK_EXIT(as, &as->a_lock);
2848                        goto setpgsz_top;
2849                }

2851                if (error == ENOTSUP) {
2852                        error = EINVAL;
2853                        break;
2854                }
```

```
2856                 if (wait && (error == EAGAIN)) {
2857                         /*
2858                          * Memory is currently locked.  It must be unlocked
2859                          * before this operation can succeed through a retry.
2860                          * The possible reasons for locked memory and
2861                          * corresponding strategies for unlocking are:
2862                          * (1) Normal I/O
2863                          *      wait for a signal that the I/O operation
2864                          *      has completed and the memory is unlocked.
2865                          * (2) Asynchronous I/O
2866                          *      The aio subsystem does not unlock pages when
2867                          *      the I/O is completed. Those pages are unlocked
2868                          *      when the application calls aiowait/aioerror.
2869                          *      So, to prevent blocking forever, cv_broadcast()
2870                          *      is done to wake up aio_cleanup_thread.
2871                          *      Subsequently, segvn_reclaim will be called, and
2872                          *      that will do AS_CLRUNMAPWAIT() and wake us up.
2873                          * (3) Long term page locking:
2874                          *      This is not relevant for as_setpagesize()
2875                          *      because we cannot change the page size for
2876                          *      driver memory. The attempt to do so will
2877                          *      fail with a different error than EAGAIN so
2878                          *      there's no need to trigger as callbacks like
2879                          *      as_unmap, as_setprot or as_free would do.
2880                          */
2881                         mutex_enter(&as->a_contents);
2882                         if (!AS_ISNOUNMAPWAIT(as)) {
2883                                 if (AS_ISUNMAPWAIT(as) == 0) {
2884                                         cv_broadcast(&as->a_cv);
2885                                 }
2886                                 AS_SETUNMAPWAIT(as);
2887                                 AS_LOCK_EXIT(as, &as->a_lock);
2888                                 while (AS_ISUNMAPWAIT(as)) {
2889                                         cv_wait(&as->a_cv, &as->a_contents);
2890                                 }
2891                         } else {
2892                                 /*
2893                                  * We may have raced with
2894                                  * segvn_reclaim()/segspt_reclaim(). In this
2895                                  * case clean nounmapwait flag and retry since
2896                                  * softlockcnt in this segment may be already
2897                                  * 0.  We don't drop as writer lock so our
2898                                  * number of retries without sleeping should
2899                                  * be very small. See segvn_reclaim() for
2900                                  * more comments.
2901                                  */
2902                                 AS_CLRNOUNMAPWAIT(as);
2903                                 mutex_exit(&as->a_contents);
2904                                 goto retry;
2905                         }
2906                         mutex_exit(&as->a_contents);
2907                         goto setpgsz_top;
2908                 } else if (error != 0) {
2909                         break;
2910                 }
2911         }
2912         as_setwatch(as);
2913         AS_LOCK_EXIT(as, &as->a_lock);
2914         return (error);
2915 }

2917 /*
2918  * as_iset3_default_lpsize() just calls segop_setpagesize() on all segments
2918  * as_iset3_default_lpsize() just calls SEGOP_SETPAGESIZE() on all segments
2919  * in its chunk where s_szc is less than the szc we want to set.
2920  */
```



```
2921 static int
2922 as_iset3_default_lpsize(struct as *as, caddr_t raddr, size_t rsize, uint_t szc,
2923     int *retry)
2924 {
2925         struct seg *seg;
2926         size_t ssize;
2927         int error;

2929         ASSERT(AS_WRITE_HELD(as, &as->a_lock));

2931         seg = as_segat(as, raddr);
2932         if (seg == NULL) {
2933                 panic("as_iset3_default_lpsize: no seg");
2934         }

2936         for (; rsize != 0; rsize -= ssize, raddr += ssize) {
2937                 if (raddr >= seg->s_base + seg->s_size) {
2938                         seg = AS_SEGNEXT(as, seg);
2939                         if (seg == NULL || raddr != seg->s_base) {
2940                                 panic("as_iset3_default_lpsize: as changed");
2941                         }
2942                 }
2943                 if ((raddr + rsize) > (seg->s_base + seg->s_size)) {
2944                         ssize = seg->s_base + seg->s_size - raddr;
2945                 } else {
2946                         ssize = rsize;
2947                 }

2949                 if (szc > seg->s_szc) {
2950                         error = segop_setpagesize(seg, raddr, ssize, szc);
2950                         error = SEGOP_SETPAGESIZE(seg, raddr, ssize, szc);
2951                         /* Only retry on EINVAL segments that have no vnode. */
2952                         if (error == EINVAL) {
2953                                 vnode_t *vp = NULL;
2954                                 if ((segop_gettype(seg, raddr) & MAP_SHARED) &&
2955                                     (segop_getvp(seg, raddr, &vp) != 0 ||
2954                                 if ((SEGOP_GETTYPE(seg, raddr) & MAP_SHARED) &&
2955                                     (SEGOP_GETVP(seg, raddr, &vp) != 0 ||
2956                                     vp == NULL)) {
2957                                         *retry = 1;
2958                                 } else {
2959                                         *retry = 0;
2960                                 }
2961                         }
2962                         if (error) {
2963                                 return (error);
2964                         }
2965                 }
2966         }
2967         return (0);
2968 }
```

_____*unchanged_portion_omitted_*

```
3155 /*
3156  * Set the default large page size for the range. Called via memcntl with
3157  * page size set to 0. as_set_default_lpsize breaks the range down into
3158  * chunks with the same type/flags, ignores-non segvn segments, and passes
3159  * each chunk to as_iset_default_lpsize().
3160  */
3161 int
3162 as_set_default_lpsize(struct as *as, caddr_t addr, size_t size)
3163 {
3164         struct seg *seg;
3165         caddr_t raddr;
3166         size_t rsize;
3167         size_t ssize;
```

```
3168         int rtype, rflags;
3169         int stype, sflags;
3170         int error;
3171         caddr_t setaddr;
3172         size_t setsize;
3173         int segvn;

3175         if (size == 0)
3176                 return (0);

3178         AS_LOCK_ENTER(as, &as->a_lock, RW_WRITER);
3179 again:
3180         error = 0;

3182         raddr = (caddr_t)((uintptr_t)addr & (uintptr_t)PAGEMASK);
3183         rsize = (((size_t)(addr + size) + PAGEOFFSET) & PAGEMASK) -
3184             (size_t)raddr;

3186         if (raddr + rsize < raddr) {            /* check for wraparound */
3187                 AS_LOCK_EXIT(as, &as->a_lock);
3188                 return (ENOMEM);
3189         }
3190         as_clearwatchprot(as, raddr, rsize);
3191         seg = as_segat(as, raddr);
3192         if (seg == NULL) {
3193                 as_setwatch(as);
3194                 AS_LOCK_EXIT(as, &as->a_lock);
3195                 return (ENOMEM);
3196         }
3197         if (seg->s_ops == &segvn_ops) {
3198                 rtype = segop_gettype(seg, addr);
3198                 rtype = SEGOP_GETTYPE(seg, addr);
3199                 rflags = rtype & (MAP_TEXT | MAP_INITDATA);
3200                 rtype = rtype & (MAP_SHARED | MAP_PRIVATE);
3201                 segvn = 1;
3202         } else {
3203                 segvn = 0;
3204         }
3205         setaddr = raddr;
3206         setsize = 0;

3208         for (; rsize != 0; rsize -= ssize, raddr += ssize, setsize += ssize) {
3209                 if (raddr >= (seg->s_base + seg->s_size)) {
3210                         seg = AS_SEGNEXT(as, seg);
3211                         if (seg == NULL || raddr != seg->s_base) {
3212                                 error = ENOMEM;
3213                                 break;
3214                         }
3215                         if (seg->s_ops == &segvn_ops) {
3216                                 stype = segop_gettype(seg, raddr);
3216                                 stype = SEGOP_GETTYPE(seg, raddr);
3217                                 sflags = stype & (MAP_TEXT | MAP_INITDATA);
3218                                 stype &= (MAP_SHARED | MAP_PRIVATE);
3219                                 if (segvn && (rflags != sflags ||
3220                                     rtype != stype)) {
3221                                         /*
3222                                          * The next segment is also segvn but
3223                                          * has different flags and/or type.
3224                                          */
3225                                         ASSERT(setsize != 0);
3226                                         error = as_iset_default_lpsize(as,
3227                                             setaddr, setsize, rflags, rtype);
3228                                         if (error) {
3229                                                 break;
3230                                         }
3231                                         rflags = sflags;
```



```
3232                                         rtype = stype;
3233                                         setaddr = raddr;
3234                                         setsize = 0;
3235                                 } else if (!segvn) {
3236                                         rflags = sflags;
3237                                         rtype = stype;
3238                                         setaddr = raddr;
3239                                         setsize = 0;
3240                                         segvn = 1;
3241                                 }
3242                         } else if (segvn) {
3243                                 /* The next segment is not segvn. */
3244                                 ASSERT(setsize != 0);
3245                                 error = as_iset_default_lpsize(as,
3246                                     setaddr, setsize, rflags, rtype);
3247                                 if (error) {
3248                                         break;
3249                                 }
3250                                 segvn = 0;
3251                         }
3252                 }
3253                 if ((raddr + rsize) > (seg->s_base + seg->s_size)) {
3254                         ssize = seg->s_base + seg->s_size - raddr;
3255                 } else {
3256                         ssize = rsize;
3257                 }
3258         }
3259         if (error == 0 && segvn) {
3260                 /* The last chunk when rsize == 0. */
3261                 ASSERT(setsize != 0);
3262                 error = as_iset_default_lpsize(as, setaddr, setsize,
3263                     rflags, rtype);
3264         }

3266         if (error == IE_RETRY) {
3267                 goto again;
3268         } else if (error == IE_NOMEM) {
3269                 error = EAGAIN;
3270         } else if (error == ENOTSUP) {
3271                 error = EINVAL;
3272         } else if (error == EAGAIN) {
3273                 mutex_enter(&as->a_contents);
3274                 if (!AS_ISNOUNMAPWAIT(as)) {
3275                         if (AS_ISUNMAPWAIT(as) == 0) {
3276                                 cv_broadcast(&as->a_cv);
3277                         }
3278                         AS_SETUNMAPWAIT(as);
3279                         AS_LOCK_EXIT(as, &as->a_lock);
3280                         while (AS_ISUNMAPWAIT(as)) {
3281                                 cv_wait(&as->a_cv, &as->a_contents);
3282                         }
3283                         mutex_exit(&as->a_contents);
3284                         AS_LOCK_ENTER(as, &as->a_lock, RW_WRITER);
3285                 } else {
3286                         /*
3287                          * We may have raced with
3288                          * segvn_reclaim()/segspt_reclaim(). In this case
3289                          * clean nounmapwait flag and retry since softlockcnt
3290                          * in this segment may be already 0.  We don't drop as
3291                          * writer lock so our number of retries without
3292                          * sleeping should be very small. See segvn_reclaim()
3293                          * for more comments.
3294                          */
3295                         AS_CLRNOUNMAPWAIT(as);
3296                         mutex_exit(&as->a_contents);
3297                 }
```

```
3298                    goto again;
3299            }

3301            as_setwatch(as);
3302            AS_LOCK_EXIT(as, &as->a_lock);
3303            return (error);
3304    }

3306    /*
3307     * Setup all of the uninitialized watched pages that we can.
3308     */
3309    void
3310    as_setwatch(struct as *as)
3311    {
3312            struct watched_page *pwp;
3313            struct seg *seg;
3314            caddr_t vaddr;
3315            uint_t prot;
3316            int  err, retrycnt;

3318            if (avl_numnodes(&as->a_wpage) == 0)
3319                    return;

3321            ASSERT(AS_WRITE_HELD(as, &as->a_lock));

3323            for (pwp = avl_first(&as->a_wpage); pwp != NULL;
3324                pwp = AVL_NEXT(&as->a_wpage, pwp)) {
3325                    retrycnt = 0;
3326            retry:
3327                    vaddr = pwp->wp_vaddr;
3328                    if (pwp->wp_oprot != 0 ||       /* already set up */
3329                        (seg = as_segat(as, vaddr)) == NULL ||
3330                        segop_getprot(seg, vaddr, 0, &prot) != 0)
3330                        SEGOP_GETPROT(seg, vaddr, 0, &prot) != 0)
3331                            continue;

3333                    pwp->wp_oprot = prot;
3334                    if (pwp->wp_read)
3335                            prot &= ~(PROT_READ|PROT_WRITE|PROT_EXEC);
3336                    if (pwp->wp_write)
3337                            prot &= ~PROT_WRITE;
3338                    if (pwp->wp_exec)
3339                            prot &= ~(PROT_READ|PROT_WRITE|PROT_EXEC);
3340                    if (!(pwp->wp_flags & WP_NOWATCH) && prot != pwp->wp_oprot) {
3341                            err = segop_setprot(seg, vaddr, PAGESIZE, prot);
3341                            err = SEGOP_SETPROT(seg, vaddr, PAGESIZE, prot);
3342                            if (err == IE_RETRY) {
3343                                    pwp->wp_oprot = 0;
3344                                    ASSERT(retrycnt == 0);
3345                                    retrycnt++;
3346                                    goto retry;
3347                            }
3348                    }
3349                    pwp->wp_prot = prot;
3350            }
3351    }

3353    /*
3354     * Clear all of the watched pages in the address space.
3355     */
3356    void
3357    as_clearwatch(struct as *as)
3358    {
3359            struct watched_page *pwp;
3360            struct seg *seg;
3361            caddr_t vaddr;
```



```
3362            uint_t prot;
3363            int err, retrycnt;

3365            if (avl_numnodes(&as->a_wpage) == 0)
3366                    return;

3368            ASSERT(AS_WRITE_HELD(as, &as->a_lock));

3370            for (pwp = avl_first(&as->a_wpage); pwp != NULL;
3371                pwp = AVL_NEXT(&as->a_wpage, pwp)) {
3372                    retrycnt = 0;
3373            retry:
3374                    vaddr = pwp->wp_vaddr;
3375                    if (pwp->wp_oprot == 0 ||       /* not set up */
3376                        (seg = as_segat(as, vaddr)) == NULL)
3377                            continue;

3379                    if ((prot = pwp->wp_oprot) != pwp->wp_prot) {
3380                            err = segop_setprot(seg, vaddr, PAGESIZE, prot);
3380                            err = SEGOP_SETPROT(seg, vaddr, PAGESIZE, prot);
3381                            if (err == IE_RETRY) {
3382                                    ASSERT(retrycnt == 0);
3383                                    retrycnt++;
3384                                    goto retry;
3385                            }
3386                    }
3387                    pwp->wp_oprot = 0;
3388                    pwp->wp_prot = 0;
3389            }
3390    }

3392    /*
3393     * Force a new setup for all the watched pages in the range.
3394     */
3395    static void
3396    as_setwatchprot(struct as *as, caddr_t addr, size_t size, uint_t prot)
3397    {
3398            struct watched_page *pwp;
3399            struct watched_page tpw;
3400            caddr_t eaddr = addr + size;
3401            caddr_t vaddr;
3402            struct seg *seg;
3403            int err, retrycnt;
3404            uint_t  wprot;
3405            avl_index_t where;

3407            if (avl_numnodes(&as->a_wpage) == 0)
3408                    return;

3410            ASSERT(AS_WRITE_HELD(as, &as->a_lock));

3412            tpw.wp_vaddr = (caddr_t)((uintptr_t)addr & (uintptr_t)PAGEMASK);
3413            if ((pwp = avl_find(&as->a_wpage, &tpw, &where)) == NULL)
3414                    pwp = avl_nearest(&as->a_wpage, where, AVL_AFTER);

3416            while (pwp != NULL && pwp->wp_vaddr < eaddr) {
3417                    retrycnt = 0;
3418                    vaddr = pwp->wp_vaddr;

3420                    wprot = prot;
3421                    if (pwp->wp_read)
3422                            wprot &= ~(PROT_READ|PROT_WRITE|PROT_EXEC);
3423                    if (pwp->wp_write)
3424                            wprot &= ~PROT_WRITE;
3425                    if (pwp->wp_exec)
3426                            wprot &= ~(PROT_READ|PROT_WRITE|PROT_EXEC);
```

```
3427                 if (!(pwp->wp_flags & WP_NOWATCH) && wprot != pwp->wp_oprot) {
3428                 retry:
3429                         seg = as_segat(as, vaddr);
3430                         if (seg == NULL) {
3431                                 panic("as_setwatchprot: no seg");
3432                                 /*NOTREACHED*/
3433                         }
3434                         err = segop_setprot(seg, vaddr, PAGESIZE, wprot);
3434                         err = SEGOP_SETPROT(seg, vaddr, PAGESIZE, wprot);
3435                         if (err == IE_RETRY) {
3436                                 ASSERT(retrycnt == 0);
3437                                 retrycnt++;
3438                                 goto retry;
3439                         }
3440                 }
3441                 pwp->wp_oprot = prot;
3442                 pwp->wp_prot = wprot;

3444                 pwp = AVL_NEXT(&as->a_wpage, pwp);
3445         }
3446 }

3448 /*
3449  * Clear all of the watched pages in the range.
3450  */
3451 static void
3452 as_clearwatchprot(struct as *as, caddr_t addr, size_t size)
3453 {
3454         caddr_t eaddr = addr + size;
3455         struct watched_page *pwp;
3456         struct watched_page tpw;
3457         uint_t prot;
3458         struct seg *seg;
3459         int err, retrycnt;
3460         avl_index_t where;

3462         if (avl_numnodes(&as->a_wpage) == 0)
3463                 return;

3465         tpw.wp_vaddr = (caddr_t)((uintptr_t)addr & (uintptr_t)PAGEMASK);
3466         if ((pwp = avl_find(&as->a_wpage, &tpw, &where)) == NULL)
3467                 pwp = avl_nearest(&as->a_wpage, where, AVL_AFTER);

3469         ASSERT(AS_WRITE_HELD(as, &as->a_lock));

3471         while (pwp != NULL && pwp->wp_vaddr < eaddr) {

3473                 if ((prot = pwp->wp_oprot) != 0) {
3474                         retrycnt = 0;

3476                         if (prot != pwp->wp_prot) {
3477                         retry:
3478                                 seg = as_segat(as, pwp->wp_vaddr);
3479                                 if (seg == NULL)
3480                                         continue;
3481                                 err = segop_setprot(seg, pwp->wp_vaddr,
3481                                 err = SEGOP_SETPROT(seg, pwp->wp_vaddr,
3482                                     PAGESIZE, prot);
3483                                 if (err == IE_RETRY) {
3484                                         ASSERT(retrycnt == 0);
3485                                         retrycnt++;
3486                                         goto retry;

3488                                 }
3489                         }
3490                         pwp->wp_oprot = 0;
```



```
3491                         pwp->wp_prot = 0;
3492                 }

3494                 pwp = AVL_NEXT(&as->a_wpage, pwp);
3495         }
3496 }
______unchanged_portion_omitted_

3515 /*
3516  * return memory object ID
3517  */
3518 int
3519 as_getmemid(struct as *as, caddr_t addr, memid_t *memidp)
3520 {
3521         struct seg      *seg;
3522         int             sts;

3524         AS_LOCK_ENTER(as, &as->a_lock, RW_READER);
3525         seg = as_segat(as, addr);
3526         if (seg == NULL) {
3527                 AS_LOCK_EXIT(as, &as->a_lock);
3528                 return (EFAULT);
3529         }
3530         /*
3531          * catch old drivers which may not support getmemid
3532          */
3533         if (seg->s_ops->getmemid == NULL) {
3534                 AS_LOCK_EXIT(as, &as->a_lock);
3535                 return (ENODEV);
3536         }

3538         sts = segop_getmemid(seg, addr, memidp);
3538         sts = SEGOP_GETMEMID(seg, addr, memidp);

3540         AS_LOCK_EXIT(as, &as->a_lock);
3541         return (sts);
3542 }
______unchanged_portion_omitted_
```

```
**********************************************************
   31976 Fri May  8 18:04:10 2015
new/usr/src/uts/common/vm/vm_pvn.c
patch lower-case-segops
**********************************************************
```

```
/*
 * CDDL HEADER START
 *
 * The contents of this file are subject to the terms of the
 * Common Development and Distribution License (the "License").
 * You may not use this file except in compliance with the License.
 *
 * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
 * or http://www.opensolaris.org/os/licensing.
 * See the License for the specific language governing permissions
 * and limitations under the License.
 *
 * When distributing Covered Code, include this CDDL HEADER in each
 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
 * If applicable, add the following below this CDDL HEADER, with the
 * fields enclosed by brackets "[]" replaced with your own identifying
 * information: Portions Copyright [yyyy] [name of copyright owner]
 *
 * CDDL HEADER END
 */
/*
 * Copyright (c) 1986, 2010, Oracle and/or its affiliates. All rights reserved.
 * Copyright 2015 Nexenta Systems, Inc.  All rights reserved.
 */

/*      Copyright (c) 1984, 1986, 1987, 1988, 1989 AT&T */
/*        All Rights Reserved   */

/*
 * University Copyright- Copyright (c) 1982, 1986, 1988
 * The Regents of the University of California
 * All Rights Reserved
 *
 * University Acknowledgment- Portions of this document are derived from
 * software developed by the University of California, Berkeley, and its
 * contributors.
 */

/*
 * VM - paged vnode.
 *
 * This file supplies vm support for the vnode operations that deal with pages.
 */
#include <sys/types.h>
#include <sys/t_lock.h>
#include <sys/param.h>
#include <sys/sysmacros.h>
#include <sys/systm.h>
#include <sys/time.h>
#include <sys/buf.h>
#include <sys/vnode.h>
#include <sys/uio.h>
#include <sys/vmsystm.h>
#include <sys/mman.h>
#include <sys/vfs.h>
#include <sys/cred.h>
#include <sys/user.h>
#include <sys/kmem.h>
#include <sys/cmn_err.h>
#include <sys/debug.h>
#include <sys/cpuvar.h>
```



```
#include <sys/vtrace.h>
#include <sys/tnf_probe.h>

#include <vm/hat.h>
#include <vm/as.h>
#include <vm/seg.h>
#include <vm/rm.h>
#include <vm/pvn.h>
#include <vm/page.h>
#include <vm/seg_map.h>
#include <vm/seg_kmem.h>
#include <sys/fs/swapnode.h>

int pvn_nofodklust = 0;
int pvn_write_noklust = 0;

uint_t pvn_vmodsort_supported = 0;      /* set if HAT supports VMODSORT */
uint_t pvn_vmodsort_disable = 0;        /* set in /etc/system to disable HAT */
                                        /* support for vmodsort for testing */

static struct kmem_cache *marker_cache = NULL;

/*
 * Find the largest contiguous block which contains ‘addr’ for file offset
 * ‘offset’ in it while living within the file system block sizes (‘vp_off’
 * and ‘vp_len’) and the address space limits for which no pages currently
 * exist and which map to consecutive file offsets.
 */
page_t *
pvn_read_kluster(
        struct vnode *vp,
        u_offset_t off,
        struct seg *seg,
        caddr_t addr,
        u_offset_t *offp,                       /* return values */
        size_t *lenp,                           /* return values */
        u_offset_t vp_off,
        size_t vp_len,
        int isra)
{
        ssize_t deltaf, deltab;
        page_t *pp;
        page_t *plist = NULL;
        spgcnt_t pagesavail;
        u_offset_t vp_end;

        ASSERT(off >= vp_off && off < vp_off + vp_len);

        /*
         * We only want to do klustering/read ahead if there
         * is more than minfree pages currently available.
         */
        pagesavail = freemem - minfree;

        if (pagesavail <= 0)
                if (isra)
                        return ((page_t *)NULL);    /* ra case - give up */
                else
                        pagesavail = 1;             /* must return a page */

        /* We calculate in pages instead of bytes due to 32-bit overflows */
        if (pagesavail < (spgcnt_t)btopr(vp_len)) {
                /*
                 * Don’t have enough free memory for the
                 * max request, try sizing down vp request.
                 */
```

```
128                 deltab = (ssize_t)(off - vp_off);
129                 vp_len -= deltab;
130                 vp_off += deltab;
131                 if (pagesavail < btopr(vp_len)) {
132                         /*
133                          * Still not enough memory, just settle for
134                          * pagesavail which is at least 1.
135                          */
136                         vp_len = ptob(pagesavail);
137                 }
138         }

140         vp_end = vp_off + vp_len;
141         ASSERT(off >= vp_off && off < vp_end);

143         if (isra && segop_kluster(seg, addr, 0))
143         if (isra && SEGOP_KLUSTER(seg, addr, 0))
144                 return ((page_t *)NULL);        /* segment driver says no */

146         if ((plist = page_create_va(vp, off,
147             PAGESIZE, PG_EXCL | PG_WAIT, seg, addr)) == NULL)
148                 return ((page_t *)NULL);

150         if (vp_len <= PAGESIZE || pvn_nofodklust) {
151                 *offp = off;
152                 *lenp = MIN(vp_len, PAGESIZE);
153         } else {
154                 /*
155                  * Scan back from front by incrementing "deltab" and
156                  * comparing "off" with "vp_off + deltab" to avoid
157                  * "signed" versus "unsigned" conversion problems.
158                  */
159                 for (deltab = PAGESIZE; off >= vp_off + deltab;
160                     deltab += PAGESIZE) {
161                         /*
162                          * Call back to the segment driver to verify that
163                          * the klustering/read ahead operation makes sense.
164                          */
165                         if (segop_kluster(seg, addr, -deltab))
165                         if (SEGOP_KLUSTER(seg, addr, -deltab))
166                                 break;          /* page not eligible */
167                         if ((pp = page_create_va(vp, off - deltab,
168                             PAGESIZE, PG_EXCL, seg, addr - deltab))
169                             == NULL)
170                                 break;          /* already have the page */
171                         /*
172                          * Add page to front of page list.
173                          */
174                         page_add(&plist, pp);
175                 }
176                 deltab -= PAGESIZE;

178                 /* scan forward from front */
179                 for (deltaf = PAGESIZE; off + deltaf < vp_end;
180                     deltaf += PAGESIZE) {
181                         /*
182                          * Call back to the segment driver to verify that
183                          * the klustering/read ahead operation makes sense.
184                          */
185                         if (segop_kluster(seg, addr, deltaf))
185                         if (SEGOP_KLUSTER(seg, addr, deltaf))
186                                 break;          /* page not file extension */
187                         if ((pp = page_create_va(vp, off + deltaf,
188                             PAGESIZE, PG_EXCL, seg, addr + deltaf))
189                             == NULL)
190                                 break;          /* already have page */
```



```
192                         /*
193                          * Add page to end of page list.
194                          */
195                         page_add(&plist, pp);
196                         plist = plist->p_next;
197                 }
198                 *offp = off = off - deltab;
199                 *lenp = deltab + deltaf;
200                 ASSERT(off >= vp_off);

202                 /*
203                  * If we ended up getting more than was actually
204                  * requested, retract the returned length to only
205                  * reflect what was requested.  This might happen
206                  * if we were allowed to kluster pages across a
207                  * span of (say) 5 frags, and frag size is less
208                  * than PAGESIZE.  We need a whole number of
209                  * pages to contain those frags, but the returned
210                  * size should only allow the returned range to
211                  * extend as far as the end of the frags.
212                  */
213                 if ((vp_off + vp_len) < (off + *lenp)) {
214                         ASSERT(vp_end > off);
215                         *lenp = vp_end - off;
216                 }
217         }
218         TRACE_3(TR_FAC_VM, TR_PVN_READ_KLUSTER,
219             "pvn_read_kluster:seg %p addr %x isra %x",
220             seg, addr, isra);
221         return (plist);
222 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   55290 Fri May  8 18:04:11 2015
new/usr/src/uts/common/vm/vm_seg.c
patch lower-case-segops
**********************************************************
______unchanged_portion_omitted_

1636 /*
1637  * Unmap a segment and free it from its associated address space.
1638  * This should be called by anybody who's finished with a whole segment's
1639  * mapping.  Just calls segop_unmap() on the whole mapping .  It is the
1639  * mapping.  Just calls SEGOP_UNMAP() on the whole mapping .  It is the
1640  * responsibility of the segment driver to unlink the the segment
1641  * from the address space, and to free public and private data structures
1642  * associated with the segment.  (This is typically done by a call to
1643  * seg_free()).
1644  */
1645 void
1646 seg_unmap(struct seg *seg)
1647 {
1648 #ifdef DEBUG
1649         int ret;
1650 #endif /* DEBUG */

1652         ASSERT(seg->s_as && AS_WRITE_HELD(seg->s_as, &seg->s_as->a_lock));

1654         /* Shouldn't have called seg_unmap if mapping isn't yet established */
1655         ASSERT(seg->s_data != NULL);

1657         /* Unmap the whole mapping */
1658 #ifdef DEBUG
1659         ret = segop_unmap(seg, seg->s_base, seg->s_size);
1659         ret = SEGOP_UNMAP(seg, seg->s_base, seg->s_size);
1660         ASSERT(ret == 0);
1661 #else
1662         segop_unmap(seg, seg->s_base, seg->s_size);
1662         SEGOP_UNMAP(seg, seg->s_base, seg->s_size);
1663 #endif /* DEBUG */
1664 }

1666 /*
1667  * Free the segment from its associated as. This should only be called
1668  * if a mapping to the segment has not yet been established (e.g., if
1669  * an error occurs in the middle of doing an as_map when the segment
1670  * has already been partially set up) or if it has already been deleted
1671  * (e.g., from a segment driver unmap routine if the unmap applies to the
1672  * entire segment). If the mapping is currently set up then seg_unmap() should
1673  * be called instead.
1674  */
1675 void
1676 seg_free(struct seg *seg)
1677 {
1678         register struct as *as = seg->s_as;
1679         struct seg *tseg = as_removeseg(as, seg);

1681         ASSERT(tseg == seg);

1683         /*
1684          * If the segment private data field is NULL,
1685          * then segment driver is not attached yet.
1686          */
1687         if (seg->s_data != NULL)
1688                 segop_free(seg);
1688                 SEGOP_FREE(seg);

1690         mutex_destroy(&seg->s_pmtx);
```



```
1691         ASSERT(seg->s_phead.p_lnext == &seg->s_phead);
1692         ASSERT(seg->s_phead.p_lprev == &seg->s_phead);
1693         kmem_cache_free(seg_cache, seg);
1694 }
______unchanged_portion_omitted_

1856 /*
1857  * General not supported function for segop_inherit
1857  * General not supported function for SEGOP_INHERIT
1858  */
1859 /* ARGSUSED */
1860 int
1861 seg_inherit_notsup(struct seg *seg, caddr_t addr, size_t len, uint_t op)
1862 {
1863         return (ENOTSUP);
1864 }
______unchanged_portion_omitted_
```