

```
*****  
29374 Tue Nov 24 09:34:44 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  
666 * Todo: A "SEGOP_CPR" like SEGOP_DUMP should be introduced, the cpr  
667 * module shouldn't need to know segment details like if it is  
668 * sparsely filled or not (makes kseg_table obsolete).  
669 */  
670 pgcnt_t  
671 cpr_count_seg_pages(int mapflag, bitfunc_t bitfunc)  
672 {  
673     struct seg *segp;  
674     pgcnt_t pages;  
675     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     }  
685     return (pages);  
686 }  
unchanged_portion_omitted
```

new/usr/src/uts/common/exec/elf/elf.c

```
*****
55632 Tue Nov 24 09:34:44 2015
new/usr/src/uts/common/exec/elf/elf.c
patch lower-case-segops
*****
_____ unchanged_portion_omitted _____
1424 #ifdef _ELF32_COMPAT
1425 extern size_t elf_datasz_max;
1426 #else
1427 size_t elf_datasz_max = 1 * 1024 * 1024;
1428 #endif
1430 /*
1431 * This function processes mappings that correspond to load objects to
1432 * examine their respective sections for elfcore(). It's called once with
1433 * v set to NULL to count the number of sections that we're going to need
1434 * and then again with v set to some allocated buffer that we fill in with
1435 * all the section data.
1436 */
1437 static int
1438 process_scns(core_content_t content, proc_t *p, cred_t *credp, vnode_t *vp,
1439     Shdr *v, int nv, rlim64_t rlimit, Off *doffsetp, int *nshdrsp)
1440 {
1441     vnode_t *lastvp = NULL;
1442     struct seg *seg;
1443     int i, j;
1444     void *data = NULL;
1445     size_t datasz = 0;
1446     shstrtab_t shstrtab;
1447     struct as *as = p->p_as;
1448     int error = 0;
1449
1450     if (v != NULL)
1451         shstrtab_init(&shstrtab);
1452
1453     i = 1;
1454     for (seg = AS_SEGFIRST(as); seg != NULL; seg = AS_SEGNEXT(as, seg)) {
1455         uint_t prot;
1456         vnode_t *mvp;
1457         void *tmp;
1458         caddr_t saddr = seg->s_base;
1459         caddr_t naddr;
1460         caddr_t eaddr;
1461         size_t segsize;
1462
1463         Ehdr ehdr;
1464         int nshdrs, shstrndx, nphdrs;
1465         caddr_t shbase;
1466         ssize_t shsize;
1467         char *shstrbase;
1468         ssize_t shstrsize;
1469
1470         Shdr *shdr;
1471         const char *name;
1472         size_t sz;
1473         uintptr_t off;
1474
1475         int ctf_ndx = 0;
1476         int symtab_ndx = 0;
1477
1478         /*
1479          * Since we're just looking for text segments of load
1480          * objects, we only care about the protection bits; we don't
1481          * care about the actual size of the segment so we use the
1482          * reserved size. If the segment's size is zero, there's
1483
1484
1485
1486
1487
1488
1489
1490
1491
1492
1493
1494
1495
1496
1497
1498
1499
1500
1501
1502
1503
1504
1505
1506
1507
1508
1509
1510
1511
1512
1513
1514
1515
1516
1517
1518
1519
1520
1521
1522
1523
1524
1525
1526
1527
1528
1529
1530
1531
1532
1533
1534
1535
1536
1537
1538
1539
1540
1541
1542
1543
1544
1545
1546
1547
1548
1549
1550
1551
1552
1553
1554
1555
1556
1557
1558
1559
1560
1561
1562
1563
1564
1565
1566
1567
1568
1569
1570
1571
1572
1573
1574
1575
1576
1577
1578
1579
1580
1581
1582
1583
1584
1585
1586
1587
1588
1589
1590
1591
1592
1593
1594
1595
1596
1597
1598
1599
1600
1601
1602
1603
1604
1605
1606
1607
1608
1609
1610
1611
1612
1613
1614
1615
1616
1617
1618
1619
1620
1621
1622
1623
1624
1625
1626
1627
1628
1629
1630
1631
1632
1633
1634
1635
1636
1637
1638
1639
1640
1641
1642
1643
1644
1645
1646
1647
1648
1649
1650
1651
1652
1653
1654
1655
1656
1657
1658
1659
1660
1661
1662
1663
1664
1665
1666
1667
1668
1669
1670
1671
1672
1673
1674
1675
1676
1677
1678
1679
1680
1681
1682
1683
1684
1685
1686
1687
1688
1689
1690
1691
1692
1693
1694
1695
1696
1697
1698
1699
1700
1701
1702
1703
1704
1705
1706
1707
1708
1709
1710
1711
1712
1713
1714
1715
1716
1717
1718
1719
1720
1721
1722
1723
1724
1725
1726
1727
1728
1729
1730
1731
1732
1733
1734
1735
1736
1737
1738
1739
1740
1741
1742
1743
1744
1745
1746
1747
1748
1749
1750
1751
1752
1753
1754
1755
1756
1757
1758
1759
1760
1761
1762
1763
1764
1765
1766
1767
1768
1769
1770
1771
1772
1773
1774
1775
1776
1777
1778
1779
1780
1781
1782
1783
1784
1785
1786
1787
1788
1789
1790
1791
1792
1793
1794
1795
1796
1797
1798
1799
1800
1801
1802
1803
1804
1805
1806
1807
1808
1809
1810
1811
1812
1813
1814
1815
1816
1817
1818
1819
1820
1821
1822
1823
1824
1825
1826
1827
1828
1829
1830
1831
1832
1833
1834
1835
1836
1837
1838
1839
1840
1841
1842
1843
1844
1845
1846
1847
1848
1849
1850
1851
1852
1853
1854
1855
1856
1857
1858
1859
1860
1861
1862
1863
1864
1865
1866
1867
1868
1869
1870
1871
1872
1873
1874
1875
1876
1877
1878
1879
1880
1881
1882
1883
1884
1885
1886
1887
1888
1889
1890
1891
1892
1893
1894
1895
1896
1897
1898
1899
1900
1901
1902
1903
1904
1905
1906
1907
1908
1909
1910
1911
1912
1913
1914
1915
1916
1917
1918
1919
1920
1921
1922
1923
1924
1925
1926
1927
1928
1929
1930
1931
1932
1933
1934
1935
1936
1937
1938
1939
1940
1941
1942
1943
1944
1945
1946
1947
1948
1949
1950
1951
1952
1953
1954
1955
1956
1957
1958
1959
1960
1961
1962
1963
1964
1965
1966
1967
1968
1969
1970
1971
1972
1973
1974
1975
1976
1977
1978
1979
1980
1981
1982
1983
1984
1985
1986
1987
1988
1989
1990
1991
1992
1993
1994
1995
1996
1997
1998
1999
2000
2001
2002
2003
2004
2005
2006
2007
2008
2009
2010
2011
2012
2013
2014
2015
2016
2017
2018
2019
2020
2021
2022
2023
2024
2025
2026
2027
2028
2029
2030
2031
2032
2033
2034
2035
2036
2037
2038
2039
2040
2041
2042
2043
2044
2045
2046
2047
2048
2049
2050
2051
2052
2053
2054
2055
2056
2057
2058
2059
2060
2061
2062
2063
2064
2065
2066
2067
2068
2069
2070
2071
2072
2073
2074
2075
2076
2077
2078
2079
2080
2081
2082
2083
2084
2085
2086
2087
2088
2089
2090
2091
2092
2093
2094
2095
2096
2097
2098
2099
2099
2100
2101
2102
2103
2104
2105
2106
2107
2108
2109
2110
2111
2112
2113
2114
2115
2116
2117
2118
2119
2120
2121
2122
2123
2124
2125
2126
2127
2128
2129
2130
2131
2132
2133
2134
2135
2136
2137
2138
2139
2140
2141
2142
2143
2144
2145
2146
2147
2148
2149
2150
2151
2152
2153
2154
2155
2156
2157
2158
2159
2159
2160
2161
2162
2163
2164
2165
2166
2167
2168
2169
2169
2170
2171
2172
2173
2174
2175
2176
2177
2178
2179
2179
2180
2181
2182
2183
2184
2185
2186
2187
2187
2188
2189
2189
2190
2191
2192
2193
2194
2195
2196
2197
2198
2199
2199
2200
2201
2202
2203
2204
2205
2206
2207
2208
2209
2209
2210
2211
2212
2213
2214
2215
2216
2217
2218
2219
2219
2220
2221
2222
2223
2224
2225
2226
2227
2228
2229
2229
2230
2231
2232
2233
2234
2235
2236
2237
2238
2239
2239
2240
2241
2242
2243
2244
2245
2246
2247
2248
2249
2249
2250
2251
2252
2253
2254
2255
2256
2257
2258
2259
2259
2260
2261
2262
2263
2264
2265
2266
2267
2268
2269
2269
2270
2271
2272
2273
2274
2275
2276
2277
2278
2279
2279
2280
2281
2282
2283
2284
2285
2286
2287
2287
2288
2289
2289
2290
2291
2292
2293
2294
2295
2296
2297
2297
2298
2299
2299
2300
2301
2302
2303
2304
2305
2306
2307
2308
2309
2309
2310
2311
2312
2313
2314
2315
2316
2317
2318
2319
2319
2320
2321
2322
2323
2324
2325
2326
2327
2328
2329
2329
2330
2331
2332
2333
2334
2335
2336
2337
2338
2339
2339
2340
2341
2342
2343
2344
2345
2346
2347
2348
2349
2349
2350
2351
2352
2353
2354
2355
2356
2357
2358
2359
2359
2360
2361
2362
2363
2364
2365
2366
2367
2368
2369
2369
2370
2371
2372
2373
2374
2375
2376
2377
2378
2379
2379
2380
2381
2382
2383
2384
2385
2386
2387
2387
2388
2389
2389
2390
2391
2392
2393
2394
2395
2396
2397
2397
2398
2399
2399
2400
2401
2402
2403
2404
2405
2406
2407
2408
2409
2409
2410
2411
2412
2413
2414
2415
2416
2417
2418
2419
2419
2420
2421
2422
2423
2424
2425
2426
2427
2428
2429
2429
2430
2431
2432
2433
2434
2435
2436
2437
2438
2439
2439
2440
2441
2442
2443
2444
2445
2446
2447
2447
2448
2449
2449
2450
2451
2452
2453
2454
2455
2456
2457
2458
2459
2459
2460
2461
2462
2463
2464
2465
2466
2467
2468
2469
2469
2470
2471
2472
2473
2474
2475
2476
2477
2478
2479
2479
2480
2481
2482
2483
2484
2485
2486
2487
2487
2488
2489
2489
2490
2491
2492
2493
2494
2495
2496
2497
2497
2498
2499
2499
2500
2501
2502
2503
2504
2505
2506
2507
2508
2509
2509
2510
2511
2512
2513
2514
2515
2516
2517
2518
2519
2519
2520
2521
2522
2523
2524
2525
2526
2527
2528
2529
2529
2530
2531
2532
2533
2534
2535
2536
2537
2538
2539
2539
2540
2541
2542
2543
2544
2545
2546
2547
2548
2548
2549
2549
2550
2551
2552
2553
2554
2555
2556
2557
2558
2559
2559
2560
2561
2562
2563
2564
2565
2566
2567
2568
2569
2569
2570
2571
2572
2573
2574
2575
2576
2577
2578
2579
2579
2580
2581
2582
2583
2584
2585
2586
2587
2587
2588
2589
2589
2590
2591
2592
2593
2594
2595
2596
2597
2597
2598
2599
2599
2600
2601
2602
2603
2604
2605
2606
2607
2608
2609
2609
2610
2611
2612
2613
2614
2615
2616
2617
2618
2619
2619
2620
2621
2622
2623
2624
2625
2626
2627
2628
2629
2629
2630
2631
2632
2633
2634
2635
2636
2637
2638
2638
2639
2639
2640
2641
2642
2643
2644
2645
2646
2647
2648
2648
2649
2649
2650
2651
2652
2653
2654
2655
2656
2657
2658
2659
2659
2660
2661
2662
2663
2664
2665
2666
2667
2668
2669
2669
2670
2671
2672
2673
2674
2675
2676
2677
2678
2679
2679
2680
2681
2682
2683
2684
2685
2686
2687
2687
2688
2689
2689
2690
2691
2692
2693
2694
2695
2696
2697
2697
2698
2699
2699
2700
2701
2702
2703
2704
2705
2706
2707
2708
2709
2709
2710
2711
2712
2713
2714
2715
2716
2717
2718
2719
2719
2720
2721
2722
2723
2724
2725
2726
2727
2728
2729
2729
2730
2731
2732
2733
2734
2735
2736
2737
2738
2738
2739
2739
2740
2741
2742
2743
2744
2745
2746
2747
2748
2748
2749
2749
2750
2751
2752
2753
2754
2755
2756
2757
2758
2759
2759
2760
2761
2762
2763
2764
2765
2766
2767
2768
2769
2769
2770
2771
2772
2773
2774
2775
2776
2777
2778
2778
2779
2779
2780
2781
2782
2783
2784
2785
2786
2787
2787
2788
2789
2789
2790
2791
2792
2793
2794
2795
2796
2797
2797
2798
2799
2799
2800
2801
2802
2803
2804
2805
2806
2807
2808
2809
2809
2810
2811
2812
2813
2814
2815
2816
2817
2818
2819
2819
2820
2821
2822
2823
2824
2825
2826
2827
2828
2828
2829
2829
2830
2831
2832
2833
2834
2835
2836
2837
2838
2838
2839
2839
2840
2841
2842
2843
2844
2845
2846
2847
2848
2848
2849
2849
2850
2851
2852
2853
2854
2855
2856
2857
2858
2859
2859
2860
2861
2862
2863
2864
2865
2866
2867
2868
2869
2869
2870
2871
2872
2873
2874
2875
2876
2877
2878
2879
2879
2880
2881
2882
2883
2884
2885
2886
2887
2887
2888
2889
2889
2890
2891
2892
2893
2894
2895
2896
2897
2897
2898
2899
2899
2900
2901
2902
2903
2904
2905
2906
2907
2908
2909
2909
2910
2911
2912
2913
2914
2915
2916
2917
2918
2919
2919
2920
2921
2922
2923
2924
2925
2926
2927
2928
2928
2929
2929
2930
2931
2932
2933
2934
2935
2936
2937
2938
2938
2939
2939
2940
2941
2942
2943
2944
2945
2946
2947
2948
2948
2949
2949
2950
2951
2952
2953
2954
2955
2956
2957
2958
2959
2959
2960
2961
2962
2963
2964
2965
2966
2967
2968
2969
2969
2970
2971
2972
2973
2974
2975
2976
2977
2978
2979
2979
2980
2981
2982
2983
2984
2985
2986
2987
2987
2988
2989
2989
2990
2991
2992
2993
2994
2995
2996
2997
2997
2998
2999
2999
3000
3001
3002
3003
3004
3005
3006
3007
3008
3009
3009
3010
3011
3012
3013
3014
3015
3016
3017
3018
3019
3019
3020
3021
3022
3023
3024
3025
3026
3027
3028
3028
3029
3029
3030
3031
3032
3033
3034
3035
3036
3037
3038
3038
3039
3039
3040
3041
3042
3043
3044
3045
3046
3047
3048
3048
3049
3049
3050
3051
3052
3053
3054
3055
3056
3057
3058
3059
3059
3060
3061
3062
3063
3064
3065
3066
3067
3068
3069
3069
3070
3071
3072
3073
3074
3075
3076
3077
3078
3079
3079
3080
3081
3082
3083
3084
3085
3086
3087
3087
3088
3089
3089
3090
3091
3092
3093
3094
3095
3096
3097
3097
3098
3099
3099
3100
3101
3102
3103
3104
3105
3106
3107
3108
3109
3109
3110
3111
3112
3113
3114
3115
3116
3117
3118
3119
3119
3120
3121
3122
3123
3124
3125
3126
3127
3128
3128
3129
3129
3130
3131
3132
3133
3134
3135
3136
3137
3138
3138
3139
3139
3140
3141
3142
3143
3144
3145
3146
3147
3148
3148
3149
3149
3150
3151
3152
3153
3154
3155
3156
3157
3158
3159
3159
3160
3161
3162
3163
3164
3165
3166
3167
3168
3169
3169
3170
3171
3172
3173
3174
3175
3176
3177
3178
3179
3179
3180
3181
3182
3183
3184
3185
3186
3187
3187
3188
3189
3189
3190
3191
3192
3193
3194
3195
3196
3197
3197
3198
3199
3199
3200
3201
3202
3203
3204
3205
3206
3207
3208
3209
3209
3210
3211
3212
3213
3214
3215
3216
3217
3218
3219
3219
3220
3221
3222
3223
3224
3225
3226
3227
3228
3228
3229
3229
3230
3231
3232
3233
3234
3235
3236
3237
3238
3238
3239
3239
3240
3241
3242
3243
3244
3245
3246
3247
3248
3248
3249
3249
3250
3251
3252
3253
3254
3255
3256
3257
3258
3259
3259
3260
3261
3262
3263
3264
3265
3266
3267
3268
3269
3269
3270
3271
3272
3273
3274
3275
3276
3277
3278
3279
3279
3280
3281
3282
3283
3284
3285
3286
3287
3287
3288
3289
3289
3290
3291
3292
3293
3294
3295
3296
3297
3297
3298
3299
3299
3300
3301
3302
3303
3304
3305
3306
3307
3308
3309
3309
3310
3311
3312
3313
3314
3315
3316
3317
3318
3319
3319
3320
3321
3322
3323
3324
3325
3326
3327
3328
3328
3329
3329
3330
3331
3332
3333
3334
3335
3336
3337
3338
3338
3339
3339
3340
3341
3342
3343
3344
3345
3346
3347
3348
3348
3349
3349
3350
3351
3352
3353
3354
3355
3356
3357
3358
3359
3359
3360
3361
3362
3363
3364
3365
3366
3367
3368
3369
3369
3370
3371
3372
3373
3374
3375
3376
3377
3378
3379
3379
3380
3381
3382
3383
3384
3385
3386
3387
3387
3388
3389
3389
3390
3391
3392
3393
3394
3395
3396
3397
3397
3398
3399
3399
3400
3401
3402
3403
3404
3405
3406
3407
3408
3409
3409
3410
3411
3412
3
```

```

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     }
1560     copy_scn(shdr, mvp, &v[i], vp,
1561             doffsetp, data, datasz, credp,
1562             rlimit);
1563 }
1564 ctf_ndx = i++;
1565 /*
1566 * We've already dumped the symtab.
1567 */
1568 if (symtab != NULL &&
1569     symtab->sh_type == SHT_SYMTAB &&
1570     symtab_ndx != 0)
1571     continue;
1572
1573 } else if (strcmp(name,
1574                   shstrtab_data[STR_SYMTAB]) == 0) {
1575     if ((content & CC_CONTENT_SYMTAB) == 0 ||
1576         symtab != 0)
1577         continue;
1578     symtab = shdr;
1579 }
1580 if (symtab != NULL) {
1581     if ((symtab->sh_type != SHT_DYNSYM &&
1582         symtab->sh_type != SHT_SYMTAB) ||
1583         symtab->sh_link == 0 ||
1584         symtab->sh_link >= nshdrs)
1585         continue;
1586     strtab = (Shdr *) (shbase +
1587                         symtab->sh_link * ehdr.e_shentsize);
1588
1589     if (strtab->sh_type != SHT_STRTAB)
1590         continue;
1591
1592     if (v != NULL && i < nv - 2) {
1593         sz = MAX(symtab->sh_size,
1594                  strtab->sh_size);
1595         if (sz > datasz &&
1596             sz <= elf_datasz_max) {
1597             if (data != NULL)
1598                 kmem_free(data, datasz);
1599
1600             datasz = sz;
1601             data = kmem_alloc(datasz,
1602                               KM_SLEEP);
1603         }
1604
1605         if (symtab->sh_type == SHT_DYNSYM) {
1606             v[i].sh_name = shstrtab_ndx(
1607                             &shstrtab, STR_DYNSYM);
1608
1609
1610
1611
1612
1613
1614
1615
1616
1617
1618
1619
1620
1621
1622
1623
1624
1625
1626
1627
1628
1629
1630
1631
1632
1633
1634
1635
1636
1637
1638
1639
1640
1641
1642
1643
1644
1645
1646
1647
1648
1649
1650
1651
1652
1653
1654
1655
1656
1657
1658
1659
1660
1661
1662
1663
1664
1665
1666
1667
1668
1669
1670
1671
1672
1673
1674
1675
1676
1677
1678
1679
1680
1681
1682
1683
1684
1685
1686
1687
1688
1689
1690
1691
1692
1693
1694
1695
1696
1697
1698
1699
1700
1701
1702
1703
1704
1705
1706
1707
1708
1709
1710
1711
1712
1713
1714
1715
1716
1717
1718
1719
1720
1721
1722
1723
1724
1725
1726
1727
1728
1729
1730
1731
1732
1733
1734
1735
1736
1737
1738
1739
1740
1741
1742
1743
1744
1745
1746
1747
1748
1749
1750
1751
1752
1753
1754
1755
1756
1757
1758
1759
1760
1761
1762
1763
1764
1765
1766
1767
1768
1769
1770
1771
1772
1773
1774
1775
1776
1777
1778
1779
1780
1781
1782
1783
1784
1785
1786
1787
1788
1789
1790
1791
1792
1793
1794
1795
1796
1797
1798
1799
1800
1801
1802
1803
1804
1805
1806
1807
1808
1809
1810
1811
1812
1813
1814
1815
1816
1817
1818
1819
1820
1821
1822
1823
1824
1825
1826
1827
1828
1829
1830
1831
1832
1833
1834
1835
1836
1837
1838
1839
1840
1841
1842
1843
1844
1845
1846
1847
1848
1849
1850
1851
1852
1853
1854
1855
1856
1857
1858
1859
1860
1861
1862
1863
1864
1865
1866
1867
1868
1869
1870
1871
1872
1873
1874
1875
1876
1877
1878
1879
1880
1881
1882
1883
1884
1885
1886
1887
1888
1889
1890
1891
1892
1893
1894
1895
1896
1897
1898
1899
1900
1901
1902
1903
1904
1905
1906
1907
1908
1909
1910
1911
1912
1913
1914
1915
1916
1917
1918
1919
1920
1921
1922
1923
1924
1925
1926
1927
1928
1929
1930
1931
1932
1933
1934
1935
1936
1937
1938
1939
1940
1941
1942
1943
1944
1945
1946
1947
1948
1949
1950
1951
1952
1953
1954
1955
1956
1957
1958
1959
1960
1961
1962
1963
1964
1965
1966
1967
1968
1969
1970
1971
1972
1973
1974
1975
1976
1977
1978
1979
1980
1981
1982
1983
1984
1985
1986
1987
1988
1989
1990
1991
1992
1993
1994
1995
1996
1997
1998
1999
2000
2001
2002
2003
2004
2005
2006
2007
2008
2009
2010
2011
2012
2013
2014
2015
2016
2017
2018
2019
2020
2021
2022
2023
2024
2025
2026
2027
2028
2029
2030
2031
2032
2033
2034
2035
2036
2037
2038
2039
2040
2041
2042
2043
2044
2045
2046
2047
2048
2049
2050
2051
2052
2053
2054
2055
2056
2057
2058
2059
2060
2061
2062
2063
2064
2065
2066
2067
2068
2069
2070
2071
2072
2073
2074
2075
2076
2077
2078
2079
2080
2081
2082
2083
2084
2085
2086
2087
2088
2089
2090
2091
2092
2093
2094
2095
2096
2097
2098
2099
20100
20101
20102
20103
20104
20105
20106
20107
20108
20109
20110
20111
20112
20113
20114
20115
20116
20117
20118
20119
20120
20121
20122
20123
20124
20125
20126
20127
20128
20129
20130
20131
20132
20133
20134
20135
20136
20137
20138
20139
20140
20141
20142
20143
20144
20145
20146
20147
20148
20149
20150
20151
20152
20153
20154
20155
20156
20157
20158
20159
20160
20161
20162
20163
20164
20165
20166
20167
20168
20169
20170
20171
20172
20173
20174
20175
20176
20177
20178
20179
20180
20181
20182
20183
20184
20185
20186
20187
20188
20189
20190
20191
20192
20193
20194
20195
20196
20197
20198
20199
20200
20201
20202
20203
20204
20205
20206
20207
20208
20209
202010
202011
202012
202013
202014
202015
202016
202017
202018
202019
202020
202021
202022
202023
202024
202025
202026
202027
202028
202029
202030
202031
202032
202033
202034
202035
202036
202037
202038
202039
202040
202041
202042
202043
202044
202045
202046
202047
202048
202049
202050
202051
202052
202053
202054
202055
202056
202057
202058
202059
202060
202061
202062
202063
202064
202065
202066
202067
202068
202069
202070
202071
202072
202073
202074
202075
202076
202077
202078
202079
202080
202081
202082
202083
202084
202085
202086
202087
202088
202089
202090
202091
202092
202093
202094
202095
202096
202097
202098
202099
2020100
2020101
2020102
2020103
2020104
2020105
2020106
2020107
2020108
2020109
2020110
2020111
2020112
2020113
2020114
2020115
2020116
2020117
2020118
2020119
2020120
2020121
2020122
2020123
2020124
2020125
2020126
2020127
2020128
2020129
2020130
2020131
2020132
2020133
2020134
2020135
2020136
2020137
2020138
2020139
2020140
2020141
2020142
2020143
2020144
2020145
2020146
2020147
2020148
2020149
2020150
2020151
2020152
2020153
2020154
2020155
2020156
2020157
2020158
2020159
2020160
2020161
2020162
2020163
2020164
2020165
2020166
2020167
2020168
2020169
2020170
2020171
2020172
2020173
2020174
2020175
2020176
2020177
2020178
2020179
2020180
2020181
2020182
2020183
2020184
2020185
2020186
2020187
2020188
2020189
20201810
20201811
20201812
20201813
20201814
20201815
20201816
20201817
20201818
20201819
202018110
202018111
202018112
202018113
202018114
202018115
202018116
202018117
202018118
202018119
2020181110
2020181111
2020181112
2020181113
2020181114
2020181115
2020181116
2020181117
2020181118
2020181119
20201811110
20201811111
20201811112
20201811113
20201811114
20201811115
20201811116
20201811117
20201811118
20201811119
202018111110
202018111111
202018111112
202018111113
202018111114
202018111115
202018111116
202018111117
202018111118
202018111119
2020181111110
2020181111111
2020181111112
2020181111113
2020181111114
2020181111115
2020181111116
2020181111117
2020181111118
2020181111119
20201811111110
20201811111111
20201811111112
20201811111113
20201811111114
20201811111115
20201811111116
20201811111117
20201811111118
20201811111119
202018111111110
202018111111111
202018111111112
202018111111113
202018111111114
202018111111115
202018111111116
202018111111117
202018111111118
202018111111119
2020181111111110
2020181111111111
2020181111111112
2020181111111113
2020181111111114
2020181111111115
2020181111111116
2020181111111117
2020181111111118
2020181111111119
20201811111111110
20201811111111111
20201811111111112
20201811111111113
20201811111111114
20201811111111115
20201811111111116
20201811111111117
20201811111111118
20201811111111119
202018111111111110
202018111111111111
202018111111111112
202018111111111113
202018111111111114
202018111111111115
202018111111111116
202018111111111117
202018111111111118
202018111111111119
2020181111111111110
2020181111111111111
2020181111111111112
2020181111111111113
2020181111111111114
2020181111111111115
2020181111111111116
2020181111111111117
2020181111111111118
2020181111111111119
20201811111111111110
20201811111111111111
20201811111111111112
20201811111111111113
20201811111111111114
20201811111111111115
20201811111111111116
20201811111111111117
20201811111111111118
20201811111111111119
202018111111111111110
202018111111111111111
202018111111111111112
202018111111111111113
202018111111111111114
202018111111111111115
202018111111111111116
202018111111111111117
202018111111111111118
202018111111111111119
2020181111111111111110
2020181111111111111111
2020181111111111111112
2020181111111111111113
2020181111111111111114
2020181111111111111115
2020181111111111111116
2020181111111111111117
2020181111111111111118
2020181111111111111119
20201811111111111111110
20201811111111111111111
20201811111111111111112
20201811111111111111113
20201811111111111111114
20201811111111111111115
20201811111111111111116
20201811111111111111117
20201811111111111111118
20201811111111111111119
202018111111111111111110
202018111111111111111111
202018111111111111111112
202018111111111111111113
202018111111111111111114
202018111111111111111115
202018111111111111111116
202018111111111111111117
202018111111111111111118
202018111111111111111119
2020181111111111111111110
2020181111111111111111111
2020181111111111111111112
2020181111111111111111113
2020181111111111111111114
2020181111111111111111115
2020181111111111111111116
2020181111111111111111117
2020181111111111111111118
2020181111111111111111119
20201811111111111111111110
20201811111111111111111111
20201811111111111111111112
20201811111111111111111113
20201811111111111111111114
20201811111111111111111115
20201811111111111111111116
20201811111111111111111117
20201811111111111111111118
20201811111111111111111119
202018111111111111111111110
202018111111111111111111111
202018111111111111111111112
202018111111111111111111113
202018111111111111111111114
202018111111111111111111115
202018111111111111111111116
202018111111111111111111117
202018111111111111111111118
202018111111111111111111119
2020181111111111111111111110
2020181111111111111111111111
2020181111111111111111111112
2020181111111111111111111113
2020181111111111111111111114
2020181111111111111111111115
2020181111111111111111111116
2020181111111111111111111117
2020181111111111111111111118
2020181111111111111111111119
20201811111111111111111111110
20201811111111111111111111111
20201811111111111111111111112
20201811111111111111111111113
20201811111111111111111111114
20201811111111111111111111115
20201811111111111111111111116
20201811111111111111111111117
20201811111111111111111111118
20201811111111111111111111119
202018111111111111111111111110
202018111111111111111111111111
202018111111111111111111111112
202018111111111111111111111113
202018111111111111111111111114
202018111111111111111111111115
202018111111111111111111111116
202018111111111111111111111117
202018111111111111111111111118
202018111111111111111111111119
2020181111111111111111111111110
2020181111111111111111111111111
2020181111111111111111111111112
2020181111111111111111111111113
2020181111111111111111111111114
2020181111111111111111111111115
2020181111111111111111111111116
2020181111111111111111111111117
2020181111111111111111111111118
2020181111111111111111111111119
20201811111111111111111111111110
20201811111111111111111111111111
20201811111111111111111111111112
20201811111111111111111111111113
20201811111111111111111111111114
20201811111111111111111111111115
20201811111111111111111111111116
20201811111111111111111111111117
20201811111111111111111111111118
20201811111111111111111111111119
202018111111111111111111111111110
202018111111111111111111111111111
202018111111111111111111111111112
202018111111111111111111111111113
202018111111111111111111111111114
202018111111111111111111111111115
202018111111111111111111111111116
202018111111111111111111111111117
202018111111111111111111111111118
202018111111111111111111111111119
2020181111111111111111111111111110
2020181111111111111111111111111111
2020181111111111111111111111111112
2020181111111111111111111111111113
2020181111111111111111111111111114
2020181111111111111111111111111115
2020181111111111111111111111111116
2020181111111111111111111111111117
2020181111111111111111111111111118
2020181111111111111111111111111119
20201811111111111111111111111111110
20201811111111111111111111111111111
202018111111111111111
```

```

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     }
1688
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;
1696
1697     if (v[i].sh_size > datasz) {
1698         if (data != NULL)
1699             kmem_free(data, datasz);
1700
1701         datasz = v[i].sh_size;
1702         data = kmem_alloc(datasz,
1703                           KM_SLEEP);
1704     }
1705
1706     shstrtab_dump(&shstrtab, data);
1707
1708     if ((error = core_write(vp, UIO_SYSSPACE, *doffsetp,
1709         data, v[i].sh_size, rlimit, credp)) != 0)
1710         goto done;
1711
1712     *doffsetp += v[i].sh_size;
1713
1714 done:
1715     if (data != NULL)
1716         kmem_free(data, datasz);
1717
1718     return (error);
1719 }
1720
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);
1754
1755     AS_LOCK_ENTER(as, &as->a_lock, RW_WRITER);
1756     nphdrs = prnsegs(as, 0) + 2; /* two CORE note sections */
1757
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);
1767
1768     ASSERT(nshdrs == 0 || nshdrs > 1);
1769
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;
1777
1778     phdrsz = nphdrs * sizeof (Phdr);
1779     shdrsz = nshdrs * sizeof (Shdr);
1780
1781     bigsize = MAX(sizeof (*bigwad), MAX(phdrsz, shdrsz));
1782     bigwad = kmem_alloc(bigsize, KM_SLEEP);
1783
1784     ehdr = &bigwad->ehdr;
1785     bzero(ehdr, sizeof (*ehdr));
1786
1787     ehdr->e_ident[EI_MAG0] = ELF_MAGIC0;
1788     ehdr->e_ident[EI_MAG1] = ELF_MAGIC1;
1789     ehdr->e_ident[EI_MAG2] = ELF_MAGIC2;
1790     ehdr->e_ident[EI_MAG3] = ELF_MAGIC3;
1791     ehdr->e_ident[EI_CLASS] = ELFCLASS;
1792     ehdr->e_type = ET_CORE;
1793
1794 #if !defined(_LP64) || defined(_ELF32_COMPAT)
1795 #if defined(__sparc)
1796     ehdr->e_ident[EI_DATA] = ELFDATA2MSB;
1797     ehdr->e_machine = EM_SPARC;
1798 #elif defined(__i386) || defined(__i386_COMPAT)
1799     ehdr->e_ident[EI_DATA] = ELFDATA2LSB;
1800     ehdr->e_machine = EM_386;
1801 #else
1802     #error "no recognized machine type is defined"
1803 #endif
1804 #endif
1805
1806 #else /* !defined(_LP64) || defined(_ELF32_COMPAT) */
1807 #if defined(__sparc)
1808     ehdr->e_ident[EI_DATA] = ELFDATA2MSB;
1809     ehdr->e_machine = EM_SPARCV9;
1810 #elif defined(__amd64)
1811

```

```

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
1817
1818 #endif /* !defined(_LP64) || defined(_ELF32_COMPAT) */
1819
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);

1829 if (nphdrs >= PN_XNUM)
1830     ehdr->e_phnum = PN_XNUM;
1831 else
1832     ehdr->e_phnum = (unsigned short)nphdrs;

1833 ehdr->e_phoff = sizeof (Ehdr);
1834 ehdr->e_phentsize = sizeof (Phdr);

1835 if (nshdrs > 0) {
1836     if (nshdrs >= SHN_LORESERVE)
1837         ehdr->e_shnum = 0;
1838     else
1839         ehdr->e_shnum = (unsigned short)nshdrs;

1840     if (nshdrs - 1 >= SHN_LORESERVE)
1841         ehdr->e_shstrndx = SHN_XINDEX;
1842     else
1843         ehdr->e_shstrndx = (unsigned short)(nshdrs - 1);

1844     ehdr->e_shoff = ehdr->e_phoff + ehdr->e_phentsize * nphdrs;
1845     ehdr->e_shentsize = sizeof (Shdr);
1846 }
1847
1848 if (error = core_write(vp, UIO_SYSSPACE, (offset_t)0, ehdr,
1849     sizeof (Ehdr), rlimit, credp))
1850     goto done;

1851 poffset = sizeof (Ehdr);
1852 softset = sizeof (Ehdr) + phdrsz;
1853 doffset = sizeof (Ehdr) + phdrsz + shdrsz;

1854 v = &bigwad->phdr[0];
1855 bzero(v, phdrsz);

1856 setup_old_note_header(&v[0], p);
1857 v[0].p_offset = doffset = roundup(doffset, sizeof (Word));
1858 doffset += v[0].p_filesz;

1859 setup_note_header(&v[1], p);
1860 v[1].p_offset = doffset = roundup(doffset, sizeof (Word));
1861 doffset += v[1].p_filesz;

1862 mutex_enter(&p->p_lock);

1863 brkbase = p->p_brkbase;
1864 brksize = p->p_brksize;
1865
1866 stkbase = p->p_usrstack - p->p_stksize;

```

```

1878     stksize = p->p_stksize;
1879
1880     mutex_exit(&p->p_lock);
1881
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;
1889
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;
1895
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;
1913
1914             /*
1915             * Figure out which mappings to include in the core.
1916             */
1917             type = segop_gettype(seg, saddr);
1918             type = SEGOP_GETTYPE(seg, saddr);
1919
1920             if (saddr == stkbase && size == stksize) {
1921                 if (!(content & CC_CONTENT_STACK))
1922                     goto exclude;
1923             } else if (saddr == brkbase && size == brksize) {
1924                 if (!(content & CC_CONTENT_HEAP))
1925                     goto exclude;
1926             } else if (seg->s_ops == &segspt_shmops) {
1927                 if (type & MAP_NORESERVE) {
1928                     if (!(content & CC_CONTENT_DISM))
1929                         goto exclude;
1930                 } else {
1931                     if (!(content & CC_CONTENT_ISM))
1932                         goto exclude;
1933                 }
1934             } else if (seg->s_ops != &segvn_ops) {
1935                 goto exclude;
1936             } else if (type & MAP_SHARED) {
1937                 if (shmgetid(p, saddr) != SHMID_NONE) {
1938                     if (!(content & CC_CONTENT_SHM))
1939                         goto exclude;
1940                 }
1941             }
1942         }
1943     }
1944
1945     if (error)
1946         goto error;
1947
1948     if (error = core_write(vp, UIO_SYSSPACE, (offset_t)0, v,
1949     sizeof (seg), rlimit, credp))
1950         goto error;
1951
1952     if (error = core_write(vp, UIO_SYSSPACE, (offset_t)0, &v[0],
1953     sizeof (seg), rlimit, credp))
1954         goto error;
1955
1956     if (error = core_write(vp, UIO_SYSSPACE, (offset_t)0, &v[1],
1957     sizeof (seg), rlimit, credp))
1958         goto error;
1959
1960     if (error = core_write(vp, UIO_SYSSPACE, (offset_t)0, &v[2],
1961     sizeof (seg), rlimit, credp))
1962         goto error;
1963
1964     if (error = core_write(vp, UIO_SYSSPACE, (offset_t)0, &v[3],
1965     sizeof (seg), rlimit, credp))
1966         goto error;
1967
1968     if (error = core_write(vp, UIO_SYSSPACE, (offset_t)0, &v[4],
1969     sizeof (seg), rlimit, credp))
1970         goto error;
1971
1972     if (error = core_write(vp, UIO_SYSSPACE, (offset_t)0, &v[5],
1973     sizeof (seg), rlimit, credp))
1974         goto error;
1975
1976     if (error = core_write(vp, UIO_SYSSPACE, (offset_t)0, &v[6],
1977     sizeof (seg), rlimit, credp))
1978         goto error;
1979
1980     if (error = core_write(vp, UIO_SYSSPACE, (offset_t)0, &v[7],
1981     sizeof (seg), rlimit, credp))
1982         goto error;
1983
1984     if (error = core_write(vp, UIO_SYSSPACE, (offset_t)0, &v[8],
1985     sizeof (seg), rlimit, credp))
1986         goto error;
1987
1988     if (error = core_write(vp, UIO_SYSSPACE, (offset_t)0, &v[9],
1989     sizeof (seg), rlimit, credp))
1990         goto error;
1991
1992     if (error = core_write(vp, UIO_SYSSPACE, (offset_t)0, &v[10],
1993     sizeof (seg), rlimit, credp))
1994         goto error;
1995
1996     if (error = core_write(vp, UIO_SYSSPACE, (offset_t)0, &v[11],
1997     sizeof (seg), rlimit, credp))
1998         goto error;
1999
2000     if (error = core_write(vp, UIO_SYSSPACE, (offset_t)0, &v[12],
2001     sizeof (seg), rlimit, credp))
2002         goto error;
2003
2004     if (error = core_write(vp, UIO_SYSSPACE, (offset_t)0, &v[13],
2005     sizeof (seg), rlimit, credp))
2006         goto error;
2007
2008     if (error = core_write(vp, UIO_SYSSPACE, (offset_t)0, &v[14],
2009     sizeof (seg), rlimit, credp))
2010         goto error;
2011
2012     if (error = core_write(vp, UIO_SYSSPACE, (offset_t)0, &v[15],
2013     sizeof (seg), rlimit, credp))
2014         goto error;
2015
2016     if (error = core_write(vp, UIO_SYSSPACE, (offset_t)0, &v[16],
2017     sizeof (seg), rlimit, credp))
2018         goto error;
2019
2020     if (error = core_write(vp, UIO_SYSSPACE, (offset_t)0, &v[17],
2021     sizeof (seg), rlimit, credp))
2022         goto error;
2023
2024     if (error = core_write(vp, UIO_SYSSPACE, (offset_t)0, &v[18],
2025     sizeof (seg), rlimit, credp))
2026         goto error;
2027
2028     if (error = core_write(vp, UIO_SYSSPACE, (offset_t)0, &v[19],
2029     sizeof (seg), rlimit, credp))
2030         goto error;
2031
2032     if (error = core_write(vp, UIO_SYSSPACE, (offset_t)0, &v[20],
2033     sizeof (seg), rlimit, credp))
2034         goto error;
2035
2036     if (error = core_write(vp, UIO_SYSSPACE, (offset_t)0, &v[21],
2037     sizeof (seg), rlimit, credp))
2038         goto error;
2039
2040     if (error = core_write(vp, UIO_SYSSPACE, (offset_t)0, &v[22],
2041     sizeof (seg), rlimit, credp))
2042         goto error;
2043
2044     if (error = core_write(vp, UIO_SYSSPACE, (offset_t)0, &v[23],
2045     sizeof (seg), rlimit, credp))
2046         goto error;
2047
2048     if (error = core_write(vp, UIO_SYSSPACE, (offset_t)0, &v[24],
2049     sizeof (seg), rlimit, credp))
2050         goto error;
2051
2052     if (error = core_write(vp, UIO_SYSSPACE, (offset_t)0, &v[25],
2053     sizeof (seg), rlimit, credp))
2054         goto error;
2055
2056     if (error = core_write(vp, UIO_SYSSPACE, (offset_t)0, &v[26],
2057     sizeof (seg), rlimit, credp))
2058         goto error;
2059
2060     if (error = core_write(vp, UIO_SYSSPACE, (offset_t)0, &v[27],
2061     sizeof (seg), rlimit, credp))
2062         goto error;
2063
2064     if (error = core_write(vp, UIO_SYSSPACE, (offset_t)0, &v[28],
2065     sizeof (seg), rlimit, credp))
2066         goto error;
2067
2068     if (error = core_write(vp, UIO_SYSSPACE, (offset_t)0, &v[29],
2069     sizeof (seg), rlimit, credp))
2070         goto error;
2071
2072     if (error = core_write(vp, UIO_SYSSPACE, (offset_t)0, &v[30],
2073     sizeof (seg), rlimit, credp))
2074         goto error;
2075
2076     if (error = core_write(vp, UIO_SYSSPACE, (offset_t)0, &v[31],
2077     sizeof (seg), rlimit, credp))
2078         goto error;
2079
2080     if (error = core_write(vp, UIO_SYSSPACE, (offset_t)0, &v[32],
2081     sizeof (seg), rlimit, credp))
2082         goto error;
2083
2084     if (error = core_write(vp, UIO_SYSSPACE, (offset_t)0, &v[33],
2085     sizeof (seg), rlimit, credp))
2086         goto error;
2087
2088     if (error = core_write(vp, UIO_SYSSPACE, (offset_t)0, &v[34],
2089     sizeof (seg), rlimit, credp))
2090         goto error;
2091
2092     if (error = core_write(vp, UIO_SYSSPACE, (offset_t)0, &v[35],
2093     sizeof (seg), rlimit, credp))
2094         goto error;
2095
2096     if (error = core_write(vp, UIO_SYSSPACE, (offset_t)0, &v[36],
2097     sizeof (seg), rlimit, credp))
2098         goto error;
2099
2100     if (error = core_write(vp, UIO_SYSSPACE, (offset_t)0, &v[37],
2101     sizeof (seg), rlimit, credp))
2102         goto error;
2103
2104     if (error = core_write(vp, UIO_SYSSPACE, (offset_t)0, &v[38],
2105     sizeof (seg), rlimit, credp))
2106         goto error;
2107
2108     if (error = core_write(vp, UIO_SYSSPACE, (offset_t)0, &v[39],
2109     sizeof (seg), rlimit, credp))
2110         goto error;
2111
2112     if (error = core_write(vp, UIO_SYSSPACE, (offset_t)0, &v[40],
2113     sizeof (seg), rlimit, credp))
2114         goto error;
2115
2116     if (error = core_write(vp, UIO_SYSSPACE, (offset_t)0, &v[41],
2117     sizeof (seg), rlimit, credp))
2118         goto error;
2119
2120     if (error = core_write(vp, UIO_SYSSPACE, (offset_t)0, &v[42],
2121     sizeof (seg), rlimit, credp))
2122         goto error;
2123
2124     if (error = core_write(vp, UIO_SYSSPACE, (offset_t)0, &v[43],
2125     sizeof (seg), rlimit, credp))
2126         goto error;
2127
2128     if (error = core_write(vp, UIO_SYSSPACE, (offset_t)0, &v[44],
2129     sizeof (seg), rlimit, credp))
2130         goto error;
2131
2132     if (error = core_write(vp, UIO_SYSSPACE, (offset_t)0, &v[45],
2133     sizeof (seg), rlimit, credp))
2134         goto error;
2135
2136     if (error = core_write(vp, UIO_SYSSPACE, (offset_t)0, &v[46],
2137     sizeof (seg), rlimit, credp))
2138         goto error;
2139
2140     if (error = core_write(vp, UIO_SYSSPACE, (offset_t)0, &v[47],
2141     sizeof (seg), rlimit, credp))
2142         goto error;
2143
2144     if (error = core_write(vp, UIO_SYSSPACE, (offset_t)0, &v[48],
2145     sizeof (seg), rlimit, credp))
2146         goto error;
2147
2148     if (error = core_write(vp, UIO_SYSSPACE, (offset_t)0, &v[49],
2149     sizeof (seg), rlimit, credp))
2150         goto error;
2151
2152     if (error = core_write(vp, UIO_SYSSPACE, (offset_t)0, &v[50],
2153     sizeof (seg), rlimit, credp))
2154         goto error;
2155
2156     if (error = core_write(vp, UIO_SYSSPACE, (offset_t)0, &v[51],
2157     sizeof (seg), rlimit, credp))
2158         goto error;
2159
2160     if (error = core_write(vp, UIO_SYSSPACE, (offset_t)0, &v[52],
2161     sizeof (seg), rlimit, credp))
2162         goto error;
2163
2164     if (error = core_write(vp, UIO_SYSSPACE, (offset_t)0, &v[53],
2165     sizeof (seg), rlimit, credp))
2166         goto error;
2167
2168     if (error = core_write(vp, UIO_SYSSPACE, (offset_t)0, &v[54],
2169     sizeof (seg), rlimit, credp))
2170         goto error;
2171
2172     if (error = core_write(vp, UIO_SYSSPACE, (offset_t)0, &v[55],
2173     sizeof (seg), rlimit, credp))
2174         goto error;
2175
2176     if (error = core_write(vp, UIO_SYSSPACE, (offset_t)0, &v[56],
2177     sizeof (seg), rlimit, credp))
2178         goto error;
2179
2180     if (error = core_write(vp, UIO_SYSSPACE, (offset_t)0, &v[57],
2181     sizeof (seg), rlimit, credp))
2182         goto error;
2183
2184     if (error = core_write(vp, UIO_SYSSPACE, (offset_t)0, &v[58],
2185     sizeof (seg), rlimit, credp))
2186         goto error;
2187
2188     if (error = core_write(vp, UIO_SYSSPACE, (offset_t)0, &v[59],
2189     sizeof (seg), rlimit, credp))
2190         goto error;
2191
2192     if (error = core_write(vp, UIO_SYSSPACE, (offset_t)0, &v[60],
2193     sizeof (seg), rlimit, credp))
2194         goto error;
2195
2196     if (error = core_write(vp, UIO_SYSSPACE, (offset_t)0, &v[61],
2197     sizeof (seg), rlimit, credp))
2198         goto error;
2199
2200     if (error = core_write(vp, UIO_SYSSPACE, (offset_t)0, &v[62],
2201     sizeof (seg), rlimit, credp))
2202         goto error;
2203
2204     if (error = core_write(vp, UIO_SYSSPACE, (offset_t)0, &v[63],
2205     sizeof (seg), rlimit, credp))
2206         goto error;
2207
2208     if (error = core_write(vp, UIO_SYSSPACE, (offset_t)0, &v[64],
2209     sizeof (seg), rlimit, credp))
2210         goto error;
2211
2212     if (error = core_write(vp, UIO_SYSSPACE, (offset_t)0, &v[65],
2213     sizeof (seg), rlimit, credp))
2214         goto error;
2215
2216     if (error = core_write(vp, UIO_SYSSPACE, (offset_t)0, &v[66],
2217     sizeof (seg), rlimit, credp))
2218         goto error;
2219
2220     if (error = core_write(vp, UIO_SYSSPACE, (offset_t)0, &v[67],
2221     sizeof (seg), rlimit, credp))
2222         goto error;
2223
2224     if (error = core_write(vp, UIO_SYSSPACE, (offset_t)0, &v[68],
2225     sizeof (seg), rlimit, credp))
2226         goto error;
2227
2228     if (error = core_write(vp, UIO_SYSSPACE, (offset_t)0, &v[69],
2229     sizeof (seg), rlimit, credp))
2230         goto error;
2231
2232     if (error = core_write(vp, UIO_SYSSPACE, (offset_t)0, &v[70],
2233     sizeof (seg), rlimit, credp))
2234         goto error;
2235
2236     if (error = core_write(vp, UIO_SYSSPACE, (offset_t)0, &v[71],
2237     sizeof (seg), rlimit, credp))
2238         goto error;
2239
2240     if (error = core_write(vp, UIO_SYSSPACE, (offset_t)0, &v[72],
2241     sizeof (seg), rlimit, credp))
2242         goto error;
2243
2244     if (error = core_write(vp, UIO_SYSSPACE, (offset_t)0, &v[73],
2245     sizeof (seg), rlimit, credp))
2246         goto error;
2247
2248     if (error = core_write(vp, UIO_SYSSPACE, (offset_t)0, &v[74],
2249     sizeof (seg), rlimit, credp))
2250         goto error;
2251
2252     if (error = core_write(vp, UIO_SYSSPACE, (offset_t)0, &v[75],
2253     sizeof (seg), rlimit, credp))
2254         goto error;
2255
2256     if (error = core_write(vp, UIO_SYSSPACE, (offset_t)0, &v[76],
2257     sizeof (seg), rlimit, credp))
2258         goto error;
2259
2260     if (error = core_write(vp, UIO_SYSSPACE, (offset_t)0, &v[77],
2261     sizeof (seg), rlimit, credp))
2262         goto error;
2263
2264     if (error = core_write(vp, UIO_SYSSPACE, (offset_t)0, &v[78],
2265     sizeof (seg), rlimit, credp))
2266         goto error;
2267
2268     if (error = core_write(vp, UIO_SYSSPACE, (offset_t)0, &v[79],
2269     sizeof (seg), rlimit, credp))
2270         goto error;
2271
2272     if (error = core_write(vp, UIO_SYSSPACE, (offset_t)0, &v[80],
2273     sizeof (seg), rlimit, credp))
2274         goto error;
2275
2276     if (error = core_write(vp, UIO_SYSSPACE, (offset_t)0, &v[81],
2277     sizeof (seg), rlimit, credp))
2278         goto error;
2279
2280     if (error = core_write(vp, UIO_SYSSPACE, (offset_t)0, &v[82],
2281     sizeof (seg), rlimit, credp))
2282         goto error;
2283
2284     if (error = core_write(vp, UIO_SYSSPACE, (offset_t)0, &v[83],
2285     sizeof (seg), rlimit, credp))
2286         goto error;
2287
2288     if (error = core_write(vp, UIO_SYSSPACE, (offset_t)0, &v[84],
2289     sizeof (seg), rlimit, credp))
2290         goto error;
2291
2292     if (error = core_write(vp, UIO_SYSSPACE, (offset_t)0, &v[85],
2293     sizeof (seg), rlimit, credp))
2294         goto error;
2295
2296     if (error = core_write(vp, UIO_SYSSPACE, (offset_t)0, &v[86],
2297     sizeof (seg), rlimit, credp))
2298         goto error;
2299
2300     if (error = core_write(vp, UIO_SYSSPACE, (offset_t)0, &v[87],
2301     sizeof (seg), rlimit, credp))
2302         goto error;
2303
2304     if (error = core_write(vp, UIO_SYSSPACE, (offset_t)0, &v[88],
2305     sizeof (seg), rlimit, credp))
2306         goto error;
2307
2308     if (error = core_write(vp, UIO_SYSSPACE, (offset_t)0, &v[89],
2309     sizeof (seg), rlimit, credp))
2310         goto error;
2311
2312     if (error = core_write(vp, UIO_SYSSPACE, (offset_t)0, &v[90],
2313     sizeof (seg), rlimit, credp))
2314         goto error;
2315
2316     if (error = core_write(vp, UIO_SYSSPACE, (offset_t)0, &v[91],
2317     sizeof (seg), rlimit, credp))
2318         goto error;
2319
2320     if (error = core_write(vp, UIO_SYSSPACE, (offset_t)0, &v[92],
2321     sizeof (seg), rlimit, credp))
2322         goto error;
2323
2324     if (error = core_write(vp, UIO_SYSSPACE, (offset_t)0, &v[93],
2325     sizeof (seg), rlimit, credp))
2326         goto error;
2327
2328     if (error = core_write(vp, UIO_SYSSPACE, (offset_t)0, &v[94],
2329     sizeof (seg), rlimit, credp))
2330         goto error;
2331
2332     if (error = core_write(vp, UIO_SYSSPACE, (offset_t)0, &v[95],
2333     sizeof (seg), rlimit, credp))
2334         goto error;
2335
2336     if (error = core_write(vp, UIO_SYSSPACE, (offset_t)0, &v[96],
2337     sizeof (seg), rlimit, credp))
2338         goto error;
2339
2340     if (error = core_write(vp, UIO_SYSSPACE, (offset_t)0, &v[97],
2341     sizeof (seg), rlimit, credp))
2342         goto error;
2343
2344     if (error = core_write(vp, UIO_SYSSPACE, (offset_t)0, &v[98],
2345     sizeof (seg), rlimit, credp))
2346         goto error;
2347
2348     if (error = core_write(vp, UIO_SYSSPACE, (offset_t)0, &v[99],
2349     sizeof (seg), rlimit, credp))
2350         goto error;
2351
2352     if (error = core_write(vp, UIO_SYSSPACE, (offset_t)0, &v[100],
2353     sizeof (seg), rlimit, credp))
2354         goto error;
2355
2356     if (error = core_write(vp, UIO_SYSSPACE, (offset_t)0, &v[101],
2357     sizeof (seg), rlimit, credp))
2358         goto error;
2359
2360     if (error = core_write(vp, UIO_SYSSPACE, (offset_t)0, &v[102],
2361     sizeof (seg), rlimit, credp))
2362         goto error;
2363
2364     if (error = core_write(vp, UIO_SYSSPACE, (offset_t)0, &v[103],
2365     sizeof (seg), rlimit, credp))
2366         goto error;
2367
2368     if (error = core_write(vp, UIO_SYSSPACE, (offset_t)0, &v[104],
2369     sizeof (seg), rlimit, credp))
2370         goto error;
2371
2372     if (error = core_write(vp, UIO_SYSSPACE, (offset_t)0, &v[105],
2373     sizeof (seg), rlimit, credp))
2374         goto error;
2375
2376     if (error = core_write(vp, UIO_SYSSPACE, (offset_t)0, &v[106],
2377     sizeof (seg), rlimit, credp))
2378         goto error;
2379
2380     if (error = core_write(vp, UIO_SYSSPACE, (offset_t)0, &v[107],
2381     sizeof (seg), rlimit, credp))
2382         goto error;
2383
2384     if (error = core_write(vp, UIO_SYSSPACE, (offset_t)0, &v[108],
2385     sizeof (seg), rlimit, credp))
2386         goto error;
2387
2388     if (error = core_write(vp, UIO_SYSSPACE, (offset_t)0, &v[109],
2389     sizeof (seg), rlimit, credp))
2390         goto error;
2391
2392     if (error = core_write(vp, UIO_SYSSPACE, (offset_t)0, &v[110],
2393     sizeof (seg), rlimit, credp))
2394         goto error;
2395
2396     if (error = core_write(vp, UIO_SYSSPACE, (offset_t)0, &v[111],
2397     sizeof (seg), rlimit, credp))
2398         goto error;
2399
2400     if (error = core_write(vp, UIO_SYSSPACE, (offset_t)0, &v[112],
2401     sizeof (seg), rlimit, credp))
2402         goto error;
2403
2404     if (error = core_write(vp, UIO_SYSSPACE, (offset_t)0, &v[113],
2405     sizeof (seg), rlimit, credp))
2406         goto error;
2407
2408     if (error = core_write(vp, UIO_SYSSPACE, (offset_t)0, &v[114],
2409     sizeof (seg), rlimit, credp))
2410         goto error;
2411
2412     if (error = core_write(vp, UIO_SYSSPACE, (offset_t)0, &v[115],
2413     sizeof (seg), rlimit, credp))
2414         goto error;
2415
2416     if (error = core_write(vp, UIO_SYSSPACE, (offset_t)0, &v[116],
2417     sizeof (seg), rlimit, credp))
2418         goto error;
2419
2420     if (error = core_write(vp, UIO_SYSSPACE, (offset_t)0, &v[117],
2421     sizeof (seg), rlimit, credp))
2422         goto error;
2423
2424     if (error = core_write(vp, UIO_SYSSPACE, (offset_t)0, &v[118],
2425     sizeof (seg), rlimit, credp))
2426         goto error;
2427
2428     if (error = core_write(vp, UIO_SYSSPACE, (offset_t)0, &v[119],
2429     sizeof (seg), rlimit, credp))
2430         goto error;
2431
2432     if (error = core_write(vp, UIO_SYSSPACE, (offset_t)0, &v[120],
2433     sizeof (seg), rlimit, credp))
2434         goto error;
2435
2436     if (error = core_write(vp, UIO_SYSSPACE, (offset_t)0, &v[121],
2437     sizeof (seg), rlimit, credp))
2438         goto error;
2439
2440     if (error = core_write(vp, UIO_SYSSPACE, (offset_t)0, &v[122],
2441     sizeof (seg), rlimit, credp))
2442         goto error;
2443
2444     if (error = core_write(vp, UIO_SYSSPACE, (offset_t)0, &v[123],
2445     sizeof (seg), rlimit, credp))
2446         goto error;
2447
2448     if (error = core_write(vp, UIO_SYSSPACE, (offset_t)0, &v[124],
2449     sizeof (seg), rlimit, credp))
2450         goto error;
2451
2452     if (error = core_write(vp, UIO_SYSSPACE, (offset_t)0, &v[125],
2453     sizeof (seg), rlimit, credp))
2454         goto error;
2455
2456     if (error = core_write(vp, UIO_SYSSPACE, (offset_t)0, &v[126],
2457     sizeof (seg), rlimit, credp))
2458         goto error;
2459
2460     if (error = core_write(vp, UIO_SYSSPACE, (offset_t)0, &v[127],
2461     sizeof (seg), rlimit, credp))
2462         goto error;
2463
2464     if (error = core_write(vp, UIO_SYSSPACE, (offset_t)0, &v[128],
2465     sizeof (seg), rlimit, credp))
2466         goto error;
2467
2468     if (error = core_write(vp, UIO_SYSSPACE, (offset_t)0, &v[129],
2469     sizeof (seg), rlimit, credp))
2470         goto error;
2471
2472     if (error = core_write(vp, UIO_SYSSPACE, (offset_t)0, &v[130],
2473     sizeof (seg), rlimit, credp))
2474         goto error;
2475
2476     if (error = core_write(vp, UIO_SYSSPACE, (offset_t)0, &v[131],
2477     sizeof (seg), rlimit, credp))
2478         goto error;
2479
2480     if (error = core_write(vp, UIO_SYSSPACE, (offset_t)0, &v[132],
2481     sizeof (seg), rlimit, credp))
2482         goto error;
2483
2484     if (error = core_write(vp, UIO_SYSSPACE, (offset_t)0, &v[133],
2485     sizeof (seg), rlimit, credp))
2486         goto error;
2487
2488     if (error = core_write(vp, UIO_SYSSPACE, (offset_t)0, &v[134],
2489     sizeof (seg), rlimit, credp))
2490         goto error;
2491
2492     if (error = core_write(vp, UIO_SYSSPACE, (offset_t)0, &v[135],
2493     sizeof (seg), rlimit, credp))
2494         goto error;
2495
2496     if (error = core_write(vp, UIO_SYSSPACE, (offset_t)0, &v[136],
2497     sizeof (seg), rlimit, credp))
2498         goto error;
2499
2500     if (error = core_write(vp, UIO_SYSSPACE, (offset_t)0, &v[137],
2501     sizeof (seg), rlimit, credp))
2502         goto error;
2503
2504     if (error = core_write(vp, UIO_SYSSPACE, (offset_t)0, &v[138],
2505     sizeof (seg), rlimit, credp))
2506         goto error;
2507
2508     if (error = core_write(vp, UIO_SYSSPACE, (offset_t)0, &v[139],
2509     sizeof (seg), rlimit, credp))
2510         goto error;
2511
2512     if (error = core_write(vp, UIO_SYSSPACE, (offset_t)0, &v[140],
2513     sizeof (seg), rlimit, credp))
2514         goto error;
2515
2516     if (error = core_write(vp, UIO_SYSSPACE, (offset_t)0, &v[141],
2517     sizeof (seg), rlimit, credp))
2518         goto error;
2519
2520     if (error = core_write(vp, UIO_SYSSPACE, (offset_t)0, &v[142],
2521     sizeof (seg), rlimit, credp))
2522         goto error;
2523
2524     if (error = core_write(vp, UIO_SYSSPACE, (offset_t)0, &v[143],
2525     sizeof (seg), rlimit, credp))
2526         goto error;
2527
2528     if (error = core_write(vp, UIO_SYSSPACE, (offset_t)0, &v[144],
2529     sizeof (seg), rlimit, credp))
2530         goto error;
2531
2532     if (error = core_write(vp, UIO_SYSSPACE, (offset_t)0, &v[145],
2533     sizeof (seg), rlimit, credp))
2534         goto error;
2535
2536     if (error = core_write(vp, UIO_SYSSPACE, (offset_t)0, &v[146],
2537     sizeof (seg), rlimit, credp))
2538         goto error;
2539
2540     if (error = core_write(vp, UIO_SYSSPACE, (offset_t)0, &v[147],
2541     sizeof (seg), rlimit, credp))
2542         goto error;
2543
2544     if (error = core_write(vp, UIO_SYSSPACE, (offset_t)0, &v[148],
2545     sizeof (seg), rlimit, credp))
2546         goto error;
2547
2548     if (error = core_write(vp, UIO_SYSSPACE, (offset_t)0, &v[149],
2549     sizeof (seg), rlimit, credp))
2550         goto error;
2551
2552     if (error = core_write(vp, UIO_SYSSPACE, (offset_t)0, &v[150],
2553     sizeof (seg), rlimit, credp))
2554         goto error;
2555
2556     if (error = core_write(vp, UIO_SYSSPACE, (offset_t)0, &v[151],
2557     sizeof (seg), rlimit, credp))
2558         goto error;
2559
2560     if (error = core_write(vp, UIO_SYSSPACE, (offset_t)0,
```

```

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;
1949
1950     } else {
1951         if (!(content & CC_CONTENT_SHFILE))
1952             goto exclude;
1953     }
1954
1955     } else if (segop_getvp(seg, seg->s_base, &mvp) != 0 ||
1956     } else if (SEGOP_GETVP(seg, seg->s_base, &mvp) != 0 ||
1957         mvp == NULL || mvp->v_type != VREG) {
1958         if (!(content & CC_CONTENT_ANON))
1959             goto exclude;
1960
1961     } else if (prot == (PROT_READ | PROT_EXEC)) {
1962         if (!(content & CC_CONTENT_TEXT))
1963             goto exclude;
1964
1965     } else if (prot == PROT_READ) {
1966         if (!(content & CC_CONTENT_RODATA))
1967             goto exclude;
1968
1969     } else {
1970         if (!(content & CC_CONTENT_DATA))
1971             goto exclude;
1972     }
1973
1974     doffset = roundup(doffset, sizeof (Word));
1975     v[i].p_offset = doffset;
1976     v[i].p_filesz = size;
1977     doffset += size;
1978
1979     exclude:
1980     i++;
1981     ASSERT(tmp == NULL);
1982     AS_LOCK_EXIT(as, &as->a_lock);
1983
1984     if (overflow || i != nphdrs) {
1985         if (ntries++ == 0) {
1986             kmem_free(bigmawd, 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     }
1995
1996     if ((error = core_write(vp, UIO_SYSSPACE, poffset,
1997         v, phdrsz, rlimit, credp)) != 0)
1998         goto done;
1999
2000     if ((error = write_old_elfnotes(p, sig, vp, v[0].p_offset, rlimit,
2001         credp)) != 0)
2002         goto done;
2003
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;
2012
2013         if (v[i].p_filesz == 0)
2014             continue;
2015
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
2029         if ((sig = lwp->lwp_cursig) == 0) {
2030             /*
2031              * We failed due to something other than a signal.
2032              * Since the space reserved for the segment is now
2033              * unused, we stash the errno in the first four
2034              * bytes. This undocumented interface will let us
2035              * understand the nature of the failure.
2036             */
2037             (void) core_write(vp, UIO_SYSSPACE, v[i].p_offset,
2038                 &error, sizeof (error), rlimit, credp);
2039
2040             v[i].p_filesz = 0;
2041             v[i].p_flags |= PF_SUNW_FAILURE;
2042             if ((error = core_write(vp, UIO_SYSSPACE,
2043                 poffset + sizeof (v[i]) * i, &v[i], sizeof (v[i]),
2044                 rlimit, credp)) != 0)
2045                 goto done;
2046
2047             continue;
2048         }
2049
2050         /*
2051          * We took a signal. We want to abort the dump entirely, but
2052          * we also want to indicate what failed and why. We therefore
2053          * use the space reserved for the first failing segment to
2054          * write our error (which, for purposes of compatibility with
2055          * older core dump readers, we set to EINTR) followed by any
2056          * siginfo associated with the signal.
2057         */
2058         bzero(&killinfo, sizeof (killinfo));
2059         killinfo.prk_error = EINTR;
2060
2061         sq = sig == SIGKILL ? curproc->p_killsq : lwp->lwp_curinfo;
2062
2063         if (sq != NULL) {
2064             bcopy(&sq->sq_info, &killinfo.prk_info,
2065                   sizeof (sq->sq_info));
2066         } else {
2067             killinfo.prk_info.si_signo = lwp->lwp_cursig;
2068             killinfo.prk_info.si_code = SI_NOINFO;
2069         }
2070
2071     #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;
2081
2082         siginfo_kto32((k_siginfo_t *)&killinfo.prk_info, &si32);
2083         bcopy(&si32, &killinfo.prk_info, sizeof (si32));
2084     }
2085 #endif
2086
2087     (void) core_write(vp, UIO_SYSSPACE, v[i].p_offset,
2088                      &killinfo, sizeof (killinfo), rlimit, credp);
2089
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;
2097
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     }
2106
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;
2114
2115     break;
2116 }
2117
2118 if (nshdrs > 0) {
2119     bzero(&bigwad->shdr[0], shdrsz);
2120
2121     if (nshdrs >= SHN_LORESERVE)
2122         bigwad->shdr[0].sh_size = nshdrs;
2123
2124     if (nshdrs - 1 >= SHN_LORESERVE)
2125         bigwad->shdr[0].sh_link = nshdrs - 1;
2126
2127     if (nphdrs >= PN_XNUM)
2128         bigwad->shdr[0].sh_info = nphdrs;
2129
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     }
2140 }
2141
2142     if ((error = core_write(vp, UIO_SYSSPACE, soffset,
2143                         &bigwad->shdr[0], shdrsz, rlimit, credp)) != 0)
2144     goto done;
2145
2146 done:
2147     kmem_free(bigwad, bigsize);
2148     return (error);
2149 }
```

```

2139 }
2140
2141     if ((error = core_write(vp, UIO_SYSSPACE, soffset,
2142                         &bigwad->shdr[0], shdrsz, rlimit, credp)) != 0)
2143         goto done;
2144
2145 done:
2146     kmem_free(bigwad, bigsize);
2147     return (error);
2148
2149 }
```

*unchanged portion omitted*

new/usr/src/uts/common/fs/nfs/nfs3\_vnops.c

1

```
*****  
171811 Tue Nov 24 09:34:44 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;  
5564  
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);  
5572  
5573     rp = VTOR(vp);  
5574  
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()  
5580      * - as_free()->SEGOP_UNMAP()->segvn_unmap()->VOP_DELMAP()->nfs3_delmap()  
5581      * - as_unmap()->SEGOP_UNMAP()->segvn_unmap()->VOP_DELMAP()->nfs3_delmap()  
5582  
5583      * With the use of address space callbacks we are allowed to drop the  
5584      * address space lock, a_lock, while executing the NFS operations that  
5585      * need to go over the wire. Returning EAGAIN to the caller of this  
5586      * function is what drives the execution of the callback that we add  
5587      * below. The callback will be executed by the address space code  
5588      * after dropping the a_lock. When the callback is finished, since  
5589      * we dropped the a_lock, it must be re-acquired and segvn_unmap()  
5590      * is called again on the same segment to finish the rest of the work  
5591      * that needs to happen during unmapping.  
5592      *  
5593      * This action of calling back into the segment driver causes  
5594      * nfs3_delmap() to get called again, but since the callback was  
5595      * already executed at this point, it already did the work and there  
5596      * is nothing left for us to do.  
5597      * To Summarize:  
5598      * - The first time nfs3_delmap is called by the current thread is when  
5599      * we add the caller associated with this delmap to the delmap caller  
5600      * list, add the callback, and return EAGAIN.
```

new/usr/src/uts/common/fs/nfs/nfs3\_vnops.c

2

```
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             }  
5619             /* current caller was not in the list */  
5620             delmap_call = nfs_init_delmapcall();  
5621  
5622             mutex_enter(&rp->r_statelock);  
5623             list_insert_tail(&rp->r_indelmap, delmap_call);  
5624             mutex_exit(&rp->r_statelock);  
5625  
5626             dmapp = kmalloc(sizeof (nfs_delmap_args_t), KM_SLEEP);  
5627  
5628             dmapp->vp = vp;  
5629             dmapp->off = off;  
5630             dmapp->addr = addr;  
5631             dmapp->len = len;  
5632             dmapp->prot = prot;  
5633             dmapp->maxprot = maxprot;  
5634             dmapp->flags = flags;  
5635             dmapp->cr = cr;  
5636             dmapp->caller = delmap_call;  
5637  
5638             error = as_add_callback(as, nfs3_delmap_callback, dmapp,  
5639                         AS_UNMAP_EVENT, addr, len, KM_SLEEP);  
5640  
5641             return (error ? error : EAGAIN);  
5642  
5643 }  
_____ unchanged_portion_omitted _____
```

```
*****
429800 Tue Nov 24 09:34:44 2015
new/usr/src/uts/common/fs/nfs/nfs4_vnops.c
patch lower-case-segops
*****
_____unchanged_portion_omitted_____
11026 /*
11027 * Setup and add an address space callback to do the work of the delmap call.
11028 * The callback will (and must be) deleted in the actual callback function.
11029 *
11030 * This is done in order to take care of the problem that we have with holding
11031 * the address space's a_lock for a long period of time (e.g. if the NFS server
11032 * is down). Callbacks will be executed in the address space code while the
11033 * a_lock is not held. Holding the address space's a_lock causes things such
11034 * as ps and fork to hang because they are trying to acquire this lock as well.
11035 */
11036 /* ARGSUSED */
11037 static int
11038 nfs4_delmap(vnode_t *vp, offset_t off, struct as *as, caddr_t addr,
11039     size_t len, uint_t prot, uint_t maxprot, uint_t flags, cred_t *cr,
11040     caller_context_t *ct)
11041 {
11042     int             caller_found;
11043     int             error;
11044     rnode4_t        *rp;
11045     nfs4_delmap_args_t *dmapp;
11046     nfs4_delmapcall_t *delmap_call;
11047
11048     if (vp->v_flag & VNOMAP)
11049         return (ENOSYS);
11050
11051     /*
11052     * A process may not change zones if it has NFS pages mmap'ed
11053     * in, so we can't legitimately get here from the wrong zone.
11054     */
11055     ASSERT(nfs_zone() == VTOMI4(vp)->mi_zone);
11056
11057     rp = VTOR4(vp);
11058
11059     /*
11060     * The way that the address space of this process deletes its mapping
11061     * of this file is via the following call chains:
11062     * - as_free()->segop_unmap()->segvn_unmap()->VOP_DELMAP()->nfs4_delmap()
11063     * - as_unmap()->segop_unmap()->segvn_unmap()->VOP_DELMAP()->nfs4_delmap()
11064     * - as_free()->SEGOP_UNMAP()->segvn_unmap()->VOP_DELMAP()->nfs4_delmap()
11065     * - as_unmap()->SEGOP_UNMAP()->segvn_unmap()->VOP_DELMAP()->nfs4_delmap()
11066
11067     * With the use of address space callbacks we are allowed to drop the
11068     * address space lock, a_lock, while executing the NFS operations that
11069     * need to go over the wire. Returning EAGAIN to the caller of this
11070     * function is what drives the execution of the callback that we add
11071     * below. The callback will be executed by the address space code
11072     * after dropping the a_lock. When the callback is finished, since
11073     * we dropped the a_lock, it must be re-acquired and segvn_unmap()
11074     * is called again on the same segment to finish the rest of the work
11075     * that needs to happen during unmapping.
11076
11077     * This action of calling back into the segment driver causes
11078     * nfs4_delmap() to get called again, but since the callback was
11079     * already executed at this point, it already did the work and there
11080     * is nothing left for us to do.
11081
11082     * To Summarize:
11083     * - The first time nfs4_delmap is called by the current thread is when
11084     * we add the caller associated with this delmap to the delmap caller
```

```
11083     * list, add the callback, and return EAGAIN.
11084     * - The second time in this call chain when nfs4_delmap is called we
11085     * will find this caller in the delmap caller list and realize there
11086     * is no more work to do thus removing this caller from the list and
11087     * returning the error that was set in the callback execution.
11088     */
11089     caller_found = nfs4_find_and_delete_delmapcall(rp, &error);
11090     if (caller_found) {
11091         /*
11092         * 'error' is from the actual delmap operations. To avoid
11093         * hangs, we need to handle the return of EAGAIN differently
11094         * since this is what drives the callback execution.
11095         * In this case, we don't want to return EAGAIN and do the
11096         * callback execution because there are none to execute.
11097         */
11098     if (error == EAGAIN)
11099         return (0);
11100     else
11101         return (error);
11102     }
11103
11104     /* current caller was not in the list */
11105     delmap_call = nfs4_init_delmapcall();
11106
11107     mutex_enter(&rp->r_statelock);
11108     list_insert_tail(&rp->r_indelmap, delmap_call);
11109     mutex_exit(&rp->r_statelock);
11110
11111     dmapp = kmem_alloc(sizeof(nfs4_delmap_args_t), KM_SLEEP);
11112
11113     dmapp->vp = vp;
11114     dmapp->off = off;
11115     dmapp->addr = addr;
11116     dmapp->len = len;
11117     dmapp->prot = prot;
11118     dmapp->maxprot = maxprot;
11119     dmapp->flags = flags;
11120     dmapp->cr = cr;
11121     dmapp->caller = delmap_call;
11122
11123     error = as_add_callback(as, nfs4_delmap_callback, dmapp,
11124                           AS_UNMAP_EVENT, addr, len, KM_SLEEP);
11125
11126     return (error ? error : EAGAIN);
11127 }
11128
11129     _____unchanged_portion_omitted_____
```

```
*****
130899 Tue Nov 24 09:34:45 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()
4677      * - as_free()->SEGOP_UNMAP()->segvn_unmap()->VOP_DELMAP()->nfs_delmap()
4678      * - as_unmap()->SEGOP_UNMAP()->segvn_unmap()->VOP_DELMAP()->nfs_delmap()
4679      *
4680      * With the use of address space callbacks we are allowed to drop the
4681      * address space lock, a_lock, while executing the NFS operations that
4682      * need to go over the wire. Returning EAGAIN to the caller of this
4683      * function is what drives the execution of the callback that we add
4684      * below. The callback will be executed by the address space code
4685      * after dropping the a_lock. When the callback is finished, since
4686      * we dropped the a_lock, it must be re-acquired and segvn_unmap()
4687      * is called again on the same segment to finish the rest of the work
4688      * that needs to happen during unmapping.
4689      *
4690      * This action of calling back into the segment driver causes
4691      * nfs_delmap() to get called again, but since the callback was
4692      * already executed at this point, it already did the work and there
4693      * is nothing left for us to do.
4694      *
4695      * To Summarize:
4696      * - The first time nfs_delmap is called by the current thread is when
4697      * we add the caller associated with this delmap to the delmap caller
4698      * list, add the callback, and return EAGAIN.
4699 
```

```
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 = kmalloc(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);
4738
4739     return (error ? error : EAGAIN);
4740 }
```

\_\_\_\_\_ unchanged\_portion\_omitted\_

```
*****
93906 Tue Nov 24 09:34:45 2015
new/usr/src/uts/common/fs/proc/priocctl.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;
3130
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 &&
3150         SEGOP_GETVP(seg, va, &xvp) == 0 &&
3151         xvp != NULL &&
3152         xvp->v_type == VREG) {
3153         VN_HOLD(xvp);
3154     } else {
3155         error = EINVAL;
3156     }
3157     AS_LOCK_EXIT(as, &as->a_lock);
3158     mutex_enter(&p->p_lock);
3159 } else if ((xvp = p->p_exec) == NULL) {
3160     error = EINVAL;
3161 } else {
3162     VN_HOLD(xvp);
3163 }
3164 prunlock(pnp);
3165
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_RELSE(xvp);
3171     } else {
3172         *rvalp = n;
3173     }
3174 }
```

```
3176         return (error);
3177     }
3178     _____unchanged_portion_omitted_____
3179
3180     /*
3181      * Return an array of structures with memory map information.
3182      * We allocate here; the caller must deallocate.
3183      * The caller is also responsible to append the zero-filled entry
3184      * that terminates the PIOCMAP output buffer.
3185      */
3186     static int
3187     oprgetmap(proc_t *p, list_t *iolhead)
3188     {
3189         struct as *as = p->p_as;
3190         prmap_t *mp;
3191         struct seg *seg;
3192         struct seg *brkseg, *stkseg;
3193         uint_t prot;
3194
3195         ASSERT(as != &kas && AS_WRITE_HELD(as, &as->a_lock));
3196
3197         /*
3198          * Request an initial buffer size that doesn't waste memory
3199          * if the address space has only a small number of segments.
3200          */
3201         pr_iol_initlist(iolhead, sizeof (*mp), avl_numnodes(&as->a_segtree));
3202
3203         if ((seg = AS_SEGFIRST(as)) == NULL)
3204             return (0);
3205
3206         brkseg = break_seg(p);
3207         stkseg = as_segat(as, prgetstackbase(p));
3208
3209         do {
3210             caddr_t eaddr = seg->s_base + pr_getsegsize(seg, 0);
3211             caddr_t saddr, naddr;
3212             void *tmp = NULL;
3213
3214             for (saddr = seg->s_base; saddr < eaddr; saddr = naddr) {
3215                 prot = pr_getprot(seg, 0, &tmp, &saddr, &naddr, eaddr);
3216                 if (saddr == naddr)
3217                     continue;
3218
3219                 mp = pr_iol_newbuf(iolhead, sizeof (*mp));
3220
3221                 mp->pr_vaddr = saddr;
3222                 mp->pr_size = naddr - saddr;
3223                 mp->pr_off = segop_getoffset(seg, saddr);
3224                 mp->pr_off = SEGOP_GETOFFSET(seg, saddr);
3225                 mp->pr_mflags = 0;
3226                 if (prot & PROT_READ)
3227                     mp->pr_mflags |= MA_READ;
3228                 if (prot & PROT_WRITE)
3229                     mp->pr_mflags |= MA_WRITE;
3230                 if (prot & PROT_EXEC)
3231                     mp->pr_mflags |= MA_EXEC;
3232
3233                 if (segop_gettype(seg, saddr) & MAP_SHARED)
3234                     if (SEGOP_GETTYPE(seg, saddr) & MAP_SHARED)
3235                         mp->pr_mflags |= MA_SHARED;
3236
3237                 if (seg == brkseg)
3238                     mp->pr_mflags |= MA_BREAK;
3239                 else if (seg == stkseg)
3240                     mp->pr_mflags |= MA_STACK;
3241
3242                 mp->pr_pagesize = PAGESIZE;
3243             }
3244         }
```

```

3570             ASSERT(tmp == NULL);
3571         } while ((seg = AS_SEGNEXT(as, seg)) != NULL);
3573
3574     return (0);
3575 }
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;
3585
3586     ASSERT(as != &kas && AS_WRITE_HELD(as, &as->a_lock));
3587
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));
3593
3594     if ((seg = AS_SEGFIRST(as)) == NULL)
3595         return (0);
3596
3597     brkseg = break_seg(p);
3598     stkseg = as_segat(as, prgetstackbase(p));
3599
3600     do {
3601         caddr_t eaddr = seg->s_base + pr_getsegsize(seg, 0);
3602         caddr_t saddr, naddr;
3603         void *tmp = NULL;
3604
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;
3609
3610             mp = pr_iol_newbuf(iolhead, sizeof (*mp));
3611
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);
3615             mp->pr_off = (off32_t)SEGOP_GETOFFSET(seg, saddr);
3616             mp->pr_mflags = 0;
3617             if (prot & PROT_READ)
3618                 mp->pr_mflags |= MA_READ;
3619             if (prot & PROT_WRITE)
3620                 mp->pr_mflags |= MA_WRITE;
3621             if (prot & PROT_EXEC)
3622                 mp->pr_mflags |= MA_EXEC;
3623             if (segop_gettype(seg, saddr) & MAP_SHARED)
3624                 if (SEGOP_GETTYPE(seg, saddr) & MAP_SHARED)
3625                     mp->pr_mflags |= MA_SHARED;
3626             if (seg == brkseg)
3627                 mp->pr_mflags |= MA_BREAK;
3628             else if (seg == stkseg)
3629                 mp->pr_mflags |= MA_STACK;
3630             mp->pr_pagesize = PAGESIZE;
3631
3632         }
3633         ASSERT(tmp == NULL);
3634     } while ((seg = AS_SEGNEXT(as, seg)) != NULL);
3635
3636     return (0);

```

```

3634 }
3635 unchanged portion omitted
3636 #endif /* _SYSCALL32_IMPL */
3637
3638 /*
3639  * Read old /proc page data information.
3640  */
3641 int
3642 oprpdread(struct as *as, uint_t hatid, struct uio *uiop)
3643 {
3644     caddr_t buf;
3645     size_t size;
3646     prpageheader_t *php;
3647     prasmmap_t *pmp;
3648     struct seg *seg;
3649     int error;
3650
3651     again:
3652     AS_LOCK_ENTER(as, &as->a_lock, RW_WRITER);
3653
3654     if ((seg = AS_SEGFIRST(as)) == NULL) {
3655         AS_LOCK_EXIT(as, &as->a_lock);
3656         return (0);
3657     }
3658     size = oprpdsize(as);
3659     if (uiop->uio_resid < size) {
3660         AS_LOCK_EXIT(as, &as->a_lock);
3661         return (E2BIG);
3662     }
3663
3664     buf = kmem_zalloc(size, KM_SLEEP);
3665     php = (prpageheader_t *)buf;
3666     pmp = (prasmmap_t *)(buf + sizeof (prpageheader_t));
3667
3668     hrt2ts(gethrtime(), &php->pr_tstamp);
3669     php->pr_nmap = 0;
3670     php->pr_npage = 0;
3671     do {
3672         caddr_t eaddr = seg->s_base + pr_getsegsize(seg, 0);
3673         caddr_t saddr, naddr;
3674         void *tmp = NULL;
3675
3676         for (saddr = seg->s_base; saddr < eaddr; saddr = naddr) {
3677             size_t len;
3678             size_t npage;
3679             uint_t prot;
3680             uintptr_t next;
3681
3682             prot = pr_getprot(seg, 0, &tmp, &saddr, &naddr, eaddr);
3683             if ((len = naddr - saddr) == 0)
3684                 continue;
3685             npage = len / PAGESIZE;
3686             next = (uintptr_t)(pmp + 1) + roundlong(npage);
3687
3688             /*
3689              * It's possible that the address space can change
3690              * subtly even though we're holding as->a_lock
3691              * due to the nondeterminism of page_exists() in
3692              * the presence of asynchronously flushed pages or
3693              * mapped files whose sizes are changing.
3694              * page_exists() may be called indirectly from
3695              * pr_getprot() by a segop_incore() routine.
3696              * pr_getprot() by a SEGOP_INCORE() routine.
3697              * If this happens we need to make sure we don't
3698              * overrun the buffer whose size we computed based
3699              * on the initial iteration through the segments.
3700              * 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);
3769
3770             kmem_free(buf, size);
3771
3772             if (ISSIG(curthread, JUSTLOOKING))
3773                 return (EINTR);
3774
3775             goto again;
3776         }
3777
3778         php->pr_nmap++;
3779         php->pr_npage += npage;
3780         php->pr_vaddr = saddr;
3781         pmp->pr_npage = npage;
3782         pmp->pr_off = segop_getoffset(seg, saddr);
3783         pmp->pr_off = SEGOP_GETOFFSET(seg, saddr);
3784         pmp->pr_mflags = 0;
3785         if (prot & PROT_READ)
3786             pmp->pr_mflags |= MA_READ;
3787         if (prot & PROT_WRITE)
3788             pmp->pr_mflags |= MA_WRITE;
3789         if (prot & PROT_EXEC)
3790             pmp->pr_mflags |= MA_EXEC;
3791         if (segop_gettime(seg, saddr) & MAP_SHARED)
3792             if (SEGOP_GETTYPE(seg, saddr) & MAP_SHARED)
3793                 pmp->pr_mflags |= MA_SHARED;
3794         pmp->pr_pagesize = PAGESIZE;
3795         hat_getstat(as, saddr, len, hatid,
3796                     (char *) (pmp + 1), HAT_SYNC_ZERORM);
3797         pmp = (prasmapping_t *) next;
3798     }
3799     ASSERT(tmp == NULL);
3800 } while ((seg = AS_SEGNEXT(as, seg)) != NULL);
3801
3802 AS_LOCK_EXIT(as, &as->a_lock);
3803
3804 ASSERT((uintptr_t)pmp <= (uintptr_t)buf + size);
3805 error = uiomove(buf, (caddr_t)pmp - buf, UIO_READ, uiop);
3806 kmem_free(buf, size);
3807
3808 return (error);
3809
3810 #ifdef _SYSCALL32_IMPL
3811 int
3812 oprpdread32(struct as *as, uint_t hatid, struct uio *uiop)
3813 {
3814     caddr_t buf;
3815     size_t size;
3816     ioc_prpageheader32_t *php;
3817     ioc_prasmapping32_t *pmp;
3818     struct seg *seg;
3819     int error;
3820
3821 again:
3822     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
3826         kmem_free(buf, size);
3827
3828         if (ISSIG(curthread, JUSTLOOKING))
3829             return (EINTR);
3830
3831         goto again;
3832
3833         php->pr_nmap++;
3834         php->pr_npage += npage;
3835         pmp->pr_vaddr = (uint32_t)(uintptr_t)saddr;
3836         pmp->pr_npage = (uint32_t)npage;
3837         pmp->pr_off = (int32_t)segop_getoffset(seg, saddr);
3838
3839         hrt2ts32(gethrtime(), &php->pr_tstamp);
3840         php->pr_nmap = 0;
3841         php->pr_npage = 0;
3842         do {
3843             caddr_t eaddr = seg->s_base + pr_getsegsize(seg, 0);
3844             caddr_t saddr, naddr;
3845             void *tmp = NULL;
3846
3847             for (saddr = seg->s_base; saddr < eaddr; saddr = naddr) {
3848                 size_t len;
3849                 size_t npage;
3850                 uint_t prot;
3851                 uintptr_t next;
3852
3853                 prot = pr_getprot(seg, 0, &tmp, &saddr, &naddr, eaddr);
3854                 if ((len = naddr - saddr) == 0)
3855                     continue;
3856                 npage = len / PAGESIZE;
3857                 next = (uintptr_t)(pmp + 1) + round4(npage);
3858
3859                 /*
3860                  * It's possible that the address space can change
3861                  * subtly even though we're holding as->a_lock
3862                  * due to the nondeterminism of page_exists() in
3863                  * the presence of asynchronously flushed pages or
3864                  * mapped files whose sizes are changing.
3865                  * page_exists() may be called indirectly from
3866                  * pr_getprot() by a segop_incore() routine.
3867                  * pr_getprot() by a SEGOP_INCORE() routine.
3868                  * If this happens we need to make sure we don't
3869                  * overrun the buffer whose size we computed based
3870                  * on the initial iteration through the segments.
3871                  * Once we've detected an overflow, we need to clean
3872                  * up the temporary memory allocated in pr_getprot()
3873                  * and retry. If there's a pending signal, we return
3874                  * EINTR so that this thread can be dislodged if
3875                  * a latent bug causes us to spin indefinitely.
3876
3877                 if (next > (uintptr_t)buf + size) {
3878                     pr_getprot_done(&tmp);
3879                     AS_LOCK_EXIT(as, &as->a_lock);
3880
3881                     kmem_free(buf, size);
3882
3883                     if (ISSIG(curthread, JUSTLOOKING))
3884                         return (EINTR);
3885
3886                     goto again;
3887
3888                     php->pr_nmap++;
3889                     php->pr_npage += npage;
3890                     pmp->pr_vaddr = (uint32_t)(uintptr_t)saddr;
3891                     pmp->pr_npage = (uint32_t)npage;
3892                     pmp->pr_off = (int32_t)segop_getoffset(seg, saddr);
3893
3894                     hrt2ts32(gethrtime(), &php->pr_tstamp);
3895                     php->pr_nmap = 0;
3896                     php->pr_npage = 0;
3897
3898                     do {
3899                         caddr_t eaddr = seg->s_base + pr_getsegsize(seg, 0);
3900                         caddr_t saddr, naddr;
3901                         void *tmp = NULL;
3902
3903                         for (saddr = seg->s_base; saddr < eaddr; saddr = naddr) {
3904                             size_t len;
3905                             size_t npage;
3906                             uint_t prot;
3907                             uintptr_t next;
3908
3909                             prot = pr_getprot(seg, 0, &tmp, &saddr, &naddr, eaddr);
3910                             if ((len = naddr - saddr) == 0)
3911                                 continue;
3912                             npage = len / PAGESIZE;
3913                             next = (uintptr_t)(pmp + 1) + round4(npage);
3914
3915                             /*
3916                                * It's possible that the address space can change
3917                                * subtly even though we're holding as->a_lock
3918                                * due to the nondeterminism of page_exists() in
3919                                * the presence of asynchronously flushed pages or
3920                                * mapped files whose sizes are changing.
3921                                * page_exists() may be called indirectly from
3922                                * pr_getprot() by a segop_incore() routine.
3923                                * pr_getprot() by a SEGOP_INCORE() routine.
3924                                * If this happens we need to make sure we don't
3925                                * overrun the buffer whose size we computed based
3926                                * on the initial iteration through the segments.
3927                                * Once we've detected an overflow, we need to clean
3928                                * up the temporary memory allocated in pr_getprot()
3929                                * and retry. If there's a pending signal, we return
3930                                * EINTR so that this thread can be dislodged if
3931                                * a latent bug causes us to spin indefinitely.
3932
3933                             if (next > (uintptr_t)buf + size) {
3934                                 pr_getprot_done(&tmp);
3935                                 AS_LOCK_EXIT(as, &as->a_lock);
3936
3937                                 kmem_free(buf, size);
3938
3939                                 if (ISSIG(curthread, JUSTLOOKING))
3940                                     return (EINTR);
3941
3942                                 goto again;
3943
3944                                 php->pr_nmap++;
3945                                 php->pr_npage += npage;
3946                                 pmp->pr_vaddr = (uint32_t)(uintptr_t)saddr;
3947                                 pmp->pr_npage = (uint32_t)npage;
3948                                 pmp->pr_off = (int32_t)segop_getoffset(seg, saddr);
3949
3950                                 hrt2ts32(gethrtime(), &php->pr_tstamp);
3951                                 php->pr_nmap = 0;
3952                                 php->pr_npage = 0;
3953
3954                                 do {
3955                                     caddr_t eaddr = seg->s_base + pr_getsegsize(seg, 0);
3956                                     caddr_t saddr, naddr;
3957                                     void *tmp = NULL;
3958
3959                                     for (saddr = seg->s_base; saddr < eaddr; saddr = naddr) {
3960                                         size_t len;
3961                                         size_t npage;
3962                                         uint_t prot;
3963                                         uintptr_t next;
3964
3965                                         prot = pr_getprot(seg, 0, &tmp, &saddr, &naddr, eaddr);
3966                                         if ((len = naddr - saddr) == 0)
3967                                             continue;
3968                                         npage = len / PAGESIZE;
3969                                         next = (uintptr_t)(pmp + 1) + round4(npage);
3970
3971                                         /*
3972                                            * It's possible that the address space can change
3973                                            * subtly even though we're holding as->a_lock
3974                                            * due to the nondeterminism of page_exists() in
3975                                            * the presence of asynchronously flushed pages or
3976                                            * mapped files whose sizes are changing.
3977                                            * page_exists() may be called indirectly from
3978                                            * pr_getprot() by a segop_incore() routine.
3979                                            * pr_getprot() by a SEGOP_INCORE() routine.
3980                                            * If this happens we need to make sure we don't
3981                                            * overrun the buffer whose size we computed based
3982                                            * on the initial iteration through the segments.
3983                                            * Once we've detected an overflow, we need to clean
3984                                            * up the temporary memory allocated in pr_getprot()
3985                                            * and retry. If there's a pending signal, we return
3986                                            * EINTR so that this thread can be dislodged if
3987                                            * a latent bug causes us to spin indefinitely.
3988
3989                                         if (next > (uintptr_t)buf + size) {
3990                                             pr_getprot_done(&tmp);
3991                                             AS_LOCK_EXIT(as, &as->a_lock);
3992
3993                                             kmem_free(buf, size);
3994
3995                                             if (ISSIG(curthread, JUSTLOOKING))
3996                                                 return (EINTR);
3997
3998                                             goto again;
3999
4000                                             php->pr_nmap++;
4001                                             php->pr_npage += npage;
4002                                             pmp->pr_vaddr = (uint32_t)(uintptr_t)saddr;
4003                                             pmp->pr_npage = (uint32_t)npage;
4004                                             pmp->pr_off = (int32_t)segop_getoffset(seg, saddr);
4005
4006                                             hrt2ts32(gethrtime(), &php->pr_tstamp);
4007                                             php->pr_nmap = 0;
4008                                             php->pr_npage = 0;
4009
4010                                             do {
4011                                                 caddr_t eaddr = seg->s_base + pr_getsegsize(seg, 0);
4012                                                 caddr_t saddr, naddr;
4013                                                 void *tmp = NULL;
4014
4015                                                 for (saddr = seg->s_base; saddr < eaddr; saddr = naddr) {
4016                                                     size_t len;
4017                                                     size_t npage;
4018                                                     uint_t prot;
4019                                                     uintptr_t next;
4020
4021                                                     prot = pr_getprot(seg, 0, &tmp, &saddr, &naddr, eaddr);
4022                                                     if ((len = naddr - saddr) == 0)
4023                                                         continue;
4024                                                     npage = len / PAGESIZE;
4025                                                     next = (uintptr_t)(pmp + 1) + round4(npage);
4026
4027                                                     /*
4028                                                        * It's possible that the address space can change
4029                                                        * subtly even though we're holding as->a_lock
4030                                                        * due to the nondeterminism of page_exists() in
4031                                                        * the presence of asynchronously flushed pages or
4032                                                        * mapped files whose sizes are changing.
4033                                                        * page_exists() may be called indirectly from
4034                                                        * pr_getprot() by a segop_incore() routine.
4035                                                        * pr_getprot() by a SEGOP_INCORE() routine.
4036                                                        * If this happens we need to make sure we don't
4037                                                        * overrun the buffer whose size we computed based
4038                                                        * on the initial iteration through the segments.
4039                                                        * Once we've detected an overflow, we need to clean
4040                                                        * up the temporary memory allocated in pr_getprot()
4041                                                        * and retry. If there's a pending signal, we return
4042                                                        * EINTR so that this thread can be dislodged if
4043                                                        * a latent bug causes us to spin indefinitely.
4044
4045                                                     if (next > (uintptr_t)buf + size) {
4046                                                         pr_getprot_done(&tmp);
4047                                                         AS_LOCK_EXIT(as, &as->a_lock);
4048
4049                                                         kmem_free(buf, size);
4050
4051                                                         if (ISSIG(curthread, JUSTLOOKING))
4052                                                             return (EINTR);
4053
4054                                                         goto again;
4055
4056                                                         php->pr_nmap++;
4057                                                         php->pr_npage += npage;
4058                                                         pmp->pr_vaddr = (uint32_t)(uintptr_t)saddr;
4059                                                         pmp->pr_npage = (uint32_t)npage;
4060                                                         pmp->pr_off = (int32_t)segop_getoffset(seg, saddr);
4061
4062                                                         hrt2ts32(gethrtime(), &php->pr_tstamp);
4063                                                         php->pr_nmap = 0;
4064                                                         php->pr_npage = 0;
4065
4066                                                         do {
4067                                                             caddr_t eaddr = seg->s_base + pr_getsegsize(seg, 0);
4068                                                             caddr_t saddr, naddr;
4069                                                             void *tmp = NULL;
4070
4071                                                             for (saddr = seg->s_base; saddr < eaddr; saddr = naddr) {
4072                                                                 size_t len;
4073                                                                 size_t npage;
4074                                                                 uint_t prot;
4075                                                                 uintptr_t next;
4076
4077                                                                 prot = pr_getprot(seg, 0, &tmp, &saddr, &naddr, eaddr);
4078                                                                 if ((len = naddr - saddr) == 0)
4079                                                                     continue;
4080                                                                 npage = len / PAGESIZE;
4081                                                                 next = (uintptr_t)(pmp + 1) + round4(npage);
4082
4083                                                                 /*
4084                                                                    * It's possible that the address space can change
4085                                                                    * subtly even though we're holding as->a_lock
4086                                                                    * due to the nondeterminism of page_exists() in
4087                                                                    * the presence of asynchronously flushed pages or
4088                                                                    * mapped files whose sizes are changing.
4089                                                                    * page_exists() may be called indirectly from
4090                                                                    * pr_getprot() by a segop_incore() routine.
4091                                                                    * pr_getprot() by a SEGOP_INCORE() routine.
4092                                                                    * If this happens we need to make sure we don't
4093                                                                    * overrun the buffer whose size we computed based
4094                                                                    * on the initial iteration through the segments.
4095                                                                    * Once we've detected an overflow, we need to clean
4096                                                                    * up the temporary memory allocated in pr_getprot()
4097                                                                    * and retry. If there's a pending signal, we return
4098                                                                    * EINTR so that this thread can be dislodged if
4099                                                                    * a latent bug causes us to spin indefinitely.
4100
4101                                                                 if (next > (uintptr_t)buf + size) {
4102                                                                     pr_getprot_done(&tmp);
4103                                                                     AS_LOCK_EXIT(as, &as->a_lock);
4104
4105                                                                     kmem_free(buf, size);
4106
4107                                                                     if (ISSIG(curthread, JUSTLOOKING))
4108                                                                         return (EINTR);
4109
4110                                                                     goto again;
4111
4112                                                                     php->pr_nmap++;
4113                                                                     php->pr_npage += npage;
4114                                                                     pmp->pr_vaddr = (uint32_t)(uintptr_t)saddr;
4115                                                                     pmp->pr_npage = (uint32_t)npage;
4116                                                                     pmp->pr_off = (int32_t)segop_getoffset(seg, saddr);
4117
4118                                                                     hrt2ts32(gethrtime(), &php->pr_tstamp);
4119                                                                     php->pr_nmap = 0;
4120                                                                     php->pr_npage = 0;
4121
4122                                                                     do {
4123                                                                         caddr_t eaddr = seg->s_base + pr_getsegsize(seg, 0);
4124                                                                         caddr_t saddr, naddr;
4125                                                                         void *tmp = NULL;
4126
4127                                                                         for (saddr = seg->s_base; saddr < eaddr; saddr = naddr) {
4128                                                                             size_t len;
4129                                                                             size_t npage;
4130                                                                             uint_t prot;
4131                                                                             uintptr_t next;
4132
4133                                                                             prot = pr_getprot(seg, 0, &tmp, &saddr, &naddr, eaddr);
4134                                                                             if ((len = naddr - saddr) == 0)
4135                                                                                 continue;
4136                                                                             npage = len / PAGESIZE;
4137                                                                             next = (uintptr_t)(pmp + 1) + round4(npage);
4138
4139                                                                             /*
4140                                                                                * It's possible that the address space can change
4141                                                                                * subtly even though we're holding as->a_lock
4142                                                                                * due to the nondeterminism of page_exists() in
4143                                                                                * the presence of asynchronously flushed pages or
4144                                                                                * mapped files whose sizes are changing.
4145                                                                                * page_exists() may be called indirectly from
4146                                                                                * pr_getprot() by a segop_incore() routine.
4147                                                                                * pr_getprot() by a SEGOP_INCORE() routine.
4148                                                                                * If this happens we need to make sure we don't
4149                                                                                * overrun the buffer whose size we computed based
4150                                                                                * on the initial iteration through the segments.
4151                                                                                * Once we've detected an overflow, we need to clean
4152                                                                                * up the temporary memory allocated in pr_getprot()
4153                                                                                * and retry. If there's a pending signal, we return
4154                                                                                * EINTR so that this thread can be dislodged if
4155                                                                                * a latent bug causes us to spin indefinitely.
4156
4157                                                                             if (next > (uintptr_t)buf + size) {
4158                                                                                 pr_getprot_done(&tmp);
4159                                                                                 AS_LOCK_EXIT(as, &as->a_lock);
4160
4161                                                                                 kmem_free(buf, size);
4162
4163                                                                                 if (ISSIG(curthread, JUSTLOOKING))
4164                                                                 return (EINTR);
4165
4166                                                                                 goto again;
4167
4168                                                                                 php->pr_nmap++;
4169                                                                                 php->pr_npage += npage;
4170                                                                                 pmp->pr_vaddr = (uint32_t)(uintptr_t)saddr;
4171                                                                                 pmp->pr_npage = (uint32_t)npage;
4172                                                                                 pmp->pr_off = (int32_t)segop_getoffset(seg, saddr);
4173
4174                                                                                 hrt2ts32(gethrtime(), &php->pr_tstamp);
4175                                                                                 php->pr_nmap = 0;
4176                                                                                 php->pr_npage = 0;
4177
4178                                                                                 do {
4179                                                                 caddr_t eaddr = seg->s_base + pr_getsegsize(seg, 0);
4180                                                                 caddr_t saddr, naddr;
4181                                                                 void *tmp = NULL;
4182
4183                                                                 for (saddr = seg->s_base; saddr < eaddr; saddr = naddr) {
4184                                                                     size_t len;
4185                                                                     size_t npage;
4186                                                                     uint_t prot;
4187                                                                     uintptr_t next;
4188
4189                                                                     prot = pr_getprot(seg, 0, &tmp, &saddr, &naddr, eaddr);
4190                                                                     if ((len = naddr - saddr) == 0)
4191                                                                         continue;
4192                                                                     npage = len / PAGESIZE;
4193                                                                     next = (uintptr_t)(pmp + 1) + round4(npage);
4194
4195                                                                     /*
4196                                                                        * It's possible that the address space can change
4197                                                                        * subtly even though we're holding as->a_lock
4198                                                                        * due to the nondeterminism of page_exists() in
4199                                                                        * the presence of asynchronously flushed pages or
4200                                                                        * mapped files whose sizes are changing.
4201                                                                        * page_exists() may be called indirectly from
4202                                                                        * pr_getprot() by a segop_incore() routine.
4203                                                                        * pr_getprot() by a SEGOP_INCORE() routine.
4204                                                                        * If this happens we need to make sure we don't
4205                                                                        * overrun the buffer whose size we computed based
4206                                                                        * on the initial iteration through the segments.
4207                                                                        * Once we've detected an overflow, we need to clean
4208                                                                        * up the temporary memory allocated in pr_getprot()
4209                                                                        * and retry. If there's a pending signal, we return
4210                                                                        * EINTR so that this thread can be dislodged if
4211                                                                        * a latent bug causes us to spin indefinitely.
4212
4213                                                                     if (next > (uintptr_t)buf + size) {
4214                                                                         pr_getprot_done(&tmp);
4215                                                                         AS_LOCK_EXIT(as, &as->a_lock);
4216
4217                                                                         kmem_free(buf, size);
4218
4219                                                                         if (ISSIG(curthread, JUSTLOOKING))
4220                                                               return (EINTR);
4221
4222                                                                         goto again;
4223
4224                                                                         php->pr_nmap++;
4225                                                                         php->pr_npage += npage;
4226                                                                         pmp->pr_vaddr = (uint32_t)(uintptr_t)saddr;
4227                                                                         pmp->pr_npage = (uint32_t)npage;
4228                                                                         pmp->pr_off = (int32_t)segop_getoffset(seg, saddr);
4229
4230                                                                         hrt2ts32(gethrtime(), &php->pr_tstamp);
4231                                                                         php->pr_nmap = 0;
4232                                                                         php->pr_npage = 0;
4233
4234                                                                         do {
4235                                                                             caddr_t eaddr = seg->s_base + pr_getsegsize(seg, 0);
4236                                                                             caddr_t saddr, naddr;
4237                                                                             void *tmp = NULL;
4238
4239                                                                             for (saddr = seg->s_base; saddr < eaddr; saddr = naddr) {
4240                                                                                 size_t len;
4241                                                                                 size_t npage;
4242                                                                                 uint_t prot;
4243                                                                                 uintptr_t next;
4244
4245                                                                                 prot = pr_getprot(seg, 0, &tmp, &saddr, &naddr, eaddr);
4246                                                                                 if ((len = naddr - saddr) == 0)
4247                                                                 continue;
4248                                                                                 npage = len / PAGESIZE;
4249                                                                                 next = (uintptr_t)(pmp + 1) + round4(npage);
4250
4251                                                                                 /*
4252                                                                                    * It's possible that the address space can change
4253                                                                                    * subtly even though we're holding as->a_lock
4254                                                                                    * due to the nondeterminism of page_exists() in
4255                                                                                    * the presence of asynchronously flushed pages or
4256                                                                                    * mapped files whose sizes are changing.
4257                                                                                    * page_exists() may be called indirectly from
4258                                                                                    * pr_getprot() by a segop_incore() routine.
4259                                                                                    * pr_getprot() by a SEGOP_INCORE() routine.
4260                                                                                    * If this happens we need to make sure we don't
4261                                                                                    * overrun the buffer whose size we computed based
4262                                                                                    * on the initial iteration through the segments.
4263                                                                                    * Once we've detected an overflow, we need to clean
4264                                                                                    * up the temporary memory allocated in pr_getprot()
4265                                                                                    * and retry. If there's a pending signal, we return
4266                                                                                    * EINTR so that this thread can be dislodged if
4267                                                                                    * a latent bug causes us to spin indefinitely.
4268
4269                                                                                 if (next > (uintptr_t)buf + size) {
4270                                                                 pr_getprot_done(&tmp);
4271                                                                 AS_LOCK_EXIT(as, &as->a_lock);
4272
4273                                                                 kmem_free(buf, size);
4274
4275                                                                 if (ISSIG(curthread, JUSTLOOKING))
4276                                                                     return (EINTR);
4277
4278                                                                 goto again;
4279
4280                                                                 php->pr_nmap++;
4281                                                                 php->pr_npage += npage;
4282                                                                 pmp->pr_vaddr = (uint32_t)(uintptr_t)saddr;
4283                                                                 pmp->pr_npage = (uint32_t)npage;
4284                                                                 pmp->pr_off = (int32_t)segop_getoffset(seg, saddr);
4285
4286                                                                 hrt2ts32(gethrtime(), &php->pr_tstamp);
4287                                                                 php->pr_nmap = 0;
4288                                                                 php->pr_npage = 0;
4289
4290                                                                 do {
4291                                                                     caddr_t eaddr = seg->s_base + pr_getsegsize(seg, 0);
4292                                                                     caddr_t saddr, naddr;
4293                                                                     void *tmp = NULL;
4294
4295                                                                     for (saddr = seg->s_base; saddr < eaddr; saddr = naddr) {
4296                                                                         size_t len;
4297                                                                         size_t npage;
4298                                                                         uint_t prot;
4299                                                                         uintptr_t next;
4300
4301                                                                         prot = pr_getprot(seg, 0, &tmp, &saddr, &naddr, eaddr);
4302                                                                         if ((len = naddr - saddr) == 0)
4303                                                                             continue;
4304                                                                         npage = len / PAGESIZE;
4305                                                                         next = (uintptr_t)(pmp + 1) + round4(npage);
4306
4307                                                                         /*
4308                                                                            * It's possible that the address space can change
4309                                                                            * subtly even though we're holding as->a_lock
4310                                                                            * due to the nondeterminism of page_exists() in
4311                                                                            * the presence of asynchronously flushed pages or
4312                                                                            * mapped files whose sizes are changing.
4313                                                                            * page_exists() may be called indirectly from
4314                                                                            * pr_getprot() by a segop_incore() routine.
4315                                                                            * pr_getprot() by a SEGOP_INCORE() routine.
4316                                                                            * If this happens we need to make sure we don't
4317                                                                            * overrun the buffer whose size we computed based
4318                                                                            * on the initial iteration through the segments.
4319                                                                            * Once we've detected an overflow, we need to clean
4320                                                                            * up the temporary memory allocated in pr_getprot()
4321                                                                            * and retry. If there's a pending signal, we return
4322                                                                            * EINTR so that this thread can be dislodged if
4323                                                                            * a latent bug causes us to spin indefinitely.
4324
4325                                                                           if (next > (uintptr_t)buf + size) {
4326                                                                               pr_getprot_done(&tmp);
4327                                                                               AS_LOCK_EXIT(as, &as->a_lock);
4328
4329                                                                               kmem_free(buf, size);
4330
4331                                                                               if (ISSIG(curthread, JUSTLOOKING))
4332                                                                 return (EINTR);
4333
4334                                                                               goto again;
4335
4336                                                                               php->pr_nmap++;
4337                                                                               php->pr_npage += npage;
4338                                                                               pmp->pr_vaddr = (uint32_t)(uintptr_t)saddr;
4339                                                                               pmp->pr_npage = (uint32_t)npage;
4340                                                                               pmp->pr_off = (int32_t)segop_getoffset(seg, saddr);
4341
4342                                                                               hrt2ts32(gethrtime(), &php->pr_tstamp);
4343                                                                               php->pr_nmap = 0;
4344                                                                               php->pr_npage = 0;
4345
4346                                                                               do {
4347                                                                 caddr_t eaddr = seg->s_base + pr_getsegsize(seg, 0);
4348                                                                 caddr_t saddr, naddr;
4349                                                                 void *tmp = NULL;
4350
4351                                                                 for (saddr = seg->s_base; saddr < eaddr; saddr = naddr) {
4352                                                                     size_t len;
4353                                                                     size_t npage;
4354                                                                     uint_t prot;
4355                                                                     uintptr_t next;
4356
4357                                                                     prot = pr_getprot(seg, 0, &tmp, &saddr, &naddr, eaddr);
4358                                                                     if ((len = naddr - saddr) == 0)
4359                                                                         continue;
4360                                                                     npage = len / PAGESIZE;
4361                                                                     next = (uintptr_t)(pmp + 1) + round4(npage);
4362
4363                                                                     /*
4364                                                                        * It's possible that the address space can change
4365                                                                        * subtly even though we're holding as->a_lock
4366                                                                        * due to the nondeterminism of page_exists() in
4367                                                                        * the presence of asynchronously flushed pages or
4368                                                                        * mapped files whose sizes are changing.
4369                                                                        * page_exists() may be called indirectly from
4370                                                                        * pr_getprot() by a segop_incore() routine.
4371                                                                        * pr_getprot() by a SEGOP_INCORE() routine.
4372                                                                        * If this happens we need to make sure we don't
4373                                                                        * overrun the buffer whose size we computed based
4374                                                                        * on the initial iteration through the segments.
4375                                                                        * Once we've detected an overflow, we need to clean
4376                                                                        * up the temporary memory allocated in pr_getprot()
4377                                                                        * and retry. If there's a pending signal, we return
4378                                                                        * EINTR so that this thread can be dislodged if
4379                                                                        * a latent bug causes us to spin indefinitely.
4380
4381                                                                       if (next > (uintptr_t)buf + size) {
4382                                                                           pr_getprot_done(&tmp);
4383                                                                           AS_LOCK_EXIT(as, &as->a_lock);
4384
4385                                                                           kmem_free(buf, size);
4386
4387                                                                           if (ISSIG(curthread, JUSTLOOKING))
4388                                                               return (EINTR);
4389
4390                                                                           goto again;
4391
4392                                                                           php->pr_nmap++;
4393                                                                           php->pr_npage += npage;
4394                                                                           pmp->pr_vaddr = (uint32_t)(uintptr_t)saddr;
4395                                                                           pmp->pr_npage = (uint32_t)npage;
4396                                                                           pmp->pr_off = (int32_t)segop_getoffset(seg, saddr);
4397
4398                                                                           hrt2ts32(gethrtime(), &php->pr_tstamp);
4399                                                                           php->pr_nmap = 0;
4400                                                                           php->pr_npage = 0;
4401
4402                                                                           do {
4403                                                                             caddr_t eaddr = seg->s_base + pr_getsegsize(seg, 0);
4404                                                                             caddr_t saddr, naddr;
4405                                                                             void *tmp = NULL;
4406
4407                                                                             for (saddr = seg->s_base; saddr < eaddr; saddr = naddr) {
4408                                                                                 size_t len;
4409                                                                                 size_t npage;
4410                                                                                 uint_t prot;
4411                                                                                 uintptr_t next;
4412
4413                                                                                 prot = pr_getprot(seg, 0, &tmp, &saddr, &naddr, eaddr);
4414                                                                                 if ((len = naddr - saddr) == 0)
4415                                                                 continue;
4416                                                                                 npage = len / PAGESIZE;
4417                                                                                 next = (uintptr_t)(pmp + 1) + round4(npage);
4418
4419                                                                                 /*
4420                                                                                    * It's possible that the address space can change
4421                                                                                    * subtly even though we're holding as->a_lock
4422                                                                                    * due to the nondeterminism of page_exists() in
4423                                                                                    * the presence of asynchronously flushed pages or
4424                                                                                    * mapped files whose sizes are changing.
4425                                                                                    * page_exists() may be called indirectly from
4426                                                                                    * pr_getprot() by a segop_incore() routine.
4427                                                                                    * pr_getprot() by a SEGOP_INCORE() routine.
4428                                                                                    * If this happens we need to make sure we don't
4429                                                                                    * overrun the buffer whose size we computed based
4430                                                                                    * on the initial iteration through the segments.
4431                                                                                    * Once we've detected an overflow, we need to clean
4432                                                                                    * up the temporary memory allocated in pr_getprot()
4433                                                                                    * and retry. If there's a pending signal, we return
4434                                                                                    * EINTR so that this thread can be dislodged if
4435                                                                                    * a latent bug causes us to spin indefinitely.
4436
4437                                                                 if (next > (uintptr_t)buf + size) {
4438                                                                     pr_getprot_done(&tmp);
4439                                                                     AS_LOCK_EXIT(as, &as->a_lock);
4440
4441                                                                     kmem_free(buf, size);
4442
4443                                                                     if (ISSIG(curthread, JUSTLOOKING))
4444                                                                       return (EINTR);
4445
4446                                                                     goto again;
4447
4448                                                                     php->pr_nmap++;
4449                                                                     php->pr_npage += npage;
4450                                                                     pmp->pr_vaddr = (uint32_t)(uintptr_t)saddr;
4451                                                                     pmp->pr_npage = (uint32_t)npage;
4452                                                                     pmp->pr_off = (int32_t)segop_getoffset(seg, saddr);
4453
4454                                                                     hrt2ts32(gethrtime(), &php->pr_tstamp);
4455                                                                     php->pr_nmap = 0;
4456                                                                     php->pr_npage = 0;
4457
4458                                                                     do {
4459                                                                         caddr_t eaddr = seg->s_base + pr_getsegsize(seg, 0);
4460                                                                         caddr_t saddr, naddr;
4461                                                                         void *tmp = NULL;
4462
4463                                                                         for (saddr = seg->s_base; saddr < eaddr; saddr = naddr) {
4464                                                                             size_t len;
4465                                                                             size_t npage;
4466                                                                             uint_t prot;
4467                                                                             uintptr_t next;
4468
4469                                                                             prot = pr_getprot(seg, 0, &tmp, &saddr, &naddr, eaddr);
4470                                                                             if ((len = naddr - saddr) == 0)
4471                                                                                 continue;
4472                                                                             npage = len / PAGESIZE;
4473                                                                             next = (uintptr_t)(pmp + 1) + round4(npage);
4474
4475                                                                             /*
4476                                                                                * It's possible that the address space can change
4477                                                                                * subtly even though we're holding as->a_lock
4478                                                                                * due to the nondeterminism of page_exists() in
4479                                                                                * the presence of asynchronously flushed pages or
4480                                                                                * mapped files whose sizes are changing.
4481                                                                                * page_exists() may be called indirectly from
4482                                                                                * pr_getprot() by a segop_incore() routine.
4483                                                                                * pr_getprot() by a SEGOP_INCORE() routine.
4484                                                                                * If this happens we need to make sure we don't
4485                                                                                * overrun the buffer whose size we computed based
4486                                                                                * on the initial iteration through the segments.
4487                                                                                * Once we've detected an overflow, we need to clean
4488                                                                                * up the temporary memory allocated in pr_getprot()
4489                                                                                * and retry. If there's a pending signal, we return
4490                                                                                * EINTR so that this thread can be dislodged if
4491                                                                                * a latent bug causes us to spin indefinitely.
4492
4493                                                                           if (next > (uintptr_t)buf + size) {
4494                                                                               pr_getprot_done(&tmp);
4495                                                                               AS_LOCK_EXIT(as, &as->a_lock);
4496
4497                                                                               kmem_free(buf, size);
4498
4499                                                                               if (ISSIG(curthread, JUSTLOOKING))
4500                                                                 return (EINTR);
4501
4502                                                                               goto again;
4503
4504                                                                               php->pr_nmap++;
4505                                                                               php->pr_npage += npage;
4506                                                                               pmp->pr_vaddr = (uint32_t)(uintptr_t)saddr;
4507                                                                               pmp->pr_npage = (uint32_t)npage;
4508                                                                               pmp->pr_off = (int32_t)segop_getoffset(seg, saddr);
4509
4510                                                                               hrt2ts32(gethrtime(), &php->pr_tstamp);
4511                                                                               php->pr_nmap = 0;
4512                                                                               php->pr_npage = 0;
4513
4514                                                                               do {
4515                                                                 caddr_t eaddr = seg->s_base + pr_getsegsize(seg, 0);
4516                                                                 caddr_t saddr, naddr;
4517                                                                 void *tmp = NULL;
4518
4519                                                                 for (saddr = seg->s_base; saddr < eaddr; saddr = naddr) {
4520                                                                     size_t len;
4521                                                                     size_t npage;
4522                                                                     uint_t prot;
4523                                                                     uintptr_t next;
4524
4525                                                                     prot = pr_getprot(seg, 0, &tmp, &saddr, &naddr, eaddr);
4526                                                                     if ((len = naddr - saddr) == 0)
4527                                                                         continue;
4528                                                                     npage = len / PAGESIZE;
4529                                                                     next = (uintptr_t)(pmp + 1) + round4(npage);
4530
4531                                                                     /*
4532                                                                        * It's possible that the address space can change
4533                                                                        * subtly even though we're holding as->a_lock
4534                                                                        * due to the nondeterminism of page_exists() in
4535                                                                        * the presence of asynchronously flushed pages or
4536                                                                        * mapped files whose sizes are changing.
4537                                                                        * page_exists() may be called indirectly from
4538                                                                        * pr_getprot() by a segop_incore() routine.
4539                                                                        * pr_getprot() by a SEGOP_INCORE() routine.
4540                                                                        * If this happens we need to make sure we don't
4541                                                                        * overrun the buffer whose size we computed based
4542                                                                        * on the initial iteration through the segments.
4543                                                                        * Once we've detected an overflow, we need to clean
4544                                                                        * up the temporary memory allocated in pr_getprot()
4545                                                                        * and retry. If there's a pending signal, we return
4546                                                                        * EINTR so that this thread can be dislodged if
4547                                                                        * a latent bug causes us to spin indefinitely.
4548
```

```
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)
3898     if (SEGOP_GETTYPE(seg, saddr) & MAP_SHARED)
3899         pmp->pr_mflags |= MA_SHARED;
3900     pmp->pr_pagesize = PAGESIZE;
3901     hat_getstat(as, saddr, len, hatid,
3902                 (char*)(pmp + 1), HAT_SYNC_ZERORM);
3903     pmp = (ioc_prasmap32_t *)next;
3904 }
3905     ASSERT(tmp == NULL);
3906 } while ((seg = AS_SEGNEXT(as, seg)) != NULL);

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

3908 ASSERT((uintptr_t)pmp == (uintptr_t)buf + size);
3909 error = uiomove(buf, (caddr_t)pmp - buf, UIO_READ, uiop);
3910 kmem_free(buf, size);

3911 return (error);
3912 }
```

unchanged\_portion\_omitted\_

```
*****
112686 Tue Nov 24 09:34:45 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);
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)
1660         mp->pr_mflags |= MA_SHARED;
1661     if (segop_gettype(seg, saddr) & MAP_NORESERVE)
1662         mp->pr_mflags |= MA_NORESERVE;
1663     if (seg->s_ops == &segsvt_shmops ||
1664         (seg->s_ops == &segvn_ops &&
1665          (segop_getvp(seg, saddr, &vp) != 0 || vp == NULL)))
1666         (SEGOP_GETVP(seg, saddr, &vp) != 0 || vp == NULL))
1667         mp->pr_mflags |= MA_ANON;
1668     if (seg == brkseg)
1669         mp->pr_mflags |= MA_BREAK;
1670     else if (seg == stkseg) {
1671         mp->pr_mflags |= MA_STACK;
1672         if (reserved) {
1673             size_t maxstack =
1674                 ((size_t)p->p_stk_ctl +
1675                  PAGEOFFSET) & PAGEMASK;
1676             mp->pr_vaddr =
1677                 (uintptr_t)prgetstackbase(p) +
1678                 p->p_stksize - maxstack;
1679             mp->pr_size = (uintptr_t)naddr -
1680                         mp->pr_vaddr;
1681         }
1682     }
1683     if (seg->s_ops == &segsvt_shmops)
1684         mp->pr_mflags |= MA_ISM | MA_SHM;
1685     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 &&
1692         SEGOP_GETVP(seg, saddr, &vp) == 0 &&
1693         vp != NULL && vp->v_type == VREG &&
1694         VOP_GETATTR(vp, &vattr, 0, CRED(), NULL) == 0) {
1695             if (vp == p->p_exec)
1696                 (void) strcpy(mp->pr_mapname, "a.out");
1697             else
1698                 pr_object_name(mp->pr_mapname,
1699                                vp, &vattr);
1700         }
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 = shmidgetid(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 {           mp->pr_shmid = -1;
1712     }
1713     }
1714     ASSERT(tmp == NULL);
1715 } 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;
1732
1733     ASSERT(as != &kas && AS_WRITE_HELD(as, &as->a_lock));
1734
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));
1740
1741     if ((seg = AS_SEGFIRST(as)) == NULL)
1742         return (0);
1743
1744     brkseg = break_seg(p);
1745     stkseg = as_segat(as, prgetstackbase(p));
1746
1747     do {
1748         caddr_t eaddr = seg->s_base + pr_getsegsize(seg, reserved);
1749         caddr_t saddr, naddr;
1750         void *tmp = NULL;
1751
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;
1757
1758             mp = pr_iol_newbuf(iolhead, sizeof (*mp));
1759
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);
1763             mp->pr_offset = SEGOP_GETOFFSET(seg, saddr);
1764             mp->pr_mflags = 0;
1765             if (prot & PROT_READ)
1766                 mp->pr_mflags |= MA_READ;
1767             if (prot & PROT_WRITE)
1768                 mp->pr_mflags |= MA_WRITE;
1769             if (prot & PROT_EXEC)
1770                 mp->pr_mflags |= MA_EXEC;
1771             if (segop_gettime(seg, saddr) & MAP_SHARED)
1772                 mp->pr_mflags |= MA_SHARED;
1773             if (segop_gettime(seg, saddr) & MAP_NORESERVE)
1774                 mp->pr_mflags |= MA_NORESERVE;
1775         }
1776     }
1777     ASSERT(tmp == NULL);
1778 } while ((seg = AS_SEGNEXT(as, seg)) != NULL);

1779     return (0);
1780 }

1781 #endif /* _SYSCALL32_IMPL */

```

```

1772     if (SEGOP_GETTYPE(seg, saddr) & MAP_NORESERVE)
1773         mp->pr_mflags |= MA_NORESERVE;
1774     if (seg->s_ops == &segpt_shmops ||
1775         (seg->s_ops == &segvn_ops &&
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 == &segpt_shmops)
1795         mp->pr_mflags |= MA_ISM | MA_SHM;
1796     mp->pr_pagesize = PAGESIZE;

1797     /*
1798      * Manufacture a filename for the "object" directory.
1799      */
1800     vattr.va_mask = AT_FSID|AT_NODEID;
1801     if (seg->s_ops == &segvn_ops &&
1802         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     }

1812     /*
1813      * Get the SysV shared memory id, if any.
1814      */
1815     if ((mp->pr_mflags & MA_SHARED) && p->p_segacct &&
1816         (mp->pr_shmid = shmidgetid(p, seg->s_base)) !=
1817         SHMID_NONE) {
1818         if (mp->pr_shmid == SHMID_FREE)
1819             mp->pr_shmid = -1;
1820
1821         mp->pr_mflags |= MA_SHM;
1822     } else {
1823         mp->pr_shmid = -1;
1824     }
1825
1826     ASSERT(tmp == NULL);
1827 } while ((seg = AS_SEGNEXT(as, seg)) != NULL);

1828     return (0);
1829 }

1830 #endif /* _SYSCALL32_IMPL */
1831
1832 unchanged portion omitted
1833
1834 #endif /* _SYSCALL32_IMPL */
1835
1836 /*

```

```

1898 * Read page data information.
1899 */
1900 int
1901 prpdread(proc_t *p, uint_t hatid, struct uio *uiop)
1902 {
1903     struct as *as = p->p_as;
1904     caddr_t buf;
1905     size_t size;
1906     prpageheader_t *php;
1907     prasmap_t *pmp;
1908     struct seg *seg;
1909     int error;
1910
1911 again:
1912     AS_LOCK_ENTER(as, &as->a_lock, RW_WRITER);
1913
1914     if ((seg = AS_SEGFIRST(as)) == NULL) {
1915         AS_LOCK_EXIT(as, &as->a_lock);
1916         return (0);
1917     }
1918     size = prpdsize(as);
1919     if (uiop->uio_resid < size) {
1920         AS_LOCK_EXIT(as, &as->a_lock);
1921         return (E2BIG);
1922     }
1923
1924     buf = kmem_zalloc(size, KM_SLEEP);
1925     php = (prpageheader_t *)buf;
1926     pmp = (prasmap_t *)(buf + sizeof(prpageheader_t));
1927
1928     hrt2ts(gethrtime(), &php->pr_tstamp);
1929     php->pr_nmap = 0;
1930     php->pr_npage = 0;
1931     do {
1932         caddr_t eaddr = seg->s_base + pr_getsegsize(seg, 0);
1933         caddr_t saddr, naddr;
1934         void *tmp = NULL;
1935
1936         for (saddr = seg->s_base; saddr < eaddr; saddr = naddr) {
1937             struct vnode *vp;
1938             struct vattr vattr;
1939             size_t len;
1940             size_t npage;
1941             uint_t prot;
1942             uintptr_t next;
1943
1944             prot = pr_getprot(seg, 0, &tmp, &saddr, &naddr, eaddr);
1945             if ((len = (size_t)(naddr - saddr)) == 0)
1946                 continue;
1947             npage = len / PAGESIZE;
1948             next = (uintptr_t)(pmp + 1) + round8(npage);
1949             /*
1950             * It's possible that the address space can change
1951             * subtly even though we're holding as->a_lock
1952             * due to the nondeterminism of page_exists() in
1953             * the presence of asynchronously flushed pages or
1954             * mapped files whose sizes are changing.
1955             * page_exists() may be called indirectly from
1956             * pr_getprot() by a SEGOP_INCORE routine.
1957             * pr_getprot() by a SEGOP_INCORE() routine.
1958             * If this happens we need to make sure we don't
1959             * overrun the buffer whose size we computed based
1960             * on the initial iteration through the segments.
1961             * Once we've detected an overflow, we need to clean
1962             * up the temporary memory allocated in pr_getprot()
1963             * and retry. If there's a pending signal, we return

```

```

1963 * EINTR so that this thread can be dislodged if
1964 * a latent bug causes us to spin indefinitely.
1965 */
1966 if (next > (uintptr_t)buf + size) {
1967     pr_getprot_done(&tmp);
1968     AS_LOCK_EXIT(as, &as->a_lock);
1969
1970     kmem_free(buf, size);
1971
1972     if (ISSIG(curthread, JUSTLOOKING))
1973         return (EINTR);
1974
1975     goto again;
1976 }
1977
1978 php->pr_nmap++;
1979 php->pr_npage += npage;
1980 pmp->pr_vaddr = (uintptr_t)saddr;
1981 pmp->pr_npage = npage;
1982 pmp->pr_offset = segop_getoffset(seg, saddr);
1983 pmp->pr_offset = SEGOP_GETOFFSET(seg, saddr);
1984 pmp->pr_mflags = 0;
1985 if (prot & PROT_READ)
1986     pmp->pr_mflags |= MA_READ;
1987 if (prot & PROT_WRITE)
1988     pmp->pr_mflags |= MA_WRITE;
1989 if (prot & PROT_EXEC)
1990     pmp->pr_mflags |= MA_EXEC;
1991 if (segop_gettype(seg, saddr) & MAP_SHARED)
1992     pmp->pr_mflags |= MA_SHARED;
1993 if (segop_gettype(seg, saddr) & MAP_NORESERVE)
1994     pmp->pr_mflags |= MA_NORESERVE;
1995 if (seg->s_ops == &segsp_shmops ||
1996     (segop_getvp(seg, saddr, &vp) != 0 || vp == NULL)))
1997     pmp->pr_mflags |= MA_ANON;
1998 if (seg->s_ops == &segsp_shmops)
1999     pmp->pr_mflags |= MA_ISM | MA_SHM;
2000 pmp->pr_pagesize = PAGESIZE;
2001 /*
2002 * Manufacture a filename for the "object" directory.
2003 */
2004 vattr.va_mask = AT_FSID|AT_NODEID;
2005 if (seg->s_ops == &segvn_ops &&
2006     segop_getvp(seg, saddr, &vp) == 0 &&
2007     SEGOP_GETVP(seg, saddr, &vp) == 0 &&
2008     vp != NULL && vp->v_type == VREG &&
2009     VOP_GETATTR(vp, &vattr, 0, CRED(), NULL) == 0) {
2010     if (vp == p->p_exec)
2011         (void) strcpy(pmp->pr_mapname, "a.out");
2012     else
2013         pr_object_name(pmp->pr_mapname,
2014                         vp, &vattr);
2015 }
2016 /*
2017 * Get the SysV shared memory id, if any.
2018 */
2019 if ((pmp->pr_mflags & MA_SHARED) && p->p_segacct &&
2020     (pmp->pr_shmid = shmgetid(p, seg->s_base)) !=
2021     SHMID_NONE) {
2022     if (pmp->pr_shmid == SHMID_FREE)
2023         pmp->pr_shmid = -1;

```

```

2025             pmp->pr_mflags |= MA_SHM;
2026         } else {
2027             pmp->pr_shmid = -1;
2028         }
2029
2030         hat_getstat(as, saddr, len, hatid,
2031                     (char *)(pmp + 1), HAT_SYNC_ZERORM);
2032         pmp = (prasmmap_t *)next;
2033     }
2034     ASSERT(tmp == NULL);
2035   } while ((seg = AS_SEGNEXT(as, seg)) != NULL);
2036
2037   AS_LOCK_EXIT(as, &as->a_lock);
2038
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);
2042
2043   return (error);
2044 }
2045
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     prasmmap32_t *pmp;
2055     struct seg *seg;
2056     int error;
2057
2058 again:
2059     AS_LOCK_ENTER(as, &as->a_lock, RW_WRITER);
2060
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     }
2070
2071     buf = kmalloc(size, KM_SLEEP);
2072     php = (prpageheader32_t *)buf;
2073     pmp = (prasmmap32_t *)buf + sizeof(prpageheader32_t);
2074
2075     hrt2ts32(gethrttime(), &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;
2082
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;

```

```

2090
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     /*
2098      * It's possible that the address space can change
2099      * subtly even though we're holding as->a_lock
2100      * due to the nondeterminism of page_exists() in
2101      * the presence of asynchronously flushed pages or
2102      * mapped files whose sizes are changing.
2103      * page_exists() may be called indirectly from
2104      * pr_getprot() by a segop_incore() routine.
2105      * pr_getprot() by a SEGOP_INCORE() routine.
2106      * If this happens we need to make sure we don't
2107      * overrun the buffer whose size we computed based
2108      * on the initial iteration through the segments.
2109      * Once we've detected an overflow, we need to clean
2110      * up the temporary memory allocated in pr_getprot()
2111      * and retry. If there's a pending signal, we return
2112      * EINTR so that this thread can be dislodged if
2113      * a latent bug causes us to spin indefinitely.
2114      */
2115     if (next > (uintptr_t)buf + size) {
2116         pr_getprot_done(&tmp);
2117         AS_LOCK_EXIT(as, &as->a_lock);
2118
2119         kmem_free(buf, size);
2120
2121         if (ISSIG(curthread, JUSTLOOKING))
2122             return (EINTR);
2123
2124         goto again;
2125     }
2126
2127     pmp->pr_vaddr = (caddr32_t)(uintptr_t)saddr;
2128     pmp->pr_npage = (size32_t)npage;
2129     pmp->pr_offset = segop_getoffset(seg, saddr);
2130     pmp->pr_offset = SEGOP_GETOFFSET(seg, saddr);
2131     pmp->pr_mflags = 0;
2132     if (prot & PROT_READ)
2133         pmp->pr_mflags |= MA_READ;
2134     if (prot & PROT_WRITE)
2135         pmp->pr_mflags |= MA_WRITE;
2136     if (prot & PROT_EXEC)
2137         pmp->pr_mflags |= MA_EXEC;
2138     if (segop_gettype(seg, saddr) & MAP_SHARED)
2139         pmp->pr_mflags |= MA_SHARED;
2140     if (segop_gettype(seg, saddr) & MAP_NORESERVE)
2141         pmp->pr_mflags |= MA_NORESERVE;
2142     if (seg->s_ops == &segpt_shmops ||
2143         (seg->s_ops == &segvn_ops &&
2144          (segop_getvp(seg, saddr, &vp) != 0 || vp == NULL)))
2145         pmp->pr_mflags |= MA_ANON;
2146     if (seg->s_ops == &segpt_shmops)
2147         pmp->pr_mflags |= MA_ISM | MA_SHM;
2148     pmp->pr_pagesize = PAGESIZE;
2149
2150     /*
2151      * Manufacture a filename for the "object" directory.
2152      */

```

```

2151     vattr.va_mask = AT_FSID|AT_NODEID;
2152     if (seg->s_ops == &segvn_ops &&
2153         segop_getvp(seg, saddr, &vp) == 0 &&
2154         SEGOP_GETVP(seg, saddr, &vp) == 0 &&
2155         vp != NULL && vp->v_type == VREG &&
2156         VOP_GETATTR(vp, &vattr, 0, CRED(), NULL) == 0) {
2157             if (vp == p->p_exec)
2158                 (void) strcpy(pmp->pr_mapname, "a.out");
2159             else
2160                 pr_object_name(pmp->pr_mapname,
2161                               vp, &vattr);
2162         }
2163
2164         /*
2165          * Get the SysV shared memory id, if any.
2166          */
2167         if ((pmp->pr_mflags & MA_SHARED) && p->p_segacct &&
2168             (pmp->pr_shmid = shmgetid(p, seg->s_base)) !=
2169             SHMID_NONE) {
2170             if (pmp->pr_shmid == SHMID_FREE)
2171                 pmp->pr_shmid = -1;
2172
2173             pmp->pr_mflags |= MA_SHM;
2174         } else {
2175             pmp->pr_shmid = -1;
2176         }
2177
2178         hat_getstat(as, saddr, len, hatid,
2179                     (char *)(pmp + 1), HAT_SYNC_ZERORM);
2180     }
2181     ASSERT(tmp == NULL);
2182 } while ((seg = AS_SEGNEXT(as, seg)) != NULL);
2183
2184 AS_LOCK_EXIT(as, &as->a_lock);
2185
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);
2189
2190 return (error);
2191 }



---



unchanged_portion_omitted


299 /*
300  * This one is called by the traced process to unwatch all the
301  * pages while deallocating the list of watched_page structs.
302  */
303 void
304 pr_free_watched_pages(proc_t *p)
305 {
306     struct as *as = p->p_as;
307     struct watched_page *pwp;
308     uint_t prot;
309     int retrycnt, err;
310     void *cookie;
311
312     if (as == NULL || avl_numnodes(&as->a_wpage) == 0)
313         return;
314
315     ASSERT(MUTEX_NOT_HELD(&curproc->p_lock));
316     AS_LOCK_ENTER(as, &as->a_lock, RW_WRITER);
317
318     pwp = avl_first(&as->a_wpage);
319
320     cookie = NULL;

```

```

3321     while ((pwp = avl_destroy_nodes(&as->a_wpage, &cookie)) != NULL) {
3322         retrycnt = 0;
3323         if ((prot = pwp->wp_oplevel) != 0) {
3324             caddr_t addr = pwp->wp_vaddr;
3325             struct seg *seg;
3326             retry:
3327
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);
3332                 err = SEGOP_SETPROT(seg, addr, PAGESIZE, prot);
3333                 if (err == IE_RETRY) {
3334                     ASSERT(retrycnt == 0);
3335                     retrycnt++;
3336                     goto retry;
3337                 }
3338             }
3339             kmem_free(pwp, sizeof (struct watched_page));
3340         }
3341
3342         avl_destroy(&as->a_wpage);
3343         p->p_wprot = NULL;
3344
3345         AS_LOCK_EXIT(as, &as->a_lock);
3346     }
3347
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;
3365
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         }
3378
3379         AS_LOCK_ENTER(as, &as->a_lock, RW_WRITER);
3380
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     }

```

```

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             (void) segop_getprot(seg, vaddr, 0, &prot);
3438             SEGOP_GETPROT(seg, vaddr, 0, &prot);
3439             pwp->wp_oprot = (uchar_t)prot;
3440             pwp->wp_prot = (uchar_t)prot;
3441         }
3442         if (pwp->wp_oprot != 0) {
3443             prot = pwp->wp_oprot;
3444             if (pwp->wp_read)
3445                 prot &= ~(PROT_READ|PROT_WRITE|PROT_EXEC);
3446             if (pwp->wp_write)
3447                 prot &= ~PROT_WRITE;
3448             if (pwp->wp_exec)
3449                 prot &= ~(PROT_READ|PROT_WRITE|PROT_EXEC);
3450             if (!!(pwp->wp_flags & WP_NOWATCH) &&
3451                 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     }
3457     pwp->wp_prot = (uchar_t)prot;
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;
3466 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_pnbbase = seg_page(seg, addr);
3630     pnlim = pagev->pg_pnbbase + pagev->pg_npaged;
3631     saddr = addr;
3633     if (lastpg < pnlim)
3634         len = (size_t)(eaddr - addr);
3635     else
3636         len = pagev->pg_npaged * 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         (void) segop_incore(seg, addr, len, pagev->pg_incore);
3644         SEGOP_INCORE(seg, addr, len, pagev->pg_incore);
3645         pn = pagev->pg_pnbbase;
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;
3655
3656     } while ((addr += PAGESIZE) < eaddr && pn < pnlim);
3657
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 }
3665
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     (void) segop_getprot(seg, saddr, len - 1, pagev->pg_protv);
3672     SEGOP_GETPROT(seg, saddr, len - 1, pagev->pg_protv);
3673     return (addr);
3674 }
```

unchanged\_portion\_omitted

```

3675 size_t
3676 pr_getsegsize(struct seg *seg, int reserved)
3677 {
3678     size_t size = seg->s_size;
3679
3680     /*
3681      * If we're interested in the reserved space, return the size of the
3682      * segment itself. Everything else in this function is a special case
3683      * to determine the actual underlying size of various segment types.
3684      */
3685     if (reserved)
3686         return (size);
3687
3688     /*
3689      * If this is a segvn mapping of a regular file, return the smaller
3690      * of the segment size and the remaining size of the file beyond
3691      * the file offset corresponding to seg->s_base.
3692      */
3693     if (seg->s_ops == &segvn_ops) {
3694         vattr_t vattr;
3695         vnode_t *vp;
3696
3697         vattr.va_mask = AT_SIZE;
3698
3699         if (segop_getvp(seg, seg->s_base, &vp) == 0 &&
3700             if (SEGOP_GETVP(seg, seg->s_base, &vp) == 0 &&
3701                 vp != NULL && vp->v_type == VREG &&
3702                 VOP_GETATTR(vp, &vattr, 0, CRED(), NULL) == 0) {
3703
3704             u_offset_t fsize = vattr.va_size;
3705             u_offset_t offset = segop_getoffset(seg, seg->s_base);
3706             u_offset_t offset = SEGOP_GETOFFSET(seg, seg->s_base);
3707
3708             if (fsize < offset)
3709                 fsize = 0;
3710             else
3711                 fsize -= offset;
3712
3713             if (fsize < offset)
3714                 fsize = 0;
3715             else
3716                 fsize -= offset;
3717
3718             if (fsize < offset)
3719                 fsize = 0;
3720             else
3721                 fsize -= offset;
3722
3723             if (fsize < offset)
3724                 fsize = 0;
3725             else
3726                 fsize -= offset;
3727
3728             if (fsize < offset)
3729                 fsize = 0;
3730             else
3731                 fsize -= offset;
3732
3733             if (fsize < offset)
3734                 fsize = 0;
3735             else
3736                 fsize -= offset;
3737
3738             if (fsize < offset)
3739                 fsize = 0;
3740             else
3741                 fsize -= offset;
3742
3743             if (fsize < offset)
3744                 fsize = 0;
3745             else
3746                 fsize -= offset;
3747
3748             if (fsize < offset)
3749                 fsize = 0;
3750             else
3751                 fsize -= offset;
3752
3753             if (fsize < offset)
3754                 fsize = 0;
3755             else
3756                 fsize -= offset;
3757
3758             if (fsize < offset)
3759                 fsize = 0;
3760             else
3761                 fsize -= offset;
3762
3763             if (fsize < offset)
3764                 fsize = 0;
3765             else
3766                 fsize -= offset;
3767
3768             if (fsize < offset)
3769                 fsize = 0;
3770             else
3771                 fsize -= offset;
3772
3773             if (fsize < offset)
3774                 fsize = 0;
3775             else
3776                 fsize -= offset;
3777
3778             if (fsize < offset)
3779                 fsize = 0;
3780             else
3781                 fsize -= offset;
3782
3783             if (fsize < offset)
3784                 fsize = 0;
3785             else
3786                 fsize -= offset;
3787
3788             if (fsize < offset)
3789                 fsize = 0;
3790             else
3791                 fsize -= offset;
3792
3793             if (fsize < offset)
3794                 fsize = 0;
3795             else
3796                 fsize -= offset;
3797
3798             if (fsize < offset)
3799                 fsize = 0;
3800             else
3801                 fsize -= offset;
3802
3803             if (fsize < offset)
3804                 fsize = 0;
3805             else
3806                 fsize -= offset;
3807
3808             if (fsize < offset)
3809                 fsize = 0;
3810             else
3811                 fsize -= offset;
3812
3813             if (fsize < offset)
3814                 fsize = 0;
3815             else
3816                 fsize -= offset;
3817
3818             if (fsize < offset)
3819                 fsize = 0;
3820             else
3821                 fsize -= offset;
3822
3823             if (fsize < offset)
3824                 fsize = 0;
3825             else
3826                 fsize -= offset;
3827
3828             if (fsize < offset)
3829                 fsize = 0;
3830             else
3831                 fsize -= offset;
3832
3833             if (fsize < offset)
3834                 fsize = 0;
3835             else
3836                 fsize -= offset;
3837
3838             if (fsize < offset)
3839                 fsize = 0;
3840             else
3841                 fsize -= offset;
3842
3843             if (fsize < offset)
3844                 fsize = 0;
3845             else
3846                 fsize -= offset;
3847
3848             if (fsize < offset)
3849                 fsize = 0;
3850             else
3851                 fsize -= offset;
3852
3853             if (fsize < offset)
3854                 fsize = 0;
3855             else
3856                 fsize -= offset;
3857
3858             if (fsize < offset)
3859                 fsize = 0;
3860             else
3861                 fsize -= offset;
3862
3863             if (fsize < offset)
3864                 fsize = 0;
3865             else
3866                 fsize -= offset;
3867
3868             if (fsize < offset)
3869                 fsize = 0;
3870             else
3871                 fsize -= offset;
3872
3873             if (fsize < offset)
3874                 fsize = 0;
3875             else
3876                 fsize -= offset;
3877
3878             if (fsize < offset)
3879                 fsize = 0;
3880             else
3881                 fsize -= offset;
3882
3883             if (fsize < offset)
3884                 fsize = 0;
3885             else
3886                 fsize -= offset;
3887
3888             if (fsize < offset)
3889                 fsize = 0;
3890             else
3891                 fsize -= offset;
3892
3893             if (fsize < offset)
3894                 fsize = 0;
3895             else
3896                 fsize -= offset;
3897
3898             if (fsize < offset)
3899                 fsize = 0;
3900             else
3901                 fsize -= offset;
3902
3903             if (fsize < offset)
3904                 fsize = 0;
3905             else
3906                 fsize -= offset;
3907
3908             if (fsize < offset)
3909                 fsize = 0;
3910             else
3911                 fsize -= offset;
3912
3913             if (fsize < offset)
3914                 fsize = 0;
3915             else
3916                 fsize -= offset;
3917
3918             if (fsize < offset)
3919                 fsize = 0;
3920             else
3921                 fsize -= offset;
3922
3923             if (fsize < offset)
3924                 fsize = 0;
3925             else
3926                 fsize -= offset;
3927
3928             if (fsize < offset)
3929                 fsize = 0;
3930             else
3931                 fsize -= offset;
3932
3933             if (fsize < offset)
3934                 fsize = 0;
3935             else
3936                 fsize -= offset;
3937
3938             if (fsize < offset)
3939                 fsize = 0;
3940             else
3941                 fsize -= offset;
3942
3943             if (fsize < offset)
3944                 fsize = 0;
3945             else
3946                 fsize -= offset;
3947
3948             if (fsize < offset)
3949                 fsize = 0;
3950             else
3951                 fsize -= offset;
3952
3953             if (fsize < offset)
3954                 fsize = 0;
3955             else
3956                 fsize -= offset;
3957
3958             if (fsize < offset)
3959                 fsize = 0;
3960             else
3961                 fsize -= offset;
3962
3963             if (fsize < offset)
3964                 fsize = 0;
3965             else
3966                 fsize -= offset;
3967
3968             if (fsize < offset)
3969                 fsize = 0;
3970             else
3971                 fsize -= offset;
3972
3973             if (fsize < offset)
3974                 fsize = 0;
3975             else
3976                 fsize -= offset;
3977
3978             if (fsize < offset)
3979                 fsize = 0;
3980             else
3981                 fsize -= offset;
3982
3983             if (fsize < offset)
3984                 fsize = 0;
3985             else
3986                 fsize -= offset;
3987
3988             if (fsize < offset)
3989                 fsize = 0;
3990             else
3991                 fsize -= offset;
3992
3993             if (fsize < offset)
3994                 fsize = 0;
3995             else
3996                 fsize -= offset;
3997
3998             if (fsize < offset)
3999                 fsize = 0;
4000             else
4001                 fsize -= offset;
4002
4003             if (fsize < offset)
4004                 fsize = 0;
4005             else
4006                 fsize -= offset;
4007
4008             if (fsize < offset)
4009                 fsize = 0;
4010             else
4011                 fsize -= offset;
4012
4013             if (fsize < offset)
4014                 fsize = 0;
4015             else
4016                 fsize -= offset;
4017
4018             if (fsize < offset)
4019                 fsize = 0;
4020             else
4021                 fsize -= offset;
4022
4023             if (fsize < offset)
4024                 fsize = 0;
4025             else
4026                 fsize -= offset;
4027
4028             if (fsize < offset)
4029                 fsize = 0;
4030             else
4031                 fsize -= offset;
4032
4033             if (fsize < offset)
4034                 fsize = 0;
4035             else
4036                 fsize -= offset;
4037
4038             if (fsize < offset)
4039                 fsize = 0;
4040             else
4041                 fsize -= offset;
4042
4043             if (fsize < offset)
4044                 fsize = 0;
4045             else
4046                 fsize -= offset;
4047
4048             if (fsize < offset)
4049                 fsize = 0;
4050             else
4051                 fsize -= offset;
4052
4053             if (fsize < offset)
4054                 fsize = 0;
4055             else
4056                 fsize -= offset;
4057
4058             if (fsize < offset)
4059                 fsize = 0;
4060             else
4061                 fsize -= offset;
4062
4063             if (fsize < offset)
4064                 fsize = 0;
4065             else
4066                 fsize -= offset;
4067
4068             if (fsize < offset)
4069                 fsize = 0;
4070             else
4071                 fsize -= offset;
4072
4073             if (fsize < offset)
4074                 fsize = 0;
4075             else
4076                 fsize -= offset;
4077
4078             if (fsize < offset)
4079                 fsize = 0;
4080             else
4081                 fsize -= offset;
4082
4083             if (fsize < offset)
4084                 fsize = 0;
4085             else
4086                 fsize -= offset;
4087
4088             if (fsize < offset)
4089                 fsize = 0;
4090             else
4091                 fsize -= offset;
4092
4093             if (fsize < offset)
4094                 fsize = 0;
4095             else
4096                 fsize -= offset;
4097
4098             if (fsize < offset)
4099                 fsize = 0;
4100             else
4101                 fsize -= offset;
4102
4103             if (fsize < offset)
4104                 fsize = 0;
4105             else
4106                 fsize -= offset;
4107
4108             if (fsize < offset)
4109                 fsize = 0;
4110             else
4111                 fsize -= offset;
4112
4113             if (fsize < offset)
4114                 fsize = 0;
4115             else
4116                 fsize -= offset;
4117
4118             if (fsize < offset)
4119                 fsize = 0;
4120             else
4121                 fsize -= offset;
4122
4123             if (fsize < offset)
4124                 fsize = 0;
4125             else
4126                 fsize -= offset;
4127
4128             if (fsize < offset)
4129                 fsize = 0;
4130             else
4131                 fsize -= offset;
4132
4133             if (fsize < offset)
4134                 fsize = 0;
4135             else
4136                 fsize -= offset;
4137
4138             if (fsize < offset)
4139                 fsize = 0;
4140             else
4141                 fsize -= offset;
4142
4143             if (fsize < offset)
4144                 fsize = 0;
4145             else
4146                 fsize -= offset;
4147
4148             if (fsize < offset)
4149                 fsize = 0;
4150             else
4151                 fsize -= offset;
4152
4153             if (fsize < offset)
4154                 fsize = 0;
4155             else
4156                 fsize -= offset;
4157
4158             if (fsize < offset)
4159                 fsize = 0;
4160             else
4161                 fsize -= offset;
4162
4163             if (fsize < offset)
4164                 fsize = 0;
4165             else
4166                 fsize -= offset;
4167
4168             if (fsize < offset)
4169                 fsize = 0;
4170             else
4171                 fsize -= offset;
4172
4173             if (fsize < offset)
4174                 fsize = 0;
4175             else
4176                 fsize -= offset;
4177
4178             if (fsize < offset)
4179                 fsize = 0;
4180             else
4181                 fsize -= offset;
4182
4183             if (fsize < offset)
4184                 fsize = 0;
4185             else
4186                 fsize -= offset;
4187
4188             if (fsize < offset)
4189                 fsize = 0;
4190             else
4191                 fsize -= offset;
4192
4193             if (fsize < offset)
4194                 fsize = 0;
4195             else
4196                 fsize -= offset;
4197
4198             if (fsize < offset)
4199                 fsize = 0;
4200             else
4201                 fsize -= offset;
4202
4203             if (fsize < offset)
4204                 fsize = 0;
4205             else
4206                 fsize -= offset;
4207
4208             if (fsize < offset)
4209                 fsize = 0;
4210             else
4211                 fsize -= offset;
4212
4213             if (fsize < offset)
4214                 fsize = 0;
4215             else
4216                 fsize -= offset;
4217
4218             if (fsize < offset)
4219                 fsize = 0;
4220             else
4221                 fsize -= offset;
4222
4223             if (fsize < offset)
4224                 fsize = 0;
4225             else
4226                 fsize -= offset;
4227
4228             if (fsize < offset)
4229                 fsize = 0;
4230             else
4231                 fsize -= offset;
4232
4233             if (fsize < offset)
4234                 fsize = 0;
4235             else
4236                 fsize -= offset;
4237
4238             if (fsize < offset)
4239                 fsize = 0;
4240             else
4241                 fsize -= offset;
4242
4243             if (fsize < offset)
4244                 fsize = 0;
4245             else
4246                 fsize -= offset;
4247
4248             if (fsize < offset)
4249                 fsize = 0;
4250             else
4251                 fsize -= offset;
4252
4253             if (fsize < offset)
4254                 fsize = 0;
4255             else
4256                 fsize -= offset;
4257
4258             if (fsize < offset)
4259                 fsize = 0;
4260             else
4261                 fsize -= offset;
4262
4263             if (fsize < offset)
4264                 fsize = 0;
4265             else
4266                 fsize -= offset;
4267
4268             if (fsize < offset)
4269                 fsize = 0;
4270             else
4271                 fsize -= offset;
4272
4273             if (fsize < offset)
4274                 fsize = 0;
4275             else
4276                 fsize -= offset;
4277
4278             if (fsize < offset)
4279                 fsize = 0;
4280             else
4281                 fsize -= offset;
4282
4283             if (fsize < offset)
4284                 fsize = 0;
4285             else
4286                 fsize -= offset;
4287
4288             if (fsize < offset)
4289                 fsize = 0;
4290             else
4291                 fsize -= offset;
4292
4293             if (fsize < offset)
4294                 fsize = 0;
4295             else
4296                 fsize -= offset;
4297
4298             if (fsize < offset)
4299                 fsize = 0;
4300             else
4301                 fsize -= offset;
4302
4303             if (fsize < offset)
4304                 fsize = 0;
4305             else
4306                 fsize -= offset;
4307
4308             if (fsize < offset)
4309                 fsize = 0;
4310             else
4311                 fsize -= offset;
4312
4313             if (fsize < offset)
4314                 fsize = 0;
4315             else
4316                 fsize -= offset;
4317
4318             if (fsize < offset)
4319                 fsize = 0;
4320             else
4321                 fsize -= offset;
4322
4323             if (fsize < offset)
4324                 fsize = 0;
4325             else
4326                 fsize -= offset;
4327
4328             if (fsize < offset)
4329                 fsize = 0;
4330             else
4331                 fsize -= offset;
4332
4333             if (fsize < offset)
4334                 fsize = 0;
4335             else
4336                 fsize -= offset;
4337
4338             if (fsize < offset)
4339                 fsize = 0;
4340             else
4341                 fsize -= offset;
4342
4343             if (fsize < offset)
4344                 fsize = 0;
4345             else
4346                 fsize -= offset;
4347
4348             if (fsize < offset)
4349                 fsize = 0;
4350             else
4351                 fsize -= offset;
4352
4353             if (fsize < offset)
4354                 fsize = 0;
4355             else
4356                 fsize -= offset;
4357
4358             if (fsize < offset)
4359                 fsize = 0;
4360             else
4361                 fsize -= offset;
4362
4363             if (fsize < offset)
4364                 fsize = 0;
4365             else
4366                 fsize -= offset;
4367
4368             if (fsize < offset)
4369                 fsize = 0;
4370             else
4371                 fsize -= offset;
4372
4373             if (fsize < offset)
4374                 fsize = 0;
4375             else
4376                 fsize -= offset;
4377
4378             if (fsize < offset)
4379                 fsize = 0;
4380             else
4381                 fsize -= offset;
4382
4383             if (fsize < offset)
4384                 fsize = 0;
4385             else
4386                 fsize -= offset;
4387
4388             if (fsize < offset)
4389                 fsize = 0;
4390             else
4391                 fsize -= offset;
4392
4393             if (fsize < offset)
4394                 fsize = 0;
4395             else
4396                 fsize -= offset;
4397
4398             if (fsize < offset)
4399                 fsize = 0;
4400             else
4401                 fsize -= offset;
4402
4403             if (fsize < offset)
4404                 fsize = 0;
4405             else
4406                 fsize -= offset;
4407
4408             if (fsize < offset)
4409                 fsize = 0;
4410             else
4411                 fsize -= offset;
4412
4413             if (fsize < offset)
4414                 fsize = 0;
4415             else
4416                 fsize -= offset;
4417
4418             if (fsize < offset)
4419                 fsize = 0;
4420             else
4421                 fsize -= offset;
4422
4423             if (fsize < offset)
4424                 fsize = 0;
4425             else
4426                 fsize -= offset;
4427
4428             if (fsize < offset)
4429                 fsize = 0;
4430             else
4431                 fsize -= offset;
4432
4433             if (fsize < offset)
4434                 fsize = 0;
4435             else
4436                 fsize -= offset;
4437
4438             if (fsize < offset)
4439                 fsize = 0;
4440             else
4441                 fsize -= offset;
4442
4443             if (fsize < offset)
4444                 fsize = 0;
4445             else
4446                 fsize -= offset;
4447
4448             if (fsize < offset)
4449                 fsize = 0;
4450             else
4451                 fsize -= offset;
4452
4453             if (fsize < offset)
4454                 fsize = 0;
4455             else
4456                 fsize -= offset;
4457
4458             if (fsize < offset)
4459                 fsize = 0;
4460             else
4461                 fsize -= offset;
4462
4463             if (fsize < offset)
4464                 fsize = 0;
4465             else
4466                 fsize -= offset;
4467
4468             if (fsize < offset)
4469                 fsize = 0;
4470             else
4471                 fsize -= offset;
4472
4473             if (fsize < offset)
4474                 fsize = 0;
4475             else
4476                 fsize -= offset;
4477
4478             if (fsize < offset)
4479                 fsize = 0;
4480             else
4481                 fsize -= offset;
4482
4483             if (fsize < offset)
4484                 fsize = 0;
4485             else
4486                 fsize -= offset;
4487
4488             if (fsize < offset)
4489                 fsize = 0;
4490             else
4491                 fsize -= offset;
4492
4493             if (fsize < offset)
4494                 fsize = 0;
4495             else
4496                 fsize -= offset;
4497
4498             if (fsize < offset)
4499                 fsize = 0;
4500             else
4501                 fsize -= offset;
4502
4503             if (fsize < offset)
4504                 fsize = 0;
4505             else
4506                 fsize -= offset;
4507
4508             if (fsize < offset)
4509                 fsize = 0;
4510             else
4511                 fsize -= offset;
4512
4513             if (fsize < offset)
4514                 fsize = 0;
4515             else
4516                 fsize -= offset;
4517
4518             if (fsize < offset)
4519                 fsize = 0;
4520             else
4521                 fsize -= offset;
4522
4523             if (fsize < offset)
4524                 fsize = 0;
4525             else
4526                 fsize -= offset;
4527
4528             if (fsize < offset)
4529                 fsize = 0;
4530             else
4531                 fsize -= offset;
4532
4533             if (fsize < offset)
4534                 fsize = 0;
4535             else
4536                 fsize -= offset;
4537
4538             if (fsize < offset)
4539                 fsize = 0;
4540             else
4541                 fsize -= offset;
4542
4543             if (fsize < offset)
4544                 fsize = 0;
4545             else
4546                 fsize -= offset;
4547
4548             if (fsize < offset)
4549                 fsize = 0;
4550             else
4551                 fsize -= offset;
4552
4553             if (fsize < offset)
4554                 fsize = 0;
4555             else
4556                 fsize -= offset;
4557
4558             if (fsize < offset)
4559                 fsize = 0;
4560             else
4561                 fsize -= offset;
4562
4563             if (fsize < offset)
4564                 fsize = 0;
4565             else
4566                 fsize -= offset;
4567
4568             if (fsize < offset)
4569                 fsize = 0;
4570             else
4571                 fsize -= offset;
4572
4573             if (fsize < offset)
4574                 fsize = 0;
4575             else
4576                 fsize -= offset;
4577
4578             if (fsize < offset)
4579                 fsize = 0;
4580             else
4581                 fsize -= offset;
4582
4583             if (fsize < offset)
4584                 fsize = 0;
4585             else
4586                 fsize -= offset;
4587
4588             if (fsize < offset)
4589                 fsize = 0;
4590             else
4591                 fsize -= offset;
4592
4593             if (fsize < offset)
4594                 fsize = 0;
4595             else
4596                 fsize -= offset;
4597
4598             if (fsize < offset)
4599                 fsize = 0;
4600             else
4601                 fsize -= offset;
4602
4603             if (fsize < offset)
4604                 fsize = 0;
4605             else
4606                 fsize -= offset;
4607
4608             if (fsize < offset)
4609                 fsize = 0;
4610             else
4611                 fsize -= offset;
4612
4613             if (fsize < offset)
4614                 fsize = 0;
4615             else
4616                 fsize -= offset;
4617
4618             if (fsize < offset)
4619                 fsize = 0;
4620             else
4621                 fsize -= offset;
4622
4623             if (fsize < offset)
4624                 fsize = 0;
4625             else
4626                 fsize -= offset;
4627
4628             if (fsize < offset)
4629                 fsize = 0;
4630             else
4631                 fsize -= offset;
4632
4633             if (fsize < offset)
4634                 fsize = 0;
4635             else
4636                 fsize -= offset;
4637
4638             if (fsize < offset)
4639                 fsize = 0;
4640             else
4641                 fsize -= offset;
4642
4643             if (fsize < offset)
4644                 fsize = 0;
4645             else
4646                 fsize -= offset;
4647
4648             if (fsize < offset)
4649                 fsize = 0;
4650             else
4651                 fsize -= offset;
4652
4653             if (fsize < offset)
4654                 fsize = 0;
4655             else
4656                 fsize -= offset;
4657
4658             if (fsize < offset)
4659                 fsize = 0;
4660             else
4661                 fsize -= offset;
4662
4663             if (fsize < offset)
4664                 fsize = 0;
4665             else
4666                 fsize -= offset;
4667
4668             if (fsize < offset)
4669                 fsize = 0;
4670             else
4671                 fsize -= offset;
4672
4673             if (fsize < offset)
4674                 fsize = 0;
4675             else
4676                 fsize -= offset;
4677
4678             if (fsize < offset)
4679                 fsize = 0;
4680             else
4681                 fsize -= offset;
4682
4683             if (fsize < offset)
4684                 fsize = 0;
4685             else
4686                 fsize -= offset;
4687
4688             if (fsize < offset)
4689                 fsize = 0;
4690             else
4691                 fsize -= offset;
4692
4693             if (fsize < offset)
4694                 fsize = 0;
4695             else
4696                 fsize -= offset;
4697
4698             if (fsize < offset)
4699                 fsize = 0;
4700             else
4701                 fsize -= offset;
4702
4703             if (fsize < offset)
4704                 fsize = 0;
4705             else
4706                 fsize -= offset;
4707
4708             if (fsize < offset)
4709                 fsize = 0;
4710             else
4711                 fsize -= offset;
4712
4713             if (fsize < offset)
4714                 fsize = 0;
4715             else
4716                 fsize -= offset;
4717
4718             if (fsize < offset)
4719                 fsize = 0;
4720             else
4721                 fsize -= offset;
4722
4723             if (fsize < offset)
4724                 fsize = 0;
4725             else
4726                 fsize -= offset;
4727
4728             if (fsize < offset)
4729                 fsize = 0;
4730             else
4731                 fsize -= offset;
4732
4733             if (fsize < offset)
4734                 fsize = 0;
4735             else
4736                 fsize -= offset;
4737
4738             if (fsize < offset)
4739                 fsize = 0;
4740             else
4741                 fsize -= offset;
4742
4743             if (fsize < offset)
4744                 fsize = 0;
4745             else
4746                 fsize -= offset;
4747
4748             if (fsize < offset)
4749                 fsize = 0;
4750             else
4751                 fsize -= offset;
4752
4753             if (fsize < offset)
4754                 fsize = 0;
4755             else
4756                 fsize -= offset;
4757
4758             if (fsize < offset)
4759                 fsize = 0;
4760             else
4761                 fsize -= offset;
4762
4763             if (fsize < offset)
4764                 fsize = 0;
4765             else
4766                 fsize -= offset;
4767
4768             if (fsize < offset)
4769                 fsize = 0;
4770             else
4771                 fsize -= offset;
4772
4773             if (fsize < offset)
4774                 fsize = 0;
4775             else
4776                 fsize -= offset;
4777
4778             if (fsize < offset)
4779                 fsize = 0;
4780             else
4781                 fsize -= offset;
4782
4783             if (fsize < offset)
4784                 fsize = 0;
4785             else
4786                 fsize -= offset;
4787
4788             if (fsize < offset)
4789                 fsize = 0;
4790             else
4791                 fsize -= offset;
4792
4793             if (fsize < offset)
4794                 fsize = 0;
4795             else
4796                 fsize -= offset;
4797
4798             if (fsize < offset)
4799                 fsize = 0;
4800             else
4801                 fsize -= offset;
4802
4803             if (fsize < offset)
4804                 fsize = 0;
4805             else
4806                 fsize -= offset;
4807
4808             if (fsize < offset)
4809                 fsize = 0;
4810             else
4811                 fsize -= offset;
4812
4813             if (fsize < offset)
4814                 fsize = 0;
4815             else
4816                 fsize -= offset;
4817

```

```

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;
4034
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);
4043
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);
4053
4054                 mp = pr_iol_newbuf(iolhead, sizeof (*mp));
4055
4056                 mp->pr_vaddr = (uintptr_t)saddr;
4057                 mp->pr_size = naddr - saddr;
4058                 mp->pr_offset = segop_getoffset(seg, saddr);
4059                 mp->pr_offset = SEGOP_GETOFFSET(seg, saddr);
4060                 mp->pr_mflags = 0;
4061                 if (prot & PROT_READ)
4062                     mp->pr_mflags |= MA_READ;
4063                 if (prot & PROT_WRITE)
4064                     mp->pr_mflags |= MA_WRITE;
4065                 if (prot & PROT_EXEC)
4066                     mp->pr_mflags |= MA_EXEC;
4067                 if (segop_gettime(seg, saddr) & MAP_SHARED)
4068                     if (SEGOP_GETTYPE(seg, saddr) & MAP_SHARED)
4069                         mp->pr_mflags |= MA_SHARED;
4070                 if (segop_gettime(seg, saddr) & MAP_NORESERVE)
4071                     if (SEGOP_GETTYPE(seg, saddr) & MAP_NORESERVE)
4072                         mp->pr_mflags |= MA_NORESERVE;
4073                 if (seg->s_ops == &segsp_shmops ||
4074                     (seg->s_ops == &segvn_ops &&
4075                     (segop_getvp(seg, saddr, &vp) != 0 ||
4076                      (SEGOP_GETVP(seg, saddr, &vp) != 0 ||
4077                      vp == NULL)))
4078                     mp->pr_mflags |= MA_ANON;
4079                 if (seg == brkseg)
4080                     mp->pr_mflags |= MA_BREAK;
4081                 else if (seg == stkseg)
4082                     mp->pr_mflags |= MA_STACK;
4083                 if (seg->s_ops == &segsp_shmops)
4084                     mp->pr_mflags |= MA_ISM | MA_SHM;
4085
4086                 mp->pr_pagesize = PAGESIZE;
4087                 if (psz == -1) {
4088                     mp->pr_hatpagesize = 0;
4089                 } else {
4090                     mp->pr_hatpagesize = psz;
4091                 }
4092             }
4093         }
4094     }
4095 }
```

```

4096     /*
4097      * Manufacture a filename for the "object" dir.
4098      */
4099     mp->pr_dev = PRNODEV;
4100     vattr.va_mask = AT_FSID|AT_NODEID;
4101     if (seg->s_ops == &segvn_ops &&
4102         segop_getvp(seg, saddr, &vp) == 0 &&
4103         SEGOP_GETVP(seg, saddr, &vp) == 0 &&
4104         vp != NULL && vp->v_type == VREG &&
4105         VOP_GETATTR(vp, &vattr, 0, CRED(), NULL) == 0) {
4106         mp->pr_dev = vattr.va_fsid;
4107         mp->pr_ino = vattr.va_nodeid;
4108         if (vp == p->p_exec)
4109             (void) strcpy(mp->pr_mapname,
4110                          "a.out");
4111         else
4112             pr_object_name(mp->pr_mapname,
4113                            vp, &vattr);
4114     }
4115
4116     /*
4117      * Get the SysV shared memory id, if any.
4118      */
4119     if ((mp->pr_mflags & MA_SHARED) &&
4120         p->p_segacct && (mp->pr_shmid = shmgetid(p,
4121             seg->s_base)) != SHMID_NONE) {
4122         if (mp->pr_shmid == SHMID_FREE)
4123             mp->pr_shmid = -1;
4124     } else {
4125         mp->pr_mflags |= MA_SHM;
4126     }
4127
4128     npages = ((uintptr_t)(naddr - saddr)) >>
4129             PAGESHIFT;
4130     parr = kmalloc(npages, KM_SLEEP);
4131
4132     (void) segop_incore(seg, saddr, naddr - saddr, parr);
4133     SEGOP_INCORE(seg, saddr, naddr - saddr, parr);
4134
4135     for (pagenum = 0; pagenum < npages; pagenum++) {
4136         if (parr[pagenum] & SEG_PAGE_INCORE)
4137             mp->pr_rss++;
4138         if (parr[pagenum] & SEG_PAGE_ANON)
4139             mp->pr_anon++;
4140         if (parr[pagenum] & SEG_PAGE_LOCKED)
4141             mp->pr_locked++;
4142     }
4143     kmem_free(parr, npages);
4144 }
4145 }
```

unchanged\_portion\_omitted

```

4146
4147     ASSERT(tmp == NULL);
4148 } while ((seg = AS_SEGNEXT(as, seg)) != NULL);
4149
4150 return (0);
4151 }
```

unchanged\_portion\_omitted

```

4152
4153 #ifdef _SYSCALL32_IMPL
4154 */
4155 * Return an array of structures with HAT memory map information.
4156 * We allocate here; the caller must deallocate.
4157 }
```

```

4185 */
4186 int
4187 prgetxmap32(proc_t *p, list_t *iolhead)
4188 {
4189     struct as *as = p->p_as;
4190     prxmap32_t *mp;
4191     struct seg *seg;
4192     struct seg *brkseg, *stkseg;
4193     struct vnode *vp;
4194     struct vattr vattr;
4195     uint_t prot;
4196
4197     ASSERT(as != &kas && AS_WRITE_HELD(as, &as->a_lock));
4198
4199     /*
4200      * Request an initial buffer size that doesn't waste memory
4201      * if the address space has only a small number of segments.
4202      */
4203     pr_iol_initlist(iolhead, sizeof (*mp), avl_numnodes(&as->a_segtree));
4204
4205     if ((seg = AS_SEGFIRST(as)) == NULL)
4206         return (0);
4207
4208     brkseg = break_seg(p);
4209     stkseg = as_segat(as, prgetstackbase(p));
4210
4211     do {
4212         caddr_t eaddr = seg->s_base + pr_getsegsize(seg, 0);
4213         caddr_t saddr, naddr, baddr;
4214         void *tmp = NULL;
4215         ssize_t psz;
4216         char *parr;
4217         uint64_t npages;
4218         uint64_t pagenum;
4219
4220         /*
4221          * Segment loop part one: iterate from the base of the segment
4222          * to its end, pausing at each address boundary (baddr) between
4223          * ranges that have different virtual memory protections.
4224          */
4225         for (saddr = seg->s_base; saddr < eaddr; saddr = baddr) {
4226             prot = pr_getprot(seg, 0, &tmp, &saddr, &baddr, eaddr);
4227             ASSERT(baddr >= saddr && baddr <= eaddr);
4228
4229             /*
4230              * Segment loop part two: iterate from the current
4231              * position to the end of the protection boundary,
4232              * pausing at each address boundary (naddr) between
4233              * ranges that have different underlying page sizes.
4234              */
4235             for (; saddr < baddr; saddr = naddr) {
4236                 psz = pr_getpagesize(seg, saddr, &naddr, baddr);
4237                 ASSERT(naddr >= saddr && naddr <= baddr);
4238
4239                 mp = pr_iol_newbuf(iolhead, sizeof (*mp));
4240
4241                 mp->pr_vaddr = (caddr32_t)(uintptr_t)saddr;
4242                 mp->pr_size = (size32_t)(naddr - saddr);
4243                 mp->pr_offset = segop_getoffset(seg, saddr);
4244                 mp->pr_offset = SEGOP_GETOFFSET(seg, saddr);
4245                 mp->pr_mflags = 0;
4246                 if (prot & PROT_READ)
4247                     mp->pr_mflags |= MA_READ;
4248                 if (prot & PROT_WRITE)
4249                     mp->pr_mflags |= MA_WRITE;
4250                 if (prot & PROT_EXEC)
4251                     mp->pr_mflags |= MA_EXEC;
4252             }
4253         }
4254     }
4255
4256     if (tmp)
4257         free(tmp, M_TEMP);
4258
4259     if (brkseg)
4260         break_free(brkseg);
4261
4262     if (stkseg)
4263         stack_free(stkseg);
4264
4265     if (vp)
4266         vattr_free(vattr);
4267
4268     if (parr)
4269         free(parr, M_TEMP);
4270
4271     if (npages)
4272         free(npages, M_TEMP);
4273
4274     if (pagenum)
4275         free(pagenum, M_TEMP);
4276
4277     if (psz)
4278         free(psz, M_TEMP);
4279
4280     if (tmp)
4281         free(tmp, M_TEMP);
4282
4283     if (naddr)
4284         free(naddr, M_TEMP);
4285
4286     if (baddr)
4287         free(baddr, M_TEMP);
4288
4289     if (eaddr)
4290         free(eaddr, M_TEMP);
4291
4292     if (saddr)
4293         free(saddr, M_TEMP);
4294
4295     if (seg)
4296         free(seg, M_TEMP);
4297
4298     if (tmp)
4299         free(tmp, M_TEMP);
4300
4301     if (seg)
4302         free(seg, M_TEMP);
4303
4304     if (tmp)
4305         free(tmp, M_TEMP);
4306
4307     if (seg)
4308         free(seg, M_TEMP);
4309
4310     if (tmp)
4311         free(tmp, M_TEMP);
4312
4313     if (seg)
4314         free(seg, M_TEMP);
4315
4316     if (tmp)
4317         free(tmp, M_TEMP);
4318
4319     if (seg)
4320         free(seg, M_TEMP);
4321
4322     if (tmp)
4323         free(tmp, M_TEMP);
4324
4325     if (seg)
4326         free(seg, M_TEMP);
4327
4328     if (tmp)
4329         free(tmp, M_TEMP);
4330
4331     if (seg)
4332         free(seg, M_TEMP);
4333
4334     if (tmp)
4335         free(tmp, M_TEMP);
4336
4337     if (seg)
4338         free(seg, M_TEMP);
4339
4340     if (tmp)
4341         free(tmp, M_TEMP);
4342
4343     if (seg)
4344         free(seg, M_TEMP);
4345
4346     if (tmp)
4347         free(tmp, M_TEMP);
4348
4349     if (seg)
4350         free(seg, M_TEMP);
4351
4352     if (tmp)
4353         free(tmp, M_TEMP);
4354
4355     if (seg)
4356         free(seg, M_TEMP);
4357
4358     if (tmp)
4359         free(tmp, M_TEMP);
4360
4361     if (seg)
4362         free(seg, M_TEMP);
4363
4364     if (tmp)
4365         free(tmp, M_TEMP);
4366
4367     if (seg)
4368         free(seg, M_TEMP);
4369
4370     if (tmp)
4371         free(tmp, M_TEMP);
4372
4373     if (seg)
4374         free(seg, M_TEMP);
4375
4376     if (tmp)
4377         free(tmp, M_TEMP);
4378
4379     if (seg)
4380         free(seg, M_TEMP);
4381
4382     if (tmp)
4383         free(tmp, M_TEMP);
4384
4385     if (seg)
4386         free(seg, M_TEMP);
4387
4388     if (tmp)
4389         free(tmp, M_TEMP);
4390
4391     if (seg)
4392         free(seg, M_TEMP);
4393
4394     if (tmp)
4395         free(tmp, M_TEMP);
4396
4397     if (seg)
4398         free(seg, M_TEMP);
4399
4400     if (tmp)
4401         free(tmp, M_TEMP);
4402
4403     if (seg)
4404         free(seg, M_TEMP);
4405
4406     if (tmp)
4407         free(tmp, M_TEMP);
4408
4409     if (seg)
4410         free(seg, M_TEMP);
4411
4412     if (tmp)
4413         free(tmp, M_TEMP);
4414
4415     if (seg)
4416         free(seg, M_TEMP);
4417
4418     if (tmp)
4419         free(tmp, M_TEMP);
4420
4421     if (seg)
4422         free(seg, M_TEMP);
4423
4424     if (tmp)
4425         free(tmp, M_TEMP);
4426
4427     if (seg)
4428         free(seg, M_TEMP);
4429
4430     if (tmp)
4431         free(tmp, M_TEMP);
4432
4433     if (seg)
4434         free(seg, M_TEMP);
4435
4436     if (tmp)
4437         free(tmp, M_TEMP);
4438
4439     if (seg)
4440         free(seg, M_TEMP);
4441
4442     if (tmp)
4443         free(tmp, M_TEMP);
4444
4445     if (seg)
4446         free(seg, M_TEMP);
4447
4448     if (tmp)
4449         free(tmp, M_TEMP);
4450
4451     if (seg)
4452         free(seg, M_TEMP);
4453
4454     if (tmp)
4455         free(tmp, M_TEMP);
4456
4457     if (seg)
4458         free(seg, M_TEMP);
4459
4460     if (tmp)
4461         free(tmp, M_TEMP);
4462
4463     if (seg)
4464         free(seg, M_TEMP);
4465
4466     if (tmp)
4467         free(tmp, M_TEMP);
4468
4469     if (seg)
4470         free(seg, M_TEMP);
4471
4472     if (tmp)
4473         free(tmp, M_TEMP);
4474
4475     if (seg)
4476         free(seg, M_TEMP);
4477
4478     if (tmp)
4479         free(tmp, M_TEMP);
4480
4481     if (seg)
4482         free(seg, M_TEMP);
4483
4484     if (tmp)
4485         free(tmp, M_TEMP);
4486
4487     if (seg)
4488         free(seg, M_TEMP);
4489
4490     if (tmp)
4491         free(tmp, M_TEMP);
4492
4493     if (seg)
4494         free(seg, M_TEMP);
4495
4496     if (tmp)
4497         free(tmp, M_TEMP);
4498
4499     if (seg)
4500         free(seg, M_TEMP);
4501
4502     if (tmp)
4503         free(tmp, M_TEMP);
4504
4505     if (seg)
4506         free(seg, M_TEMP);
4507
4508     if (tmp)
4509         free(tmp, M_TEMP);
4510
4511     if (seg)
4512         free(seg, M_TEMP);
4513
4514     if (tmp)
4515         free(tmp, M_TEMP);
4516
4517     if (seg)
4518         free(seg, M_TEMP);
4519
4520     if (tmp)
4521         free(tmp, M_TEMP);
4522
4523     if (seg)
4524         free(seg, M_TEMP);
4525
4526     if (tmp)
4527         free(tmp, M_TEMP);
4528
4529     if (seg)
4530         free(seg, M_TEMP);
4531
4532     if (tmp)
4533         free(tmp, M_TEMP);
4534
4535     if (seg)
4536         free(seg, M_TEMP);
4537
4538     if (tmp)
4539         free(tmp, M_TEMP);
4540
4541     if (seg)
4542         free(seg, M_TEMP);
4543
4544     if (tmp)
4545         free(tmp, M_TEMP);
4546
4547     if (seg)
4548         free(seg, M_TEMP);
4549
4550     if (tmp)
4551         free(tmp, M_TEMP);
4552
4553     if (seg)
4554         free(seg, M_TEMP);
4555
4556     if (tmp)
4557         free(tmp, M_TEMP);
4558
4559     if (seg)
4560         free(seg, M_TEMP);
4561
4562     if (tmp)
4563         free(tmp, M_TEMP);
4564
4565     if (seg)
4566         free(seg, M_TEMP);
4567
4568     if (tmp)
4569         free(tmp, M_TEMP);
4570
4571     if (seg)
4572         free(seg, M_TEMP);
4573
4574     if (tmp)
4575         free(tmp, M_TEMP);
4576
4577     if (seg)
4578         free(seg, M_TEMP);
4579
4580     if (tmp)
4581         free(tmp, M_TEMP);
4582
4583     if (seg)
4584         free(seg, M_TEMP);
4585
4586     if (tmp)
4587         free(tmp, M_TEMP);
4588
4589     if (seg)
4590         free(seg, M_TEMP);
4591
4592     if (tmp)
4593         free(tmp, M_TEMP);
4594
4595     if (seg)
4596         free(seg, M_TEMP);
4597
4598     if (tmp)
4599         free(tmp, M_TEMP);
4599
4600     if (seg)
4601         free(seg, M_TEMP);
4602
4603     if (tmp)
4604         free(tmp, M_TEMP);
4605
4606     if (seg)
4607         free(seg, M_TEMP);
4608
4609     if (tmp)
4610         free(tmp, M_TEMP);
4611
4612     if (seg)
4613         free(seg, M_TEMP);
4614
4615     if (tmp)
4616         free(tmp, M_TEMP);
4617
4618     if (seg)
4619         free(seg, M_TEMP);
4620
4621     if (tmp)
4622         free(tmp, M_TEMP);
4623
4624     if (seg)
4625         free(seg, M_TEMP);
4626
4627     if (tmp)
4628         free(tmp, M_TEMP);
4629
4630     if (seg)
4631         free(seg, M_TEMP);
4632
4633     if (tmp)
4634         free(tmp, M_TEMP);
4635
4636     if (seg)
4637         free(seg, M_TEMP);
4638
4639     if (tmp)
4640         free(tmp, M_TEMP);
4641
4642     if (seg)
4643         free(seg, M_TEMP);
4644
4645     if (tmp)
4646         free(tmp, M_TEMP);
4647
4648     if (seg)
4649         free(seg, M_TEMP);
4650
4651     if (tmp)
4652         free(tmp, M_TEMP);
4653
4654     if (seg)
4655         free(seg, M_TEMP);
4656
4657     if (tmp)
4658         free(tmp, M_TEMP);
4659
4660     if (seg)
4661         free(seg, M_TEMP);
4662
4663     if (tmp)
4664         free(tmp, M_TEMP);
4665
4666     if (seg)
4667         free(seg, M_TEMP);
4668
4669     if (tmp)
4670         free(tmp, M_TEMP);
4671
4672     if (seg)
4673         free(seg, M_TEMP);
4674
4675     if (tmp)
4676         free(tmp, M_TEMP);
4677
4678     if (seg)
4679         free(seg, M_TEMP);
4680
4681     if (tmp)
4682         free(tmp, M_TEMP);
4683
4684     if (seg)
4685         free(seg, M_TEMP);
4686
4687     if (tmp)
4688         free(tmp, M_TEMP);
4689
4690     if (seg)
4691         free(seg, M_TEMP);
4692
4693     if (tmp)
4694         free(tmp, M_TEMP);
4695
4696     if (seg)
4697         free(seg, M_TEMP);
4698
4699     if (tmp)
4699         free(tmp, M_TEMP);
4700
4701     if (seg)
4702         free(seg, M_TEMP);
4703
4704     if (tmp)
4705         free(tmp, M_TEMP);
4706
4707     if (seg)
4708         free(seg, M_TEMP);
4709
4710     if (tmp)
4711         free(tmp, M_TEMP);
4712
4713     if (seg)
4714         free(seg, M_TEMP);
4715
4716     if (tmp)
4717         free(tmp, M_TEMP);
4718
4719     if (seg)
4720         free(seg, M_TEMP);
4721
4722     if (tmp)
4723         free(tmp, M_TEMP);
4724
4725     if (seg)
4726         free(seg, M_TEMP);
4727
4728     if (tmp)
4729         free(tmp, M_TEMP);
4729
4730     if (seg)
4731         free(seg, M_TEMP);
4732
4733     if (tmp)
4734         free(tmp, M_TEMP);
4735
4736     if (seg)
4737         free(seg, M_TEMP);
4738
4739     if (tmp)
4740         free(tmp, M_TEMP);
4740
4741     if (seg)
4742         free(seg, M_TEMP);
4743
4744     if (tmp)
4745         free(tmp, M_TEMP);
4745
4746     if (seg)
4747         free(seg, M_TEMP);
4748
4749     if (tmp)
4750         free(tmp, M_TEMP);
4750
4751     if (seg)
4752         free(seg, M_TEMP);
4753
4754     if (tmp)
4755         free(tmp, M_TEMP);
4755
4756     if (seg)
4757         free(seg, M_TEMP);
4758
4759     if (tmp)
4760         free(tmp, M_TEMP);
4760
4761     if (seg)
4762         free(seg, M_TEMP);
4763
4764     if (tmp)
4765         free(tmp, M_TEMP);
4765
4766     if (seg)
4767         free(seg, M_TEMP);
4768
4769     if (tmp)
4770         free(tmp, M_TEMP);
4770
4771     if (seg)
4772         free(seg, M_TEMP);
4773
4774     if (tmp)
4775         free(tmp, M_TEMP);
4775
4776     if (seg)
4777         free(seg, M_TEMP);
4778
4779     if (tmp)
4780         free(tmp, M_TEMP);
4780
4781     if (seg)
4782         free(seg, M_TEMP);
4783
4784     if (tmp)
4785         free(tmp, M_TEMP);
4785
4786     if (seg)
4787         free(seg, M_TEMP);
4788
4789     if (tmp)
4790         free(tmp, M_TEMP);
4790
4791     if (seg)
4792         free(seg, M_TEMP);
4793
4794     if (tmp)
4795         free(tmp, M_TEMP);
4795
4796     if (seg)
4797         free(seg, M_TEMP);
4798
4799     if (tmp)
4799         free(tmp, M_TEMP);
4800
4801     if (seg)
4802         free(seg, M_TEMP);
4803
4804     if (tmp)
4805         free(tmp, M_TEMP);
4805
4806     if (seg)
4807         free(seg, M_TEMP);
4808
4809     if (tmp)
4810         free(tmp, M_TEMP);
4810
4811     if (seg)
4812         free(seg, M_TEMP);
4813
4814     if (tmp)
4815         free(tmp, M_TEMP);
4815
4816     if (seg)
4817         free(seg, M_TEMP);
4818
4819     if (tmp)
4820         free(tmp, M_TEMP);
4820
4821     if (seg)
4822         free(seg, M_TEMP);
4823
4824     if (tmp)
4825         free(tmp, M_TEMP);
4825
4826     if (seg)
4827         free(seg, M_TEMP);
4828
4829     if (tmp)
4830         free(tmp, M_TEMP);
4830
4831     if (seg)
4832         free(seg, M_TEMP);
4833
4834     if (tmp)
4835         free(tmp, M_TEMP);
4835
4836     if (seg)
4837         free(seg, M_TEMP);
4838
4839     if (tmp)
4840         free(tmp, M_TEMP);
4840
4841     if (seg)
4842         free(seg, M_TEMP);
4843
4844     if (tmp)
4845         free(tmp, M_TEMP);
4845
4846     if (seg)
4847         free(seg, M_TEMP);
4848
4849     if (tmp)
4850         free(tmp, M_TEMP);
4850
4851     if (seg)
4852         free(seg, M_TEMP);
4853
4854     if (tmp)
4855         free(tmp, M_TEMP);
4855
4856     if (seg)
4857         free(seg, M_TEMP);
4858
4859     if (tmp)
4860         free(tmp, M_TEMP);
4860
4861     if (seg)
4862         free(seg, M_TEMP);
4863
4864     if (tmp)
4865         free(tmp, M_TEMP);
4865
4866     if (seg)
4867         free(seg, M_TEMP);
4868
4869     if (tmp)
4870         free(tmp, M_TEMP);
4870
4871     if (seg)
4872         free(seg, M_TEMP);
4873
4874     if (tmp)
4875         free(tmp, M_TEMP);
4875
4876     if (seg)
4877         free(seg, M_TEMP);
4878
4879     if (tmp)
4880         free(tmp, M_TEMP);
4880
4881     if (seg)
4882         free(seg, M_TEMP);
4883
4884     if (tmp)
4885         free(tmp, M_TEMP);
4885
4886     if (seg)
4887         free(seg, M_TEMP);
4888
4889     if (tmp)
4890         free(tmp, M_TEMP);
4890
4891     if (seg)
4892         free(seg, M_TEMP);
4893
4894     if (tmp)
4895         free(tmp, M_TEMP);
4895
4896     if (seg)
4897         free(seg, M_TEMP);
4898
4899     if (tmp)
4900         free(tmp, M_TEMP);
4900
4901     if (seg)
4902         free(seg, M_TEMP);
4903
4904     if (tmp)
4905         free(tmp, M_TEMP);
4905
4906     if (seg)
4907         free(seg, M_TEMP);
4908
4909     if (tmp)
4910         free(tmp, M_TEMP);
4910
4911     if (seg)
4912         free(seg, M_TEMP);
4913
4914     if (tmp)
4915         free(tmp, M_TEMP);
4915
4916     if (seg)
4917         free(seg, M_TEMP);
4918
4919     if (tmp)
4920         free(tmp, M_TEMP);
4920
4921     if (seg)
4922         free(seg, M_TEMP);
4923
4924     if (tmp)
4925         free(tmp, M_TEMP);
4925
4926     if (seg)
4927         free(seg, M_TEMP);
4928
4929     if (tmp)
4930         free(tmp, M_TEMP);
4930
4931     if (seg)
4932         free(seg, M_TEMP);
4933
4934     if (tmp)
4935         free(tmp, M_TEMP);
4935
4936     if (seg)
4937         free(seg, M_TEMP);
4938
4939     if (tmp)
4940         free(tmp, M_TEMP);
4940
4941     if (seg)
4942         free(seg, M_TEMP);
4943
4944     if (tmp)
4945         free(tmp, M_TEMP);
4945
4946     if (seg)
4947         free(seg, M_TEMP);
4948
4949     if (tmp)
4950         free(tmp, M_TEMP);
4950
4951     if (seg)
4952         free(seg, M_TEMP);
4953
4954     if (tmp)
4955         free(tmp, M_TEMP);
4955
4956     if (seg)
4957         free(seg, M_TEMP);
4958
4959     if (tmp)
4960         free(tmp, M_TEMP);
4960
4961     if (seg)
4962         free(seg, M_TEMP);
4963
4964     if (tmp)
4965         free(tmp, M_TEMP);
4965
4966     if (seg)
4967         free(seg, M_TEMP);
4968
4969     if (tmp)
4970         free(tmp, M_TEMP);
4970
4971     if (seg)
4972         free(seg, M_TEMP);
4973
4974     if (tmp)
4975         free(tmp, M_TEMP);
4975
4976     if (seg)
4977         free(seg, M_TEMP);
4978
4979     if (tmp)
4980         free(tmp, M_TEMP);
4980
4981     if (seg)
4982         free(seg, M_TEMP);
4983
4984     if (tmp)
4985         free(tmp, M_TEMP);
4985
4986     if (seg)
4987         free(seg, M_TEMP);
4988
4989     if (tmp)
4990         free(tmp, M_TEMP);
4990
4991     if (seg)
4992         free(seg, M_TEMP);
4993
4994     if (tmp)
4995         free(tmp, M_TEMP);
4995
4996     if (seg)
4997         free(seg, M_TEMP);
4998
4999     if (tmp)
5000         free(tmp, M_TEMP);
5000
5001     if (seg)
5002         free(seg, M_TEMP);
5003
5004     if (tmp)
5005         free(tmp, M_TEMP);
5005
5006     if (seg)
5007         free(seg, M_TEMP);
5008
5009     if (tmp)
5010         free(tmp, M_TEMP);
5010
5011     if (seg)
5012         free(seg, M_TEMP);
5013
5014     if (tmp)
5015         free(tmp, M_TEMP);
5015
5016     if (seg)
5017         free(seg, M_TEMP);
5018
5019     if (tmp)
5020         free(tmp, M_TEMP);
5020
5021     if (seg)
5022         free(seg, M_TEMP);
5023
5024     if (tmp)
5025         free(tmp, M_TEMP);
5025
5026     if (seg)
5027         free(seg, M_TEMP);
5028
5029     if (tmp)
5030         free(tmp, M_TEMP);
5030
5031     if (seg)
5032         free(seg, M_TEMP);
5033
5034     if (tmp)
5035         free(tmp, M_TEMP);
5035
5036     if (seg)
5037         free(seg, M_TEMP);
5038
5039     if (tmp)
5040         free(tmp, M_TEMP);
5040
5041     if (seg)
5042         free(seg, M_TEMP);
5043
5044     if (tmp)
5045         free(tmp, M_TEMP);
5045
5046     if (seg)
5047         free(seg, M_TEMP);
5048
5049     if (tmp)
5050         free(tmp, M_TEMP);
5050
5051     if (seg)
5052         free(seg, M_TEMP);
5053
5054     if (tmp)
5055         free(tmp, M_TEMP);
5055
5056     if (seg)
5057         free(seg, M_TEMP);
5058
5059     if (tmp)
5060         free(tmp, M_TEMP);
5060
5061     if (seg)
5062         free(seg, M_TEMP);
5063
5064     if (tmp)
5065         free(tmp, M_TEMP);
5065
5066     if (seg)
5067         free(seg, M_TEMP);
5068
5069     if (tmp)
5070         free(tmp, M_TEMP);
5070
5071     if (seg)
5072         free(seg, M_TEMP);
5073
5074     if (tmp)
5075         free(tmp, M_TEMP);
5075
5076     if (seg)
5077         free(seg, M_TEMP);
5078
5079     if (tmp)
5080         free(tmp, M_TEMP);
5080
5081     if (seg)
5082         free(seg, M_TEMP);
5083
5084     if (tmp)
5085         free(tmp, M_TEMP);
5085
5086     if (seg)
5087         free(seg, M_TEMP);
5088
5089     if (tmp)
5090         free(tmp, M_TEMP);
5090
5091     if (seg)
5092         free(seg, M_TEMP);
5093
5094     if (tmp)
5095         free(tmp, M_TEMP);
5095
5096     if (seg)
5097         free(seg, M_TEMP);
5098
5099     if (tmp)
5099         free(tmp, M_TEMP);
5100
5101     if (seg)
5102         free(seg, M_TEMP);
5103
5104     if (tmp)
5105         free(tmp, M_TEMP);
5105
5106     if (seg)
5107         free(seg, M_TEMP);
5108
5109     if (tmp)
5110         free(tmp, M_TEMP);
5110
5111     if (seg)
5112         free(seg, M_TEMP);
5113
5114     if (tmp)
5115         free(tmp, M_TEMP);
5115
5116     if (seg)
5117         free(seg, M_TEMP);
5118
5119     if (tmp)
5120         free(tmp, M_TEMP);
5120
5121     if (seg)
5122         free(seg, M_TEMP);
5123
5124     if (tmp)
5125         free(tmp, M_TEMP);
5125
5126     if (seg)
5127         free(seg, M_TEMP);
5128
5129     if (tmp)
5130         free(tmp, M_TEMP);
5130
5131     if (seg)
5132         free(seg, M_TEMP);
5133
5134     if (tmp)
5135         free(tmp, M_TEMP);
5135
5136     if (seg)
5137         free(seg, M_TEMP);
5138
5139     if (tmp)
5140         free(tmp, M_TEMP);
5140
5141     if (seg)
5142         free(seg, M_TEMP);
5143
5144     if (tmp)
5145         free(tmp, M_TEMP);
5145
5146     if (seg)
5147         free(seg, M_TEMP);
5148
5149     if (tmp)
5150         free(tmp, M_TEMP);
5150
5151     if (seg)
5152         free(seg, M_TEMP);
5153
5154     if (tmp)
5155         free(tmp, M_TEMP);
5155
5156     if (seg)
5157         free(seg, M_TEMP);
5158
5159     if (tmp)
5160         free(tmp, M_TEMP);
5160
5161     if (seg)
5162         free(seg, M_TEMP);
5163
5164     if (tmp)
5165         free(tmp, M_TEMP);
5165
5166     if (seg)
5167         free(seg, M_TEMP);
5168
5169     if (tmp)
5170         free(tmp, M_TEMP);
5170
5171     if (seg)
5172         free(seg, M_TEMP);
5173
5174     if (tmp)
5175         free(tmp, M_TEMP);
5175
5176     if (seg)
5177         free(seg, M_TEMP);
5178
5179     if (tmp)
5180         free(tmp, M_TEMP);
5180
5181     if (seg)
5182         free(seg, M_TEMP);
5183
5184     if (tmp)
5185         free(tmp, M_TEMP);
5185
5186     if (seg)
5187         free(seg, M_TEMP);
5188
5189     if (tmp)
5190         free(tmp, M_TEMP);
5190
5191     if (seg)
5192         free(seg, M_TEMP);
5193
5194     if (tmp)
5195         free(tmp, M_TEMP);
5195
5196     if (seg)
5197         free(seg, M_TEMP);
5198
5199     if (tmp)
5200         free(tmp, M_TEMP);
5200
5201     if (seg)
5202         free(seg, M_TEMP);
5203
5204     if (tmp)
5205         free(tmp, M_TEMP);
5205
5206     if (seg)
5207         free(seg, M_TEMP);
5208
5209     if (tmp)
5210         free(tmp, M_TEMP);
5210
5211     if (seg)
5212         free(seg, M_TEMP);
5213
5214     if (tmp)
5215         free(tmp, M_TEMP);
5215
5216     if (seg)
5217         free(seg, M_TEMP);
5218
5219     if (tmp)
5220         free(tmp, M_TEMP);
5220
5221     if (seg)
5222         free(seg, M_TEMP);
5223
5224     if (tmp)
5225         free(tmp, M_TEMP);
5225
5226     if (seg)
5227         free(seg, M_TEMP);
5228
5229     if (tmp)
5230         free(tmp, M_TEMP);
5230
5231     if (seg)
5232         free(seg, M_TEMP);
5233
5234     if (tmp)
5235         free(tmp, M_TEMP);
5235
5236     if (seg)
5237         free(seg, M_TEMP);
5238
5239     if (tmp)
5240         free(tmp, M_TEMP);
5240
5241     if (seg)
5242         free(seg, M_TEMP);
5243
5244     if (tmp)
5245         free(tmp, M_TEMP);
5245
5246     if (seg)
5247         free(seg, M_TEMP);
5248
5249     if (tmp)
5250         free(tmp, M_TEMP);
5250
5251     if (seg)
5252         free(seg, M_TEMP);
5253
5254     if (tmp)
5255         free(tmp, M_TEMP);
5255
5256     if (seg)
5257         free(seg, M_TEMP);
5258
5259     if (tmp)
5260         free(tmp, M_TEMP);
5260
5261     if (seg)
5262         free(seg, M_TEMP);
5263
5264     if (tmp)
5265         free(tmp, M_TEMP);
5265
5266     if (seg)
5267         free(seg, M_TEMP);
5268
5269     if (tmp)
5270         free(tmp, M_TEMP);
5270
5271     if (seg)
5272         free(seg, M_TEMP);
5273
5274     if (tmp)
5275         free(tmp, M_TEMP);
5275
5276     if (seg)
5277         free(seg, M_TEMP);
5278
5279     if (tmp)
5280         free(tmp, M_TEMP);
5280
5281     if (seg)
5282         free(seg, M_TEMP);
5283
5284     if (tmp)
5285         free(tmp, M_TEMP);
5285
5286     if (seg)
5287         free(seg, M_TEMP);
5288
5289     if (tmp)
5290         free(tmp, M_TEMP);
5290
5291     if (seg)
5292         free(seg, M_TEMP);
5293
5294     if (tmp)
5295         free(tmp, M_TEMP);
5295
5296     if (seg)
5297         free(seg, M_TEMP);
5298
5299     if (tmp)
5300         free(tmp, M_TEMP);
5300
5301     if (seg)
5302         free(seg, M_TEMP);
5303
5304     if (tmp)
5305         free(tmp, M_TEMP);
5305
5306     if (seg)
5307         free(seg, M_TEMP);
5308
5309     if (tmp)
5310         free(tmp, M_TEMP);
5310
5311     if (seg)
5312         free(seg, M_TEMP);
5313
5314     if (tmp)
5315         free(tmp, M_TEMP);
5315
5316     if (seg)
5317         free(seg, M_TEMP);
5318
5319     if (tmp)
5320         free(tmp, M_TEMP);
5320
5321     if (seg)
5322         free(seg, M_TEMP);
5323
5324     if (tmp)
5325         free(tmp, M_TEMP);
5325
5326     if (seg)
5327         free(seg, M_TEMP);
5328
5329     if (tmp)
5330         free(tmp, M_TEMP);
5330
5331     if (seg)
5332         free(seg, M_TEMP);
5333
5334     if (tmp)
5335         free(tmp, M_TEMP);
5335
5336     if (seg)
5337         free(seg, M_TEMP);
5338
5339     if (tmp)
5340         free(tmp, M_TEMP);
5340
5341     if (seg)
5342         free(seg, M_TEMP);
5343
5344     if (tmp)
5345         free(tmp, M_TEMP);
5345
5346     if (seg)
5347         free(seg, M_TEMP);
5348
5349     if (tmp)
5350         free(tmp, M_TEMP);
5350
5351     if (seg)
5352         free(seg, M_TEMP);
5353
5354     if (tmp)
5355         free(tmp, M_TEMP);
5355
5356     if (seg)
5357         free(seg, M_TEMP);
5358
5359     if (tmp)
5360         free(tmp, M_TEMP);
5360
5361     if (seg)
5362         free(seg, M_TEMP);
5363
5364     if (tmp)
5365         free(tmp, M_TEMP);
5365
5366     if (seg)
5367         free(seg, M_TEMP);
5368
5369     if (tmp)
5370         free(tmp, M_TEMP);
5370
5371     if (seg)
5372         free(seg, M_TEMP);
5373
5374     if (tmp)
5375         free(tmp, M_TEMP);
5375
5376     if (seg)
5377         free(seg, M_TEMP);
5378
5379     if (tmp)
5380         free(tmp, M_TEMP);
5380
5381     if (seg)
5382         free(seg, M_TEMP);
5383
5384     if (tmp)
5385         free(tmp, M_TEMP);
5385
5386     if (seg)
5387         free(seg, M_TEMP);
5388
5389     if (tmp)
5390         free(tmp, M_TEMP);
5390
5391     if (seg)
5392         free(seg, M_TEMP);
5393
5394     if (tmp)
5395         free(tmp, M_TEMP);
5395
5396     if (seg)
5397         free(seg, M_TEMP);
5398
5399     if (tmp)
5400         free(tmp, M_TEMP);
5400
5401     if (seg)
5402         free(seg, M_TEMP);
5403
5404     if (tmp)
5405         free(tmp, M_TEMP);
5405
5406     if (seg)
5407         free(seg, M_TEMP);
5408
5409     if (tmp)
5410         free(tmp, M_TEMP);
5410
5411     if (seg)
5412         free(seg, M_TEMP);
5413
5414     if (tmp)
5415         free(tmp, M_TEMP);
5415
5416     if (seg)
5417         free(seg, M_TEMP);
5418
5419     if (tmp)
5420         free(tmp, M_TEMP);
5420
5421     if (seg)
5422         free(seg, M_TEMP);
5423
5424     if (tmp)
5425         free(tmp, M_TEMP);
5425
5426     if (seg)
5427         free(seg, M_TEMP);
5428
5429     if (tmp)
5430         free(tmp, M_TEMP);
5430
5431     if (seg)
5432         free(seg, M_TEMP);
5433
5434     if (tmp)
5435         free(tmp, M_TEMP);
5435
5436     if (seg)
5437         free(seg, M_TEMP);
5438
5439     if (tmp)
5440         free(tmp, M_TEMP);
5440
5441     if (seg)
5442         free(seg, M_TEMP);
5443
5444     if (tmp)
5445         free(tmp, M_TEMP);
5445
5446     if (seg)
5447         free(seg, M_TEMP);
5448
5449     if (tmp)
5450         free(tmp, M_TEMP);
5450
5451     if (seg)
5452         free(seg, M_TEMP);
5453
5454     if (tmp)
5455         free(tmp, M_TEMP);
5455
5456     if (seg)
5457         free(seg, M_TEMP);
5458
5459     if (tmp)
5460         free(tmp, M_TEMP);
5460
5461     if (seg)
5462         free(seg, M_TEMP);
5463
5464     if (tmp)
5465         free(tmp, M_TEMP);
5465
5466     if (seg)
5467         free(seg, M_TEMP);

```

```

4250         mp->pr_mflags |= MA_EXEC;
4251         if (segop_gettype(seg, saddr) & MAP_SHARED)
4252             if (SEGOP_GETTYPE(seg, saddr) & MAP_SHARED)
4253                 mp->pr_mflags |= MA_SHARED;
4254         if (segop_gettype(seg, saddr) & MAP_NORESERVE)
4255             if (SEGOP_GETTYPE(seg, saddr) & MAP_NORESERVE)
4256                 mp->pr_mflags |= MA_NORESERVE;
4257         if (seg->s_ops == &segpt_shmops ||
4258             (seg->s_ops == &segvn_ops &&
4259             (segop_getvp(seg, saddr, &vp) != 0 ||
4260              (SEGOP_GETVP(seg, saddr, &vp) != 0 ||
4261               vp == NULL)))
4262                 mp->pr_mflags |= MA_ANON;
4263         if (seg == prkseg)
4264             mp->pr_mflags |= MA_BREAK;
4265         else if (seg == stkseg)
4266             mp->pr_mflags |= MA_STACK;
4267         if (seg->s_ops == &segpt_shmops)
4268             mp->pr_mflags |= MA_ISM | MA_SHM;

4269         mp->pr_pagesize = PAGESIZE;
4270         if (psz == -1) {
4271             mp->pr_hatpagesize = 0;
4272         } else {
4273             mp->pr_hatpagesize = psz;
4274         }
4275
4276         /*
4277          * Manufacture a filename for the "object" dir.
4278          */
4279         mp->pr_dev = PRNODEV32;
4280         vattr.va_mask = AT_FSID|AT_NODEID;
4281         if (seg->s_ops == &segvn_ops &&
4282             segop_getvp(seg, saddr, &vp) == 0 &&
4283             SEGOP_GETVP(seg, saddr, &vp) == 0 &&
4284             vp != NULL && vp->v_type == VREG &&
4285             VOP_GETATTR(vp, &vattr, 0, CRED(),
4286                         NULL) == 0) {
4287             (void) cmpldev(&mp->pr_dev,
4288                            vattr.va_fsid);
4289             mp->pr_ino = vattr.va_nodeid;
4290             if (vp == p->p_exec)
4291                 (void) strcpy(mp->pr_mapname,
4292                               "a.out");
4293             else
4294                 pr_object_name(mp->pr_mapname,
4295                               vp, &vattr);
4296         }
4297
4298         /*
4299          * Get the SysV shared memory id, if any.
4300          */
4301         if ((mp->pr_mflags & MA_SHARED) &&
4302             p->p_segacct && (mp->pr_shmid = shmgetid(p,
4303                 seg->s_base)) != SHMID_NONE) {
4304             if (mp->pr_shmid == SHMID_FREE)
4305                 mp->pr_shmid = -1;
4306
4307             mp->pr_mflags |= MA_SHM;
4308         } else {
4309             mp->pr_shmid = -1;
4310         }
4311
4312         npages = ((uintptr_t)(naddr - saddr)) >>
4313             PAGESHIFT;
4314         parr = kmalloc_zalloc(npages, KM_SLEEP);

```

```
4313     (void) segop_incore(seg, saddr, naddr - saddr,
4314                           parr);
4315     SEGOP_INCORE(seg, saddr, naddr - saddr, parr);
4316
4317     for (pagenum = 0; pagenum < npages; pagenum++) {
4318         if (parr[pagenum] & SEG_PAGE_INCORE)
4319             mp->pr_rss++;
4320         if (parr[pagenum] & SEG_PAGE_ANON)
4321             mp->pr_anon++;
4322         if (parr[pagenum] & SEG_PAGE_LOCKED)
4323             mp->pr_locked++;
4324     }
4325     kmem_free(parr, npages);
4326 }
4327 ASSERT(tmp == NULL);
4328 } while ((seg = AS_SEGNEXT(as, seg)) != NULL);
4329
4330 return (0);
4331 }
```

unchanged portion omitted

[new/usr/src/uts/common/fs/proc/prvnops.c](#)

```

*****142974 Tue Nov 24 09:34:45 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 *dnpn = 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(dnpn->pr_type == PR_OBJECTDIR);

3676     pnp = prgetnode(dp, PR_OBJECT);

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

3689     /*
3690      * We drop p_lock before grabbing the address space
3691      * in order to avoid a deadlock with the clock thread.
3692      * The process will not disappear and its address
3693      * space 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"
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 &&
3712             SEGOP_GETVP(seg, seg->s_base, &vp) == 0 &&
3713             vp != NULL && vp->v_type == VREG &&
3714             VOP_GETATTR(vp, &vattr, 0, CRED(), NULL,
3715                         name[64];
3716
3717             if (vp == p->p_exec) /* "a.out"
3718                 continue;
3719             pr_object_name(name, vp, &vattr);
3720             if (strcmp(name, comp) == 0)
3721                 goto out;

```

1

```

new/usr/src/uts/common/fs/proc/prvnops.c

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(dnpn);

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 *dnpn = 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(dnpn->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 */

```

new/usr/src/uts/common/fs/proc/prvnops.c

```

4087             type = NAME_UNKNOWN;
4088             break;
4089         }
4090     }
4091
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     }
4101
4102     /*
4103      * Grab the necessary data from the process
4104      */
4105     if (prlock(dppnp, ZNO) != 0)
4106         return (NULL);
4107     p = dppnp->pr_common->prc_proc;
4108
4109     fip = P_INFO(p);
4110
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(dppnp);
4124                 return (NULL);
4125             }
4126     }
4127     mutex_exit(&p->p_lock);
4128
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);
4140
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 {

```

1

```

new/usr/src/uts/common/fs/proc/prvnops.c

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)
4163                         SEGOP_GETVP(seg, seg->s_base, &vp)
4164                         == 0 &&
4165                         vp != NULL && vp->v_type == VREG &&
4166                         VOP_GETATTR(vp, &vattr, 0, CRED(), NULL) == 0) {
4167                         char name[64];
4168
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             }
4179
4180             if (seg == NULL) {
4181                 vp = NULL;
4182             } else {
4183                 VN_HOLD(vp);
4184                 type = NAME_OBJECT;
4185             }
4186
4187             AS_LOCK_EXIT(as, &as->a_lock);
4188         }
4189     }

4190     switch (type) {
4191     case NAME_FD:
4192         mutex_enter(&fip->fi_lock);
4193         if (fd < fip->fi_nfiles) {
4194             UF_ENTER(ufp, fip, fd);
4195             if (ufp->uf_file != NULL) {
4196                 vp = ufp->uf_file->f_vnode;
4197                 VN_HOLD(vp);
4198             }
4199             UF_EXIT(ufp);
4200         }
4201         mutex_exit(&fip->fi_lock);
4202         idx = fd + 4;
4203         break;
4204     case NAME_ROOT:
4205         idx = 2;
4206         break;
4207     case NAME_CWD:
4208         idx = 3;
4209         break;
4210     case NAME_OBJECT:
4211         /* Nothing to do */
4212         break;
4213     case NAME_UNKNOWN:
4214         /* Nothing to do */
4215         break;
4216     }

```

```

4218     mutex_enter(&p->p_lock);
4219     prunlock(dpnp);
4220
4221     if (vp != NULL) {
4222         pnp = prgetnode(dp, PR_PATH);
4223
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     }
4234
4235     return (vp);
4236 }


---


unchanged_portion_omitted_
4237
4238 static void
4239 rebuild_objdir(struct as *as)
4240 {
4241     struct seg *seg;
4242     vnode_t *vp;
4243     vattr_t vattr;
4244     vnode_t **dir;
4245     ulong_t nalloc;
4246     ulong_t nentries;
4247     int i, j;
4248     ulong_t nold, nnew;
4249
4250     ASSERT(AS_WRITE_HELD(as, &as->a_lock));
4251
4252     if (as->a_updatedir == 0 && as->a_objectdir != NULL)
4253         return;
4254     as->a_updatedir = 0;
4255
4256     if ((nalloc = avl_numnodes(&as->a_segtree)) == 0 ||
4257         (seg = AS_SEGFIRST(as)) == NULL) /* can't happen? */
4258         return;
4259
4260     /*
4261      * Allocate space for the new object directory.
4262      * (This is usually about two times too many entries.)
4263      */
4264     nalloc = (nalloc + 0xf) & ~0xf; /* multiple of 16 */
4265     dir = kmem_zalloc(nalloc * sizeof (vnode_t *), KM_SLEEP);
4266
4267     /* fill in the new directory with desired entries */
4268     nentries = 0;
4269     do {
4270         vatr.va_mask = AT_FSID|AT_NODEID;
4271         if (seg->s_ops == &segvn_ops &&
4272             SEGOP_GETVP(seg, seg->s_base, &vp) == 0 &&
4273             SEGOP_GETVP(seg, seg->s_base, &vp) == 0 &&
4274             vp != NULL && vp->v_type == VREG &&
4275             VOP_GETATTR(vp, &vattr, 0, CRED(), NULL) == 0) {
4276                 for (i = 0; i < nentries; i++)
4277                     if (vp == dir[i])
4278                         break;
4279                 if (i == nentries) {
4280                     ASSERT(nentries < nalloc);
4281                     dir[nentries++] = vp;
4282                 }
4283             }
4284         }
4285     }

```

```

4873             }
4874         } while ((seg = AS_SEGNEXT(as, seg)) != NULL);
4875
4876         if (as->a_objectdir == NULL) { /* first time */
4877             as->a_objectdir = dir;
4878             as->a_sizedir = nalloc;
4879             return;
4880         }
4881
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         }
4902
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         }
4918
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_

```

```
new/usr/src/uts/common/io/mem.c
```

```
1
```

```
*****  
23668 Tue Nov 24 09:34:45 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;  
290  
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;  
294     i = (seg != NULL)? SECOP_CAPABLE(seg, S_CAPABILITY_NOMINFLT) : 0;  
295     AS_LOCK_EXIT(as, &as->a_lock);  
296     return (i);  
297 }  
unchanged_portion_omitted
```

```
new/usr/src/uts/common/os/dumpsubr.c
```

```
1
```

```
*****  
80155 Tue Nov 24 09:34:46 2015  
new/usr/src/uts/common/os/dumpsubr.c  
patch lower-case-segops  
*****  
_____ unchanged_portion_omitted_
```

```
1401 /*  
1402  * Dump the <as, va, pfn> information for a given address space.  
1403  * segop_dump() will call dump_addpage() for each page in the segment.  
1403  * SEGOP_DUMP() will call dump_addpage() for each page in the segment.  
1404 */  
1405 static void  
1406 dump_as(struct as *as)  
1407 {  
1408     struct seg *seg;  
1409  
1410     AS_LOCK_ENTER(as, &as->a_lock, RW_READER);  
1411     for (seg = AS_SEGFIRST(as); seg; seg = AS_SEGNEXT(as, seg)) {  
1412         if (seg->s_as != as)  
1413             break;  
1414         if (seg->s_ops == NULL)  
1415             continue;  
1416         segop_dump(seg);  
1416         SEGOP_DUMP(seg);  
1417     }  
1418     AS_LOCK_EXIT(as, &as->a_lock);  
1419  
1420     if (seg != NULL)  
1421         cmn_err(CE_WARN, "invalid segment %p in address space %p",  
1422                 (void *)seg, (void *)as);  
1423 }  
_____ unchanged_portion_omitted_
```

```
*****
52333 Tue Nov 24 09:34:46 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;
1187
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
1204
1205
1206
1207
1208
1209
1210
1211
1212
1213
1214
1215
1216
1217
1218
1219
1220
1221
1222
1223
1224
1225
1226
1227
1228
1229
1230
1231
1232
1233
1234
1235
1236
1237
1238
1239
1240
1241
1242
1243
1244
1245
1246
1247
1248
1249
1250
1251
1252
1253
1254
1255
1256
1257
1258
1259
1260
1261
1262
1263
1264
1265
1266
1267
1268
1269
1270
1271
1272
1273
1274
1275
1276
1277
1278
1279
1280
1281
1282
1283
1284
1285
1286
1287
1288
1289
1290
1291
1292
1293
1294
1295
1296
1297
1298
1299
1300
1301
1302
1303
1304
1305
1306
1307
1308
1309
1310
1311
1312
1313
1314
1315
1316
1317
1318
1319
1320
1321
1322
1323
1324
1325
1326
1327
1328
1329
1330
1331
1332
1333
1334
1335
1336
1337
1338
1339
1340
1341
1342
1343
1344
1345
1346
1347
1348
1349
1350
1351
1352
1353
1354
1355
1356
1357
1358
1359
1360
1361
1362
1363
1364
1365
1366
1367
1368
1369
1370
1371
1372
1373
1374
1375
1376
1377
1378
1379
1380
1381
1382
1383
1384
1385
1386
1387
1388
1389
1390
1391
1392
1393
1394
1395
1396
1397
1398
1399
1400
1401
1402
1403
1404
1405
1406
1407
1408
1409
1410
1411
1412
1413
1414
1415
1416
1417
1418
1419
1420
1421
1422
1423
1424
1425
1426
1427
1428
1429
1430
1431
1432
1433
1434
1435
1436
1437
1438
1439
1440
1441
1442
1443
1444
1445
1446
1447
1448
1449
1450
1451
1452
1453
1454
1455
1456
1457
1458
1459
1460
1461
1462
1463
1464
1465
1466
1467
1468
1469
1470
1471
1472
1473
1474
1475
1476
1477
1478
1479
1480
1481
1482
1483
1484
1485
1486
1487
1488
1489
1490
1491
1492
1493
1494
1495
1496
1497
1498
1499
1500
1501
1502
1503
1504
1505
1506
1507
1508
1509
1510
1511
1512
1513
1514
1515
1516
1517
1518
1519
1520
1521
1522
1523
1524
1525
1526
1527
1528
1529
1530
1531
1532
1533
1534
1535
1536
1537
1538
1539
1540
1541
1542
1543
1544
1545
1546
1547
1548
1549
1550
1551
1552
1553
1554
1555
1556
1557
1558
1559
1560
1561
1562
1563
1564
1565
1566
1567
1568
1569
1570
1571
1572
1573
1574
1575
1576
1577
1578
1579
1580
1581
1582
1583
1584
1585
1586
1587
1588
1589
1590
1591
1592
1593
1594
1595
1596
1597
1598
1599
1600
1601
1602
1603
1604
1605
1606
1607
1608
1609
1610
1611
1612
1613
1614
1615
1616
1617
1618
1619
1620
1621
1622
1623
1624
1625
1626
1627
1628
1629
1630
1631
1632
1633
1634
1635
1636
1637
1638
1639
1640
1641
1642
1643
1644
1645
1646
1647
1648
1649
1650
1651
1652
1653
1654
1655
1656
1657
1658
1659
1660
1661
1662
1663
1664
1665
1666
1667
1668
1669
1670
1671
1672
1673
1674
1675
1676
1677
1678
1679
1680
1681
1682
1683
1684
1685
1686
1687
1688
1689
1690
1691
1692
1693
1694
1695
1696
1697
1698
1699
1700
1701
1702
1703
1704
1705
1706
1707
1708
1709
1710
1711
1712
1713
1714
1715
1716
1717
1718
1719
1720
1721
1722
1723
1724
1725
1726
1727
1728
1729
1730
1731
1732
1733
1734
1735
1736
1737
1738
1739
1740
1741
1742
1743
1744
1745
1746
1747
1748
1749
1750
1751
1752
1753
1754
1755
1756
1757
1758
1759
1760
1761
1762
1763
1764
1765
1766
1767
1768
1769
1770
1771
1772
1773
1774
1775
1776
1777
1778
1779
1779
1780
1781
1782
1783
1784
1785
1786
1787
1788
1789
1790
1791
1792
1793
1794
1795
1796
1797
1798
1799
1800
1801
1802
1803
1804
1805
1806
1807
1808
1809
1809
1810
1811
1812
1813
1814
1815
1816
1817
1818
1819
1819
1820
1821
1822
1823
1824
1825
1826
1827
1828
1829
1829
1830
1831
1832
1833
1834
1835
1836
1837
1838
1839
1839
1840
1841
1842
1843
1844
1845
1846
1847
1848
1849
1849
1850
1851
1852
1853
1854
1855
1856
1857
1858
1859
1859
1860
1861
1862
1863
1864
1865
1866
1867
1868
1869
1869
1870
1871
1872
1873
1874
1875
1876
1877
1878
1879
1879
1880
1881
1882
1883
1884
1885
1886
1887
1888
1889
1889
1890
1891
1892
1893
1894
1895
1896
1897
1898
1899
1900
1901
1902
1903
1904
1905
1906
1907
1908
1909
1909
1910
1911
1912
1913
1914
1915
1916
1917
1918
1919
1919
1920
1921
1922
1923
1924
1925
1926
1927
1928
1929
1929
1930
1931
1932
1933
1934
1935
1936
1937
1938
1939
1939
1940
1941
1942
1943
1944
1945
1946
1947
1948
1949
1949
1950
1951
1952
1953
1954
1955
1956
1957
1958
1959
1959
1960
1961
1962
1963
1964
1965
1966
1967
1968
1969
1969
1970
1971
1972
1973
1974
1975
1976
1977
1978
1979
1979
1980
1981
1982
1983
1984
1985
1986
1987
1988
1989
1989
1990
1991
1992
1993
1994
1995
1996
1997
1998
1999
1999
2000
2001
2002
2003
2004
2005
2006
2007
2008
2009
2009
2010
2011
2012
2013
2014
2015
2016
2017
2018
2019
2019
2020
2021
2022
2023
2024
2025
2026
2027
2028
2029
2029
2030
2031
2032
2033
2034
2035
2036
2037
2038
2039
2039
2040
2041
2042
2043
2044
2045
2046
2047
2048
2049
2049
2050
2051
2052
2053
2054
2055
2056
2057
2058
2059
2059
2060
2061
2062
2063
2064
2065
2066
2067
2068
2069
2069
2070
2071
2072
2073
2074
2075
2076
2077
2078
2079
2079
2080
2081
2082
2083
2084
2085
2086
2087
2088
2089
2089
2090
2091
2092
2093
2094
2095
2096
2097
2098
2099
2099
2100
2101
2102
2103
2104
2105
2106
2107
2108
2109
2109
2110
2111
2112
2113
2114
2115
2116
2117
2118
2119
2119
2120
2121
2122
2123
2124
2125
2126
2127
2128
2129
2129
2130
2131
2132
2133
2134
2135
2136
2137
2138
2139
2139
2140
2141
2142
2143
2144
2145
2146
2147
2148
2149
2149
2150
2151
2152
2153
2154
2155
2156
2157
2158
2159
2159
2160
2161
2162
2163
2164
2165
2166
2167
2168
2169
2169
2170
2171
2172
2173
2174
2175
2176
2177
2178
2179
2179
2180
2181
2182
2183
2184
2185
2186
2187
2188
2189
2189
2190
2191
2192
2193
2194
2195
2196
2197
2198
2199
2199
2200
2201
2202
2203
2204
2205
2206
2207
2208
2209
2209
2210
2211
2212
2213
2214
2215
2216
2217
2218
2219
2219
2220
2221
2222
2223
2224
2225
2226
2227
2228
2229
2229
2230
2231
2232
2233
2234
2235
2236
2237
2238
2239
2239
2240
2241
2242
2243
2244
2245
2246
2247
2248
2249
2249
2250
2251
2252
2253
2254
2255
2256
2257
2258
2259
2259
2260
2261
2262
2263
2264
2265
2266
2267
2268
2269
2269
2270
2271
2272
2273
2274
2275
2276
2277
2278
2279
2279
2280
2281
2282
2283
2284
2285
2286
2287
2288
2289
2289
2290
2291
2292
2293
2294
2295
2296
2297
2298
2299
2299
2300
2301
2302
2303
2304
2305
2306
2307
2308
2309
2309
2310
2311
2312
2313
2314
2315
2316
2317
2318
2319
2319
2320
2321
2322
2323
2324
2325
2326
2327
2328
2329
2329
2330
2331
2332
2333
2334
2335
2336
2337
2338
2339
2339
2340
2341
2342
2343
2344
2345
2346
2347
2348
2349
2349
2350
2351
2352
2353
2354
2355
2356
2357
2358
2359
2359
2360
2361
2362
2363
2364
2365
2366
2367
2368
2369
2369
2370
2371
2372
2373
2374
2375
2376
2377
2378
2379
2379
2380
2381
2382
2383
2384
2385
2386
2387
2388
2389
2389
2390
2391
2392
2393
2394
2395
2396
2397
2398
2399
2399
2400
2401
2402
2403
2404
2405
2406
2407
2408
2409
2409
2410
2411
2412
2413
2414
2415
2416
2417
2418
2419
2419
2420
2421
2422
2423
2424
2425
2426
2427
2428
2429
2429
2430
2431
2432
2433
2434
2435
2436
2437
2438
2439
2439
2440
2441
2442
2443
2444
2445
2446
2447
2448
2449
2449
2450
2451
2452
2453
2454
2455
2456
2457
2458
2459
2459
2460
2461
2462
2463
2464
2465
2466
2467
2468
2469
2469
2470
2471
2472
2473
2474
2475
2476
2477
2478
2479
2479
2480
2481
2482
2483
2484
2485
2486
2487
2488
2489
2489
2490
2491
2492
2493
2494
2495
2496
2497
2498
2498
2499
2500
2501
2502
2503
2504
2505
2506
2507
2508
2509
2509
2510
2511
2512
2513
2514
2515
2516
2517
2518
2519
2519
2520
2521
2522
2523
2524
2525
2526
2527
2528
2529
2529
2530
2531
2532
2533
2534
2535
2536
2537
2538
2539
2539
2540
2541
2542
2543
2544
2545
2546
2547
2548
2549
2549
2550
2551
2552
2553
2554
2555
2556
2557
2558
2559
2559
2560
2561
2562
2563
2564
2565
2566
2567
2568
2569
2569
2570
2571
2572
2573
2574
2575
2576
2577
2578
2579
2579
2580
2581
2582
2583
2584
2585
2586
2587
2588
2589
2589
2590
2591
2592
2593
2594
2595
2596
2597
2598
2598
2599
2600
2601
2602
2603
2604
2605
2606
2607
2608
2609
2609
2610
2611
2612
2613
2614
2615
2616
2617
2618
2619
2619
2620
2621
2622
2623
2624
2625
2626
2627
2628
2629
2629
2630
2631
2632
2633
2634
2635
2636
2637
2638
2639
2639
2640
2641
2642
2643
2644
2645
2646
2647
2648
2649
2649
2650
2651
2652
2653
2654
2655
2656
2657
2658
2659
2659
2660
2661
2662
2663
2664
2665
2666
2667
2668
2669
2669
2670
2671
2672
2673
2674
2675
2676
2677
2678
2679
2679
2680
2681
2682
2683
2684
2685
2686
2687
2688
2689
2689
2690
2691
2692
2693
2694
2695
2696
2697
2698
2698
2699
2700
2701
2702
2703
2704
2705
2706
2707
2708
2709
2709
2710
2711
2712
2713
2714
2715
2716
2717
2718
2719
2719
2720
2721
2722
2723
2724
2725
2726
2727
2728
2729
2729
2730
2731
2732
2733
2734
2735
2736
2737
2738
2739
2739
2740
2741
2742
2743
2744
2745
2746
2747
2748
2749
2749
2750
2751
2752
2753
2754
2755
2756
2757
2758
2759
2759
2760
2761
2762
2763
2764
2765
2766
2767
2768
2769
2769
2770
2771
2772
2773
2774
2775
2776
2777
2778
2779
2779
2780
2781
2782
2783
2784
2785
2786
2787
2788
2789
2789
2790
2791
2792
2793
2794
2795
2796
2797
2798
2798
2799
2800
2801
2802
2803
2804
2805
2806
2807
2808
2809
2809
2810
2811
2812
2813
2814
2815
2816
2817
2818
2819
2819
2820
2821
2822
2823
2824
2825
2826
2827
2828
2829
2829
2830
2831
2832
2833
2834
2835
2836
2837
2838
2839
2839
2840
2841
2842
2843
2844
2845
2846
2847
2848
2849
2849
2850
2851
2852
2853
2854
2855
2856
2857
2858
2859
2859
2860
2861
2862
2863
2864
2865
2866
2867
2868
2869
2869
2870
2871
2872
2873
2874
2875
2876
2877
2878
2879
2879
2880
2881
2882
2883
2884
2885
2886
2887
2888
2889
2889
2890
2891
2892
2893
2894
2895
2896
2897
2898
2898
2899
2900
2901
2902
2903
2904
2905
2906
2907
2908
2909
2909
2910
2911
2912
2913
2914
2915
2916
2917
2918
2919
2919
2920
2921
2922
2923
2924
2925
2926
2927
2928
2929
2929
2930
2931
2932
2933
2934
2935
2936
2937
2938
2939
2939
2940
2941
2942
2943
2944
2945
2946
2947
2948
2949
2949
2950
2951
2952
2953
2954
2955
2956
2957
2958
2959
2959
2960
2961
2962
2963
2964
2965
2966
2967
2968
2969
2969
2970
2971
2972
2973
2974
2975
2976
2977
2978
2979
2979
2980
2981
2982
2983
2984
2985
2986
2987
2988
2989
2989
2990
2991
2992
2993
2994
2995
2996
2997
2998
2998
2999
2999
3000
3001
3002
3003
3004
3005
3006
3007
3008
3009
3009
3010
3011
3012
3013
3014
3015
3016
3017
3018
3019
3019
3020
3021
3022
3023
3024
3025
3026
3027
3028
3029
3029
3030
3031
3032
3033
3034
3035
3036
3037
3038
3039
3039
3040
3041
3042
3043
3044
3045
3046
3047
3048
3049
3049
3050
3051
3052
3053
3054
3055
3056
3057
3058
3059
3059
3060
3061
3062
3063
3064
3065
3066
3067
3068
3069
3069
3070
3071
3072
3073
3074
3075
3076
3077
3078
3079
3079
3080
3081
3082
3083
3084
3085
3086
3087
3088
3089
3089
3090
3091
3092
3093
3094
3095
3096
3097
3098
3098
3099
3099
3100
3101
3102
3103
3104
3105
3106
3107
3108
3109
3109
3110
3111
3112
3113
3114
3115
3116
3117
3118
3119
3119
3120
3121
3122
3123
3124
3125
3126
3127
3128
3129
3129
3130
3131
3132
3133
3134
3135
3136
3137
3138
3139
3139
3140
3141
3142
3143
31
```

```

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         */
1275
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         (void) segop_getprot(seg, (caddr_t)end,
1280                               zfoddiff - 1, &zprot);
1281     SEGOP_GETPROT(seg, (caddr_t)end, zfoddiff - 1,
1282                   &zprot);
1282     AS_LOCK_EXIT(as, &as->a_lock);
1283
1284     if (seg != NULL && (zprot & PROT_WRITE) == 0) {
1285         if (as_setprot(as, (caddr_t)end, zfoddiff - 1,
1286                        zprot | PROT_WRITE) == ENOMEM) {
1287             error = ENOMEM;
1288             goto bad;
1289         }
1290
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);
1308
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));
1323
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

```

```

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 bad:
1342     return (0);
1343
1344 }  

1345 unchanged portion omitted

```

```
*****
119448 Tue Nov 24 09:34:46 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 Tue Nov 24 09:34:46 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;

1453     /* No need to reserve swap space now since it will be reserved later */
1454     crargs.flags |= MAP_NORESERVE;
1455     as_rangelock(as);
1456     for (i = 0; i < loadable; i++) {
1457
1458         myaddr = start_addr + (size_t)mrp[i].mr_addr;
1459         mylen = mrp[i].mr_msize;
1460
1461         /* See if there is a hole in the as for this range */
1462         if (as_gap(as, mylen, &myaddr, &mylen, 0, NULL) == 0) {
1463             ASSERT(myaddr == start_addr + (size_t)mrp[i].mr_addr);
1464             ASSERT(mylen == mrp[i].mr_msize);
1465
1466 #ifdef DEBUG
1467             if (MR_GET_TYPE(mrp[i].mr_flags) == MR_PADDING) {
1468                 MOBJ_STAT_ADD(exec_padding);
1469             }
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) &
1492                   (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                 } else {
1509                     AS_LOCK_EXIT(as, &as->a_lock);
1510                     as_rangeunlock(as);
1511                     mmapobj_unmap_exec(mrp, i, start_addr);
1512                     MOBJ_STAT_ADD(exec_addr_in_use);
1513                     return (EADDRINUSE);
1514                 }
1515             }
1516         }
1517     }
1518     as_rangeunlock(as);
1519     return (0);
1520 }
_____ unchanged_portion_omitted_
```

```
*****
248841 Tue Nov 24 09:34:46 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;
8267
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 */
8273
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);
8282
8283     /* addr and len must be page-aligned */
8284     if (((uintptr_t)addr & PAGEOFFSET) != 0)
8285         return (EINVAL);
8286
8287     if ((len & PAGEOFFSET) != 0)
8288         return (EINVAL);
8289
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);
8303
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();
8310
8311     /* Allocate memory for the cookie */
8312     p = kmalloc(sizeof (struct ddi_umem_cookie), KM_SLEEP);
8313
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     }
8320
8321     /* Store procp in cookie for later iosetup/unlock */
8322     p->procp = (void *)procp;
8323
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;
8330
8331     /*
8332      * The size field is needed for lockmem accounting.
8333      */
8334     p->size = len;
8335     init_lockedmem_rctl_flag(p);
8336
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     }
8345
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     }
8354
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->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 &&
8382                  ((SEGOP_GETVP(seg, addr, &vp) == 0 &&
8383                  vp != NULL && vp->v_type == VREG) &&
8384                  (segop_gettime(seg, addr) & MAP_SHARED))) ||
8385                  (SEGOP_GETTYPE(seg, addr) & MAP_SHARED))) {
8386                 as_pageunlock(as, p->pparray,
8387                             addr, len, p->s_flags);
8388                 AS_LOCK_EXIT(as, &as->a_lock);
8389                 umem_decr_devlockmem(p);
8390                 kmem_free(p, sizeof (struct ddi_umem_cookie));
8391                 *cookie = (ddi_umem_cookie_t)NULL;
8392                 return (EFAULT);
8393             }
8394         }
8395         AS_LOCK_EXIT(as, &as->a_lock);
8396     }
8397
8398     /* Initialize the fields in the ddi_umem_cookie */
8399     p->cvaddr = addr;
8400     p->type = UMEM_LOCKED;
8401     if (driver_callback != NULL) {
8402         /* i_ddi_umem_unlock and umem_lock_undo may need the cookie */
8403         p->cook_refcnt = 2;
8404         p->callbacks = *ops_vector;
8405     } else {
8406         /* only i_ddi_umme_unlock needs the cookie */
8407         p->cook_refcnt = 1;
8408     }

```

```

8409     *cookie = (ddi_umem_cookie_t)p;
8410
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

```
*****
8540 Tue Nov 24 09:34:47 2015
new/usr/src/uts/common/os/urw.c
patch lower-case-segops
*****
1 /*
2 * CDDL HEADER START
3 *
4 * The contents of this file are subject to the terms of the
5 * Common Development and Distribution License (the "License").
6 * You may not use this file except in compliance with the License.
7 *
8 * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
9 * or http://www.opensolaris.org/os/licensing.
10 * See the License for the specific language governing permissions
11 * and limitations under the License.
12 *
13 * When distributing Covered Code, include this CDDL HEADER in each
14 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
15 * If applicable, add the following below this CDDL HEADER, with the
16 * fields enclosed by brackets "[]" replaced with your own identifying
17 * information: Portions Copyright [yyyy] [name of copyright owner]
18 *
19 * CDDL HEADER END
20 */
21 /*
22 * Copyright 2007 Sun Microsystems, Inc. All rights reserved.
23 * Use is subject to license terms.
24 */
25 /* Copyright (c) 1984, 1986, 1987, 1988, 1989 AT&T */
26 /* All Rights Reserved */
27
28 #pragma ident "%Z%%M% %I% %E% SMI"
29
30 #include <sys/atomic.h>
31 #include <sys/errno.h>
32 #include <sys/stat.h>
33 #include <sys/modctl.h>
34 #include <sys/conf.h>
35 #include <sys/sysm.h>
36 #include <sys/ddi.h>
37 #include <sys/sunddi.h>
38 #include <sys/cpuvar.h>
39 #include <sys/kmem.h>
40 #include <sys/strsubr.h>
41 #include <sys/sysmacros.h>
42 #include <sys/frame.h>
43 #include <sys/stack.h>
44 #include <sys/proc.h>
45 #include <sys/priv.h>
46 #include <sys/policy.h>
47 #include <sys/ontrap.h>
48 #include <sys/vmsystm.h>
49 #include <sys/prsystm.h>
50 #include <vm/as.h>
51 #include <vm/seg.h>
52 #include <vm/seg_dev.h>
53 #include <vm/seg_vn.h>
54 #include <vm/seg_spt.h>
55 #include <vm/seg_kmem.h>
56
57 extern struct seg_ops segdev_ops; /* needs a header file */
58 extern struct seg_ops segspt_shmops; /* needs a header file */
```

```
60 static int
61 page_valid(struct seg *seg, caddr_t addr)
62 {
63     struct segvn_data *svd;
64     vnode_t *vp;
65     vattr_t vattr;
66
67     /*
68      * Fail if the page doesn't map to a page in the underlying
69      * mapped file, if an underlying mapped file exists.
70      */
71     vattr.va_mask = AT_SIZE;
72     if (seg->s_ops == &segvn_ops &&
73         segop_getvp(seg, addr, &vp) == 0 &&
74         SEGOP_GETVP(seg, addr, &vp) == 0 &&
75         vp != NULL && vp->v_type == VREG &&
76         VOP_GETATTR(vp, &vattr, 0, CRED(), NULL) == 0) {
77         u_offset_t size = roundup(vattr.va_size, (u_offset_t)PAGESIZE);
78         u_offset_t offset = segop_getoffset(seg, addr);
79         u_offset_t offset = SEGOP_GETOFFSET(seg, addr);
80
81         if (offset >= size)
82             return (0);
83     }
84
85     /*
86      * Fail if this is an ISM shared segment and the address is
87      * not within the real size of the spt segment that backs it.
88      */
89     if (seg->s_ops == &segspt_shmops &&
90         addr >= seg->s_base + spt_realsize(seg))
91         return (0);
92
93     /*
94      * Fail if the segment is mapped from /dev/null.
95      * The key is that the mapping comes from segdev and the
96      * type is neither MAP_SHARED nor MAP_PRIVATE.
97      */
98     if (seg->s_ops == &segdev_ops &&
99         ((segop_gettype(seg, addr) & (MAP_SHARED | MAP_PRIVATE)) == 0) ||
100        ((SEGOP_GETTYPE(seg, addr) & (MAP_SHARED | MAP_PRIVATE)) == 0))
101         return (0);
102
103     /*
104      * Fail if the page is a MAP_NORESERVE page that has
105      * not actually materialized.
106      * We cheat by knowing that segvn is the only segment
107      * driver that supports MAP_NORESERVE.
108      */
109     if (seg->s_ops == &segvn_ops &&
110         (svd = (struct segvn_data *)seg->s_data) != NULL &&
111         (svd->vp == NULL || svd->vp->v_type != VREG) &&
112         (svd->flags & MAP_NORESERVE)) {
113         /*
114          * Guilty knowledge here. We know that
115          * segvn_incore returns more than just the
116          * low-order bit that indicates the page is
117          * actually in memory. If any bits are set,
118          * then there is backing store for the page.
119          */
120         char incore = 0;
121         (void) segop_incore(seg, addr, PAGESIZE, &incore);
122         (void) SEGOP_INCORE(seg, addr, PAGESIZE, &incore);
123         if (incore == 0)
124             return (0);
125     }
126 }
```

```

122         return (1);
123 }
unchanged_portion_omitted_
176 /*
177  * Perform I/O to a given process. This will return EIO if we detect
178  * corrupt memory and ENXIO if there is no such mapped address in the
179  * user process's address space.
180 */
181 static int
182 urw(proc_t *p, int writing, void *buf, size_t len, uintptr_t a)
183 {
184     caddr_t addr = (caddr_t)a;
185     caddr_t page;
186     caddr_t vaddr;
187     struct seg *seg;
188     int error = 0;
189     int err = 0;
190     uint_t prot;
191     uint_t prot_rw = writing ? PROT_WRITE : PROT_READ;
192     int protchanged;
193     on_trap_data_t otd;
194     int retrycnt;
195     struct as *as = p->p_as;
196     enum seg_rw rw;
197
198     /*
199      * Locate segment containing address of interest.
200      */
201     page = (caddr_t)(uintptr_t)((uintptr_t)addr & PAGEMASK);
202     retrycnt = 0;
203     AS_LOCK_ENTER(as, &as->a_lock, RW_WRITER);
204     retry:
205     if ((seg = as_segat(as, page)) == NULL ||
206         !page_valid(seg, page)) {
207         AS_LOCK_EXIT(as, &as->a_lock);
208         return (ENXIO);
209     }
210     (void) segop_getprot(seg, page, 0, &prot);
211     SEGOP_GETPROT(seg, page, 0, &prot);
212
213     protchanged = 0;
214     if ((prot & prot_rw) == 0) {
215         protchanged = 1;
216         err = segop_setprot(seg, page, PAGESIZE, prot | prot_rw);
217         err = SEGOP_SETPROT(seg, page, PAGESIZE, prot | prot_rw);
218
219         if (err == IE_RETRY) {
220             protchanged = 0;
221             ASSERT(retrycnt == 0);
222             retrycnt++;
223             goto retry;
224         }
225         if (err != 0) {
226             AS_LOCK_EXIT(as, &as->a_lock);
227             return (ENXIO);
228         }
229
230     /* segvn may do a copy-on-write for F_SOFTLOCK/S_READ case to break
231     * sharing to avoid a copy on write of a softlocked page by another
232     * thread. But since we locked the address space as a writer no other
233     * thread can cause a copy on write. S_READ_NOCOW is passed as the
234     * access type to tell segvn that it's ok not to do a copy-on-write
235 
```

```

236             * for this SOFTLOCK fault.
237             */
238             if (writing)
239                 rw = S_WRITE;
240             else if (seg->s_ops == &segvn_ops)
241                 rw = S_READ_NOCOW;
242             else
243                 rw = S_READ;
244
245             if (segop_fault(as->a_hat, seg, page, PAGESIZE, F_SOFTLOCK, rw)) {
246                 if (SEGOP_FAULT(as->a_hat, seg, page, PAGESIZE, F_SOFTLOCK, rw)) {
247                     if (protchanged)
248                         (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);
254
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);
259
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);
265
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                         }
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();
293
294                     /*
295                      * If we're writing to an executable page, we may need to synchronize
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);
300
301                     mapout(as, addr, vaddr, writing);
302
303                 }
304             }
305         }
306     }
307 
```

```
301     if (rw == S_READ_NOCOW)
302         rw = S_READ;
304     (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);
306     if (protchanged)
307         (void) segop_setprot(seg, page, PAGESIZE, prot);
309         (void) SEGOP_SETPROT(seg, page, PAGESIZE, prot);
309     AS_LOCK_EXIT(as, &as->a_lock);
311 }
312 }  
unchanged portion omitted
```

```
*****
14034 Tue Nov 24 09:34:47 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;
364
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     */
379     if (seg->s_ops != &segvn_ops) {
380         AS_LOCK_EXIT(as, &as->a_lock);
381         return (ENOTSUP);
382     } else if ((segop_gettype(seg, uaddr) & MAP_PRIVATE) == 0) {
383     } else if ((SEGOP_GETTYPE(seg, uaddr) & MAP_PRIVATE) == 0) {
384         AS_LOCK_EXIT(as, &as->a_lock);
385         return (ENOTSUP);
386     }
387     hat = as->a_hat;
388     size = total;
389     tryagain:
390     /*
391      * If (cow), hat_softlock will also change the usr protection to RO.
392      * This is the first step toward setting up cow. Before we
393      * bump up an_refcnt, we can't allow any cow-fault on this
394      * address. Otherwise segvn_fault will change the protection back
395      * to RW upon seeing an_refcnt == 1.
396      * The solution is to hold the writer lock on "as".
397
398     res = hat_softlock(hat, uaddr, &size, &ppp[0], cow ? HAT_COW : 0);
399     size = total - size;
400     *lenp += size;
401     size = size >> PAGESHIFT;
402     i = 0;
403     while (i < size) {
404         pp = ppp[i];
405         if (cow) {
406             kmutex_t *ahm;
407             /*
408              * Another solution is to hold SE_EXCL on pp, and
409              * disable PROT_WRITE. This also works for MAP_SHARED
410              * segment. The disadvantage is that it locks the
411              * page from being used by anybody else.
412
413             ahm = AH_MUTEX(pp->p_vnode, pp->p_offset);
414             mutex_enter(ahm);
415             *app = swap_anon(pp->p_vnode, pp->p_offset);
416             /*
417              * Since we are holding the as lock, this avoids a
418              * potential race with anon_decref. (segvn_unmap and
419              * segvn_free needs the as writer lock to do anon_free.)
420             */
421             #if 0
422                 if ((*app != NULL) {
423                     if ((*app)->an_refcnt == 0)
424                         /*
425                          * Consider the following scenario (unlikely
426                          * though):
427                          * 1. an_refcnt == 2
428                          * 2. we softlock the page.
429                          * 3. cow occurs on this addr. So a new ap,
430                          * page and mapping is established on addr.
431                          * 4. an_refcnt drops to 1 (segvn_faultpage
432                          * -> anon_decref(olddap))
433                          * 5. the last ref to ap also drops (from
434                          * another as). It ends up blocked inside
435                          * anon_decref trying to get page's excl lock.
436                          * 6. Later kcfree unlocks the page, call
437                          * anon_decref -> oops, ap is gone already.
438                          */
439
440             #endif
441             *app = NULL;
442         }
443     }
444 }
```

```

442 #endif
443         (*app)->an_refcnt++;
444         mutex_exit(ahm);
445     } else {
446         *app = NULL;
447     }
448     if (kaddr != (caddr_t)-1) {
449         if (pp != *cached_ppp) {
450             if (*cached_ppp == NULL)
451                 flags = HAT_LOAD_LOCK | HAT_NOSYNC |
452                         HAT_LOAD_NOCONSIST;
453             else
454                 flags = HAT_LOAD_REMAP |
455                         HAT_LOAD_NOCONSIST;
456             /*
457             * In order to cache the kernel mapping after
458             * the user page is unlocked, we call
459             * hat_devload instead of hat_memload so
460             * that the kernel mapping we set up here is
461             * "invisible" to the rest of the world. This
462             * is not very pretty. But as long as the
463             * caller bears the responsibility of keeping
464             * cache consistency, we should be ok -
465             * HAT_NOCONSIST will get us a uncached
466             * mapping on VAC. hat_softlock will flush
467             * a VAC_WRITEBACK cache. Therefore the kaddr
468             * doesn't have to be of the same vcolor as
469             * uaddr.
470             * The alternative is - change hat_devload
471             * to get a cached mapping. Allocate a kaddr
472             * with the same vcolor as uaddr. Then
473             * hat_softlock won't need to flush the VAC.
474             */
475         hat_devload(kas.a_hat, kaddr, PAGESIZE,
476                     page_pptonum(pp), PROT_READ, flags);
477         *cached_ppp = pp;
478     }
479     kaddr += PAGESIZE;
480 }
481 cached_ppp++;
482 app++;
483 ++i;
484 }
485 if (cow) {
486     AS_LOCK_EXIT(as, &as->a_lock);
487 }
488 if (first && res == FC_NOMAP) {
489     /*
490     * If the address is not mapped yet, we call as_fault to
491     * fault the pages in. We could've fallen back to copy and
492     * let it fault in the pages. But for a mapped file, we
493     * normally reference each page only once. For zero-copy to
494     * be of any use, we'd better fall in the page now and try
495     * again.
496     */
497     first = 0;
498     size = size << PAGESHIFT;
499     uaddr += size;
500     total -= size;
501     size = total;
502     res = as_fault(as->a_hat, as, uaddr, size, F_INVAL, S_READ);
503     if (cow)
504         AS_LOCK_ENTER(as, &as->a_lock, RW_WRITER);
505     goto tryagain;
506 }

```

```

508     switch (res) {
509         case FC_NOSUPPORT:
510             return (ENOTSUP);
511         case FC_PROT: /* Pretend we don't know about it. This will be */
512             /* * caught by the caller when uiomove fails. */
513         case FC_NOMAP:
514         case FC_OBJERR:
515             default:
516                 return (0);
517     }
518 }
```

unchanged portion omitted

```
new/usr/src/uts/common/os/watchpoint.c
```

```
*****
38864 Tue Nov 24 09:34:47 2015
new/usr/src/uts/common/os/watchpoint.c
patch lower-case-segops
*****
1 /*
2  * CDDL HEADER START
3 *
4  * The contents of this file are subject to the terms of the
5  * Common Development and Distribution License, Version 1.0 only
6  * (the "License"). You may not use this file except in compliance
7  * with the License.
8 *
9 * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
10 * or http://www.opensolaris.org/os/licensing.
11 * See the License for the specific language governing permissions
12 * and limitations under the License.
13 *
14 * When distributing Covered Code, include this CDDL HEADER in each
15 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
16 * If applicable, add the following below this CDDL HEADER, with the
17 * fields enclosed by brackets "[]" replaced with your own identifying
18 * information: Portions Copyright [yyyy] [name of copyright owner]
19 *
20 * CDDL HEADER END
21 */
22 /*
23 * Copyright 2005 Sun Microsystems, Inc. All rights reserved.
24 * Use is subject to license terms.
25 */

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

27 #include <sys/types.h>
28 #include <sys/t_lock.h>
29 #include <sys/param.h>
30 #include <sys/cred.h>
31 #include <sys/debug.h>
32 #include <sys/inline.h>
33 #include <sys/kmem.h>
34 #include <sys/proc.h>
35 #include <sys/regset.h>
36 #include <sys/sysmacros.h>
37 #include <sys/sysm.h>
38 #include <sys/prsystm.h>
39 #include <sys/buf.h>
40 #include <sys/signal.h>
41 #include <sys/user.h>
42 #include <sys/cpuvar.h>

44 #include <sys/fault.h>
45 #include <sys/syscall.h>
46 #include <sys/procfs.h>
47 #include <sys/cmn_err.h>
48 #include <sys/stack.h>
49 #include <sys/watchpoint.h>
50 #include <sys/copyops.h>
51 #include <sys/schedctl.h>

53 #include <sys/mman.h>
54 #include <vm/as.h>
55 #include <vm/seg.h>

57 /*
58 * Copy ops vector for watchpoints.
59 */
```

```
1
```

```
new/usr/src/uts/common/os/watchpoint.c
```

```
60 static int      watch_copyin(const void *, void *, size_t);
61 static int      watch_xcopyin(const void *, void *, size_t);
62 static int      watch_copyout(const void *, void *, size_t);
63 static int      watch_xcopyout(const void *, void *, size_t);
64 static int      watch_copyinstr(const char *, char *, size_t, size_t *);
65 static int      watch_copyoutstr(const char *, char *, size_t, size_t *);
66 static int      watch_fuword8(const void *, uint8_t *);
67 static int      watch_fuword16(const void *, uint16_t *);
68 static int      watch_fuword32(const void *, uint32_t *);
69 static int      watch_suword8(void *, uint8_t);
70 static int      watch_suword16(void *, uint16_t);
71 static int      watch_suword32(void *, uint32_t);
72 static int      watch_physio(int (*)(struct buf *, struct buf *,
73        dev_t, int, void (*)(struct buf *, struct uio *));
74 #ifdef _LP64
75 static int      watch_fuword64(const void *, uint64_t *);
76 static int      watch_suword64(void *, uint64_t);
77#endif

79 struct copyops watch_copyops = {
80     watch_copyin,
81     watch_xcopyin,
82     watch_copyout,
83     watch_xcopyout,
84     watch_copyinstr,
85     watch_copyoutstr,
86     watch_fuword8,
87     watch_fuword16,
88     watch_fuword32,
89 #ifdef _LP64
90     watch_fuword64,
91 #else
92     NULL,
93 #endif
94     watch_suword8,
95     watch_suword16,
96     watch_suword32,
97 #ifdef _LP64
98     watch_suword64,
99 #else
100    NULL,
101 #endif
102    watch_physio
103};

 unchanged_portion_omitted

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

156 /*
157 * Common code for pr_mappage() and pr_unmappage().
158 */
159 static int
160 pr_do_mappage(caddr_t addr, size_t size, int mapin, enum seg_rw rw, int kernel)
161 {
162     proc_t *p = curproc;
163     struct as *as = p->p_as;
164     char *eaddr = addr + size;
165     int prot_rw = rw_to_prot(rw);
166     int xrw = rw_to_index(rw);
167     int rv = 0;
168     struct watched_page *pwp;
169     struct watched_page tpw;
170     avl_index_t where;
```

```
2
```

```

171     uint_t prot;
173     ASSERT(as != &kas);
175 startover:
176     ASSERT(rv == 0);
177     if (avl_numnodes(&as->a_wpage) == 0)
178         return (0);
179
180     /*
181      * as->a_wpage can only be changed while the process is totally stopped.
182      * Don't grab p_lock here. Holding p_lock while grabbing the address
183      * space lock leads to deadlocks with the clock thread. Note that if an
184      * as_fault() is servicing a fault to a watched page on behalf of an
185      * XHAT provider, watchpoint will be temporarily cleared (and wp_prot
186      * will be set to wp_oprot). Since this is done while holding as writer
187      * lock, we need to grab as lock (reader lock is good enough).
188      *
189      * p_maplock prevents simultaneous execution of this function. Under
190      * normal circumstances, holdwatch() will stop all other threads, so the
191      * lock isn't really needed. But there may be multiple threads within
192      * stop() when SWATCHOK is set, so we need to handle multiple threads
193      * at once. See holdwatch() for the details of this dance.
194 */
195
196 mutex_enter(&p->p_maplock);
197 AS_LOCK_ENTER(as, &as->a_lock, RW_READER);
198
199 tpw.wp_vaddr = (caddr_t)((uintptr_t)addr & (uintptr_t)PAGEMASK);
200 if ((pwp = avl_find(&as->a_wpage, &tpw, &where)) == NULL)
201     pwp = avl_nearest(&as->a_wpage, where, AVL_AFTER);
202
203 for (; pwp != NULL && pwp->wp_vaddr < eaddr;
204     pwp = AVL_NEXT(&as->a_wpage, pwp)) {
205
206     /*
207      * If the requested protection has not been
208      * removed, we need not remap this page.
209      */
210     prot = pwp->wp_prot;
211     if (kernel || (prot & PROT_USER))
212         if (prot & prot_rw)
213             continue;
214
215     /*
216      * If the requested access does not exist in the page's
217      * original protections, we need not remap this page.
218      * If the page does not exist yet, we can't test it.
219      */
220     if ((prot = pwp->wp_oprot) != 0) {
221         if (!kernel || (prot & PROT_USER)))
222             continue;
223         if (!(prot & prot_rw))
224             continue;
225     }
226
227     if (mapin) {
228
229         /*
230          * Before mapping the page in, ensure that
231          * all other lwp's are held in the kernel.
232          */
233         if (p->p_mapcnt == 0) {
234
235             /*
236              * Release as lock while in holdwatch()
237              * in case other threads need to grab it.
238              */
239         AS_LOCK_EXIT(as, &as->a_lock);
240     }
241 }
242
243 mutex_exit(&p->p_maplock);
244 if (holdwatch() != 0) {
245
246     /*
247      * We stopped in holdwatch().
248      * Start all over again because the
249      * watched page list may have changed.
250      */
251     goto startover;
252 }
253
254 mutex_enter(&p->p_maplock);
255 AS_LOCK_ENTER(as, &as->a_lock, RW_READER);
256
257 p->p_mapcnt++;
258
259 addr = pwp->wp_vaddr;
260 rv++;
261
262 prot = pwp->wp_prot;
263 if (mapin) {
264
265     if (kernel)
266         pwp->wp_kmap[xrw]++;
267     else
268         pwp->wp_umap[xrw]++;
269     pwp->wp_flags |= WP_NOWATCH;
270     if (pwp->wp_kmap[X] + pwp->wp_umap[X])
271         /* cannot have exec-only protection */
272         prot |= PROT_READ|PROT_EXEC;
273     if (pwp->wp_kmap[R] + pwp->wp_umap[R])
274         prot |= PROT_READ;
275     if (pwp->wp_kmap[W] + pwp->wp_umap[W])
276         /* cannot have write-only protection */
277         prot |= PROT_READ|PROT_WRITE;
278
279 #if 0 /* damned broken mmu feature! */
280     if (sum(pwp->wp_umap) == 0)
281         prot &= ~PROT_USER;
282
283 #endif
284 } else {
285     ASSERT(pwp->wp_flags & WP_NOWATCH);
286     if (kernel) {
287         ASSERT(pwp->wp_kmap[xrw] != 0);
288         --pwp->wp_kmap[xrw];
289     } else {
290         ASSERT(pwp->wp_umap[xrw] != 0);
291         --pwp->wp_umap[xrw];
292     }
293     if (sum(pwp->wp_kmap) + sum(pwp->wp_umap) == 0)
294         pwp->wp_flags &= ~WP_NOWATCH;
295     else {
296         if (pwp->wp_kmap[X] + pwp->wp_umap[X])
297             /* cannot have exec-only protection */
298             prot |= PROT_READ|PROT_EXEC;
299         if (pwp->wp_kmap[R] + pwp->wp_umap[R])
300             prot |= PROT_READ;
301         if (pwp->wp_kmap[W] + pwp->wp_umap[W])
302             /* cannot have write-only protection */
303             prot |= PROT_READ|PROT_WRITE;
304
305 #if 0 /* damned broken mmu feature! */
306     if (sum(pwp->wp_umap) == 0)
307         prot &= ~PROT_USER;
308
309 #endif
310     }
311 }
312
313 if (pwp->wp_oprot != 0) { /* if page exists */
314
315     /*
316      * If the page exists, we must update its
317      * oprot value to match what was requested.
318      */
319     if (prot != pwp->wp_oprot)
320         pwp->wp_oprot = prot;
321
322     /*
323      * If the page exists, we must update its
324      * flags value to match what was requested.
325      */
326     if (pwp->wp_flags != prot)
327         pwp->wp_flags = prot;
328
329     /*
330      * If the page exists, we must update its
331      * kmap and umap values to match what was requested.
332      */
333     if (pwp->wp_kmap[X] != prot)
334         pwp->wp_kmap[X] = prot;
335     if (pwp->wp_umap[X] != prot)
336         pwp->wp_umap[X] = prot;
337
338     /*
339      * If the page exists, we must update its
340      * flags value to match what was requested.
341      */
342     if (pwp->wp_flags != prot)
343         pwp->wp_flags = prot;
344
345     /*
346      * If the page exists, we must update its
347      * kmap and umap values to match what was requested.
348      */
349     if (pwp->wp_kmap[R] != prot)
350         pwp->wp_kmap[R] = prot;
351     if (pwp->wp_umap[R] != prot)
352         pwp->wp_umap[R] = prot;
353
354     /*
355      * If the page exists, we must update its
356      * flags value to match what was requested.
357      */
358     if (pwp->wp_flags != prot)
359         pwp->wp_flags = prot;
360
361     /*
362      * If the page exists, we must update its
363      * kmap and umap values to match what was requested.
364      */
365     if (pwp->wp_kmap[W] != prot)
366         pwp->wp_kmap[W] = prot;
367     if (pwp->wp_umap[W] != prot)
368         pwp->wp_umap[W] = prot;
369
370     /*
371      * If the page exists, we must update its
372      * flags value to match what was requested.
373      */
374     if (pwp->wp_flags != prot)
375         pwp->wp_flags = prot;
376
377     /*
378      * If the page exists, we must update its
379      * kmap and umap values to match what was requested.
380      */
381     if (pwp->wp_kmap[X] != prot)
382         pwp->wp_kmap[X] = prot;
383     if (pwp->wp_umap[X] != prot)
384         pwp->wp_umap[X] = prot;
385
386     /*
387      * If the page exists, we must update its
388      * flags value to match what was requested.
389      */
390     if (pwp->wp_flags != prot)
391         pwp->wp_flags = prot;
392
393     /*
394      * If the page exists, we must update its
395      * kmap and umap values to match what was requested.
396      */
397     if (pwp->wp_kmap[R] != prot)
398         pwp->wp_kmap[R] = prot;
399     if (pwp->wp_umap[R] != prot)
400         pwp->wp_umap[R] = prot;
401
402     /*
403      * If the page exists, we must update its
404      * flags value to match what was requested.
405      */
406     if (pwp->wp_flags != prot)
407         pwp->wp_flags = prot;
408
409     /*
410      * If the page exists, we must update its
411      * kmap and umap values to match what was requested.
412      */
413     if (pwp->wp_kmap[W] != prot)
414         pwp->wp_kmap[W] = prot;
415     if (pwp->wp_umap[W] != prot)
416         pwp->wp_umap[W] = prot;
417
418     /*
419      * If the page exists, we must update its
420      * flags value to match what was requested.
421      */
422     if (pwp->wp_flags != prot)
423         pwp->wp_flags = prot;
424
425     /*
426      * If the page exists, we must update its
427      * kmap and umap values to match what was requested.
428      */
429     if (pwp->wp_kmap[X] != prot)
430         pwp->wp_kmap[X] = prot;
431     if (pwp->wp_umap[X] != prot)
432         pwp->wp_umap[X] = prot;
433
434     /*
435      * If the page exists, we must update its
436      * flags value to match what was requested.
437      */
438     if (pwp->wp_flags != prot)
439         pwp->wp_flags = prot;
440
441     /*
442      * If the page exists, we must update its
443      * kmap and umap values to match what was requested.
444      */
445     if (pwp->wp_kmap[R] != prot)
446         pwp->wp_kmap[R] = prot;
447     if (pwp->wp_umap[R] != prot)
448         pwp->wp_umap[R] = prot;
449
450     /*
451      * If the page exists, we must update its
452      * flags value to match what was requested.
453      */
454     if (pwp->wp_flags != prot)
455         pwp->wp_flags = prot;
456
457     /*
458      * If the page exists, we must update its
459      * kmap and umap values to match what was requested.
460      */
461     if (pwp->wp_kmap[W] != prot)
462         pwp->wp_kmap[W] = prot;
463     if (pwp->wp_umap[W] != prot)
464         pwp->wp_umap[W] = prot;
465
466     /*
467      * If the page exists, we must update its
468      * flags value to match what was requested.
469      */
470     if (pwp->wp_flags != prot)
471         pwp->wp_flags = prot;
472
473     /*
474      * If the page exists, we must update its
475      * kmap and umap values to match what was requested.
476      */
477     if (pwp->wp_kmap[X] != prot)
478         pwp->wp_kmap[X] = prot;
479     if (pwp->wp_umap[X] != prot)
480         pwp->wp_umap[X] = prot;
481
482     /*
483      * If the page exists, we must update its
484      * flags value to match what was requested.
485      */
486     if (pwp->wp_flags != prot)
487         pwp->wp_flags = prot;
488
489     /*
490      * If the page exists, we must update its
491      * kmap and umap values to match what was requested.
492      */
493     if (pwp->wp_kmap[R] != prot)
494         pwp->wp_kmap[R] = prot;
495     if (pwp->wp_umap[R] != prot)
496         pwp->wp_umap[R] = prot;
497
498     /*
499      * If the page exists, we must update its
500      * flags value to match what was requested.
501      */
502     if (pwp->wp_flags != prot)
503         pwp->wp_flags = prot;
504
505     /*
506      * If the page exists, we must update its
507      * kmap and umap values to match what was requested.
508      */
509     if (pwp->wp_kmap[W] != prot)
510         pwp->wp_kmap[W] = prot;
511     if (pwp->wp_umap[W] != prot)
512         pwp->wp_umap[W] = prot;
513
514     /*
515      * If the page exists, we must update its
516      * flags value to match what was requested.
517      */
518     if (pwp->wp_flags != prot)
519         pwp->wp_flags = prot;
520
521     /*
522      * If the page exists, we must update its
523      * kmap and umap values to match what was requested.
524      */
525     if (pwp->wp_kmap[X] != prot)
526         pwp->wp_kmap[X] = prot;
527     if (pwp->wp_umap[X] != prot)
528         pwp->wp_umap[X] = prot;
529
530     /*
531      * If the page exists, we must update its
532      * flags value to match what was requested.
533      */
534     if (pwp->wp_flags != prot)
535         pwp->wp_flags = prot;
536
537     /*
538      * If the page exists, we must update its
539      * kmap and umap values to match what was requested.
540      */
541     if (pwp->wp_kmap[R] != prot)
542         pwp->wp_kmap[R] = prot;
543     if (pwp->wp_umap[R] != prot)
544         pwp->wp_umap[R] = prot;
545
546     /*
547      * If the page exists, we must update its
548      * flags value to match what was requested.
549      */
550     if (pwp->wp_flags != prot)
551         pwp->wp_flags = prot;
552
553     /*
554      * If the page exists, we must update its
555      * kmap and umap values to match what was requested.
556      */
557     if (pwp->wp_kmap[W] != prot)
558         pwp->wp_kmap[W] = prot;
559     if (pwp->wp_umap[W] != prot)
560         pwp->wp_umap[W] = prot;
561
562     /*
563      * If the page exists, we must update its
564      * flags value to match what was requested.
565      */
566     if (pwp->wp_flags != prot)
567         pwp->wp_flags = prot;
568
569     /*
570      * If the page exists, we must update its
571      * kmap and umap values to match what was requested.
572      */
573     if (pwp->wp_kmap[X] != prot)
574         pwp->wp_kmap[X] = prot;
575     if (pwp->wp_umap[X] != prot)
576         pwp->wp_umap[X] = prot;
577
578     /*
579      * If the page exists, we must update its
580      * flags value to match what was requested.
581      */
582     if (pwp->wp_flags != prot)
583         pwp->wp_flags = prot;
584
585     /*
586      * If the page exists, we must update its
587      * kmap and umap values to match what was requested.
588      */
589     if (pwp->wp_kmap[R] != prot)
590         pwp->wp_kmap[R] = prot;
591     if (pwp->wp_umap[R] != prot)
592         pwp->wp_umap[R] = prot;
593
594     /*
595      * If the page exists, we must update its
596      * flags value to match what was requested.
597      */
598     if (pwp->wp_flags != prot)
599         pwp->wp_flags = prot;
600
601     /*
602      * If the page exists, we must update its
603      * kmap and umap values to match what was requested.
604      */
605     if (pwp->wp_kmap[W] != prot)
606         pwp->wp_kmap[W] = prot;
607     if (pwp->wp_umap[W] != prot)
608         pwp->wp_umap[W] = prot;
609
610     /*
611      * If the page exists, we must update its
612      * flags value to match what was requested.
613      */
614     if (pwp->wp_flags != prot)
615         pwp->wp_flags = prot;
616
617     /*
618      * If the page exists, we must update its
619      * kmap and umap values to match what was requested.
620      */
621     if (pwp->wp_kmap[X] != prot)
622         pwp->wp_kmap[X] = prot;
623     if (pwp->wp_umap[X] != prot)
624         pwp->wp_umap[X] = prot;
625
626     /*
627      * If the page exists, we must update its
628      * flags value to match what was requested.
629      */
630     if (pwp->wp_flags != prot)
631         pwp->wp_flags = prot;
632
633     /*
634      * If the page exists, we must update its
635      * kmap and umap values to match what was requested.
636      */
637     if (pwp->wp_kmap[R] != prot)
638         pwp->wp_kmap[R] = prot;
639     if (pwp->wp_umap[R] != prot)
640         pwp->wp_umap[R] = prot;
641
642     /*
643      * If the page exists, we must update its
644      * flags value to match what was requested.
645      */
646     if (pwp->wp_flags != prot)
647         pwp->wp_flags = prot;
648
649     /*
650      * If the page exists, we must update its
651      * kmap and umap values to match what was requested.
652      */
653     if (pwp->wp_kmap[W] != prot)
654         pwp->wp_kmap[W] = prot;
655     if (pwp->wp_umap[W] != prot)
656         pwp->wp_umap[W] = prot;
657
658     /*
659      * If the page exists, we must update its
660      * flags value to match what was requested.
661      */
662     if (pwp->wp_flags != prot)
663         pwp->wp_flags = prot;
664
665     /*
666      * If the page exists, we must update its
667      * kmap and umap values to match what was requested.
668      */
669     if (pwp->wp_kmap[X] != prot)
670         pwp->wp_kmap[X] = prot;
671     if (pwp->wp_umap[X] != prot)
672         pwp->wp_umap[X] = prot;
673
674     /*
675      * If the page exists, we must update its
676      * flags value to match what was requested.
677      */
678     if (pwp->wp_flags != prot)
679         pwp->wp_flags = prot;
680
681     /*
682      * If the page exists, we must update its
683      * kmap and umap values to match what was requested.
684      */
685     if (pwp->wp_kmap[R] != prot)
686         pwp->wp_kmap[R] = prot;
687     if (pwp->wp_umap[R] != prot)
688         pwp->wp_umap[R] = prot;
689
690     /*
691      * If the page exists, we must update its
692      * flags value to match what was requested.
693      */
694     if (pwp->wp_flags != prot)
695         pwp->wp_flags = prot;
696
697     /*
698      * If the page exists, we must update its
699      * kmap and umap values to match what was requested.
700      */
701     if (pwp->wp_kmap[W] != prot)
702         pwp->wp_kmap[W] = prot;
703     if (pwp->wp_umap[W] != prot)
704         pwp->wp_umap[W] = prot;
705
706     /*
707      * If the page exists, we must update its
708      * flags value to match what was requested.
709      */
710     if (pwp->wp_flags != prot)
711         pwp->wp_flags = prot;
712
713     /*
714      * If the page exists, we must update its
715      * kmap and umap values to match what was requested.
716      */
717     if (pwp->wp_kmap[X] != prot)
718         pwp->wp_kmap[X] = prot;
719     if (pwp->wp_umap[X] != prot)
720         pwp->wp_umap[X] = prot;
721
722     /*
723      * If the page exists, we must update its
724      * flags value to match what was requested.
725      */
726     if (pwp->wp_flags != prot)
727         pwp->wp_flags = prot;
728
729     /*
730      * If the page exists, we must update its
731      * kmap and umap values to match what was requested.
732      */
733     if (pwp->wp_kmap[R] != prot)
734         pwp->wp_kmap[R] = prot;
735     if (pwp->wp_umap[R] != prot)
736         pwp->wp_umap[R] = prot;
737
738     /*
739      * If the page exists, we must update its
740      * flags value to match what was requested.
741      */
742     if (pwp->wp_flags != prot)
743         pwp->wp_flags = prot;
744
745     /*
746      * If the page exists, we must update its
747      * kmap and umap values to match what was requested.
748      */
749     if (pwp->wp_kmap[W] != prot)
750         pwp->wp_kmap[W] = prot;
751     if (pwp->wp_umap[W] != prot)
752         pwp->wp_umap[W] = prot;
753
754     /*
755      * If the page exists, we must update its
756      * flags value to match what was requested.
757      */
758     if (pwp->wp_flags != prot)
759         pwp->wp_flags = prot;
760
761     /*
762      * If the page exists, we must update its
763      * kmap and umap values to match what was requested.
764      */
765     if (pwp->wp_kmap[X] != prot)
766         pwp->wp_kmap[X] = prot;
767     if (pwp->wp_umap[X] != prot)
768         pwp->wp_umap[X] = prot;
769
770     /*
771      * If the page exists, we must update its
772      * flags value to match what was requested.
773      */
774     if (pwp->wp_flags != prot)
775         pwp->wp_flags = prot;
776
777     /*
778      * If the page exists, we must update its
779      * kmap and umap values to match what was requested.
780      */
781     if (pwp->wp_kmap[R] != prot)
782         pwp->wp_kmap[R] = prot;
783     if (pwp->wp_umap[R] != prot)
784         pwp->wp_umap[R] = prot;
785
786     /*
787      * If the page exists, we must update its
788      * flags value to match what was requested.
789      */
790     if (pwp->wp_flags != prot)
791         pwp->wp_flags = prot;
792
793     /*
794      * If the page exists, we must update its
795      * kmap and umap values to match what was requested.
796      */
797     if (pwp->wp_kmap[W] != prot)
798         pwp->wp_kmap[W] = prot;
799     if (pwp->wp_umap[W] != prot)
800         pwp->wp_umap[W] = prot;
801
802     /*
803      * If the page exists, we must update its
804      * flags value to match what was requested.
805      */
806     if (pwp->wp_flags != prot)
807         pwp->wp_flags = prot;
808
809     /*
810      * If the page exists, we must update its
811      * kmap and umap values to match what was requested.
812      */
813     if (pwp->wp_kmap[X] != prot)
814         pwp->wp_kmap[X] = prot;
815     if (pwp->wp_umap[X] != prot)
816         pwp->wp_umap[X] = prot;
817
818     /*
819      * If the page exists, we must update its
820      * flags value to match what was requested.
821      */
822     if (pwp->wp_flags != prot)
823         pwp->wp_flags = prot;
824
825     /*
826      * If the page exists, we must update its
827      * kmap and umap values to match what was requested.
828      */
829     if (pwp->wp_kmap[R] != prot)
830         pwp->wp_kmap[R] = prot;
831     if (pwp->wp_umap[R] != prot)
832         pwp->wp_umap[R] = prot;
833
834     /*
835      * If the page exists, we must update its
836      * flags value to match what was requested.
837      */
838     if (pwp->wp_flags != prot)
839         pwp->wp_flags = prot;
840
841     /*
842      * If the page exists, we must update its
843      * kmap and umap values to match what was requested.
844      */
845     if (pwp->wp_kmap[W] != prot)
846         pwp->wp_kmap[W] = prot;
847     if (pwp->wp_umap[W] != prot)
848         pwp->wp_umap[W] = prot;
849
850     /*
851      * If the page exists, we must update its
852      * flags value to match what was requested.
853      */
854     if (pwp->wp_flags != prot)
855         pwp->wp_flags = prot;
856
857     /*
858      * If the page exists, we must update its
859      * kmap and umap values to match what was requested.
860      */
861     if (pwp->wp_kmap[X] != prot)
862         pwp->wp_kmap[X] = prot;
863     if (pwp->wp_umap[X] != prot)
864         pwp->wp_umap[X] = prot;
865
866     /*
867      * If the page exists, we must update its
868      * flags value to match what was requested.
869      */
870     if (pwp->wp_flags != prot)
871         pwp->wp_flags = prot;
872
873     /*
874      * If the page exists, we must update its
875      * kmap and umap values to match what was requested.
876      */
877     if (pwp->wp_kmap[R] != prot)
878         pwp->wp_kmap[R] = prot;
879     if (pwp->wp_umap[R] != prot)
880         pwp->wp_umap[R] = prot;
881
882     /*
883      * If the page exists, we must update its
884      * flags value to match what was requested.
885      */
886     if (pwp->wp_flags != prot)
887         pwp->wp_flags = prot;
888
889     /*
890      * If the page exists, we must update its
891      * kmap and umap values to match what was requested.
892      */
893     if (pwp->wp_kmap[W] != prot)
894         pwp->wp_kmap[W] = prot;
895     if (pwp->wp_umap[W] != prot)
896         pwp->wp_umap[W] = prot;
897
898     /*
899      * If the page exists, we must update its
900      * flags value to match what was requested.
901      */
902     if (pwp->wp_flags != prot)
903         pwp->wp_flags = prot;
904
905     /*
906      * If the page exists, we must update its
907      * kmap and umap values to match what was requested.
908      */
909     if (pwp->wp_kmap[X] != prot)
910         pwp->wp_kmap[X] = prot;
911     if (pwp->wp_umap[X] != prot)
912         pwp->wp_umap[X] = prot;
913
914     /*
915      * If the page exists, we must update its
916      * flags value to match what was requested.
917      */
918     if (pwp->wp_flags != prot)
919         pwp->wp_flags = prot;
920
921     /*
922      * If the page exists, we must update its
923      * kmap and umap values to match what was requested.
924      */
925     if (pwp->wp_kmap[R] != prot)
926         pwp->wp_kmap[R] = prot;
927     if (pwp->wp_umap[R] != prot)
928         pwp->wp_umap[R] = prot;
929
930     /*
931      * If the page exists, we must update its
932      * flags value to match what was requested.
933      */
934     if (pwp->wp_flags != prot)
935         pwp->wp_flags = prot;
936
937     /*
938      * If the page exists, we must update its
939      * kmap and umap values to match what was requested.
940      */
941     if (pwp->wp_kmap[W] != prot)
942         pwp->wp_kmap[W] = prot;
943     if (pwp->wp_umap[W] != prot)
944         pwp->wp_umap[W] = prot;
945
946     /*
947      * If the page exists, we must update its
948      * flags value to match what was requested.
949      */
950     if (pwp->wp_flags != prot)
951         pwp->wp_flags = prot;
952
953     /*
954      * If the page exists, we must update its
955      * kmap and umap values to match what was requested.
956      */
957     if (pwp->wp_kmap[X] != prot)
958         pwp->wp_kmap[X] = prot;
959     if (pwp->wp_umap[X] != prot)
960         pwp->wp_umap[X] = prot;
961
962     /*
963      * If the page exists, we must update its
964      * flags value to match what was requested.
965      */
966     if (pwp->wp_flags != prot)
967         pwp->wp_flags = prot;
968
969     /*
970      * If the page exists, we must update its
971      * kmap and umap values to match what was requested.
972      */
973     if (pwp->wp_kmap[R] != prot)
974         pwp->wp_kmap[R] = prot;
975     if (pwp->wp_umap[R] != prot)
976         pwp->wp_umap[R] = prot;
977
978     /*
979      * If the page exists, we must update its
980      * flags value to match what was requested.
981      */
982     if (pwp->wp_flags != prot)
983         pwp->wp_flags = prot;
984
985     /*
986      * If the page exists, we must update its
987      * kmap and umap values to match what was requested.
988      */
989     if (pwp->wp_kmap[W] != prot)
990         pwp->wp_kmap[W] = prot;
991     if (pwp->wp_umap[W] != prot)
992         pwp->wp_umap[W] = prot;
993
994     /*
995      * If the page exists, we must update its
996      * flags value to match what was requested.
997      */
998     if (pwp->wp_flags != prot)
999         pwp->wp_flags = prot;
1000
1001     /*
1002      * If the page exists, we must update its
1003      * kmap and umap values to match what was requested.
1004      */
1005     if (pwp->wp_kmap[X] != prot)
1006         pwp->wp_kmap[X] = prot;
1007     if (pwp->wp_umap[X] != prot)
1008         pwp->wp_umap[X] = prot;
1009
1010     /*
1011      * If the page exists, we must update its
1012      * flags value to match what was requested.
1013      */
1014     if (pwp->wp_flags != prot)
1015         pwp->wp_flags = prot;
1016
1017     /*
1018      * If the page exists, we must update its
1019      * kmap and umap values to match what was requested.
1020      */
1021     if (pwp->wp_kmap[R] != prot)
1022         pwp->wp_kmap[R] = prot;
1023     if (pwp->wp_umap[R] != prot)
1024         pwp->wp_umap[R] = prot;
1025
1026     /*
1027      * If the page exists, we must update its
1028      * flags value to match what was requested.
1029      */
1030     if (pwp->wp_flags != prot)
1031         pwp->wp_flags = prot;
1032
1033     /*
1034      * If the page exists, we must update its
1035      * kmap and umap values to match what was requested.
1036      */
1037     if (pwp->wp_kmap[W] != prot)
1038         pwp->wp_kmap[W] = prot;
1039     if (pwp->wp_umap[W] != prot)
1040         pwp->wp_umap[W] = prot;
1041
1042     /*
1043      * If the page exists, we must update its
1044      * flags value to match what was requested.
1045      */
1046     if (pwp->wp_flags != prot)
1047         pwp->wp_flags = prot;
1048
1049     /*
1050      * If the page exists, we must update its
1051      * kmap and umap values to match what was requested.
1052      */
1053     if (pwp->wp_kmap[X] != prot)
1054         pwp->wp_kmap[X] = prot;
1055     if (pwp->wp_umap[X] != prot)
1056         pwp->wp_umap[X] = prot;
1057
1058     /*
1059      * If the page exists, we must update its
1060      * flags value to match what was requested.
1061      */
1062     if (pwp->wp_flags != prot)
1063         pwp->wp_flags = prot;
1064
1065     /*
1066      * If the page exists, we must update its
1067      * kmap and umap values to match what was requested.
1068      */
1069     if (pwp->wp_kmap[R] != prot)
1070         pwp->wp_kmap[R] = prot;
1071     if (pwp->wp_umap[R] != prot)
1072         pwp->wp_umap[R] = prot;
1073
1074     /*
1075      * If the page exists, we must update its
1076      * flags value to match what was requested.
1077      */
1078     if (pwp->wp_flags != prot)
1079         pwp->wp_flags = prot;
1080
1081     /*
1082      * If the page exists, we must update its
1083      * kmap and umap values to match what was requested.
1084      */
1085     if (pwp->wp_kmap[W] != prot)
1086         pwp->wp_kmap[W] = prot;
1087     if (pwp->wp_umap[W] != prot)
1088         pwp->wp_umap[W] = prot;
1089
1090     /*
1091      * If the page exists, we must update its
1092      * flags value to match what was requested.
1093      */
1094     if (pwp->wp_flags != prot)
1095         pwp->wp_flags = prot;
1096
1097     /*
1098      * If the page exists, we must update its
1099      * kmap and umap values to match what was requested.
1100      */
1101     if (pwp->wp_k
```

```

303     struct seg *seg;
304     uint_t oprot;
305     int err, retrycnt = 0;
306
307     AS_LOCK_EXIT(as, &as->a_lock);
308     AS_LOCK_ENTER(as, &as->a_lock, RW_WRITER);
309
310     retry:
311     seg = as_segat(as, addr);
312     ASSERT(seg != NULL);
313     (void) segop_getprot(seg, addr, 0, &oprot);
314     SEGOP_GETPROT(seg, addr, 0, &oprot);
315     if (prot != oprot) {
316         err = segop_setprot(seg, addr, PAGESIZE, prot);
317         err = SEGOP_SETPROT(seg, addr, PAGESIZE, prot);
318         if (err == IE_RETRY) {
319             ASSERT(retrycnt == 0);
320             retrycnt++;
321             goto retry;
322         }
323     }
324     AS_LOCK_EXIT(as, &as->a_lock);
325 } else
326     AS_LOCK_EXIT(as, &as->a_lock);
327
328 /*
329  * When all pages are mapped back to their normal state,
330  * continue the other lwps.
331 */
332 if (!mapin) {
333     ASSERT(p->p_mapcnt > 0);
334     p->p_mapcnt--;
335     if (p->p_mapcnt == 0) {
336         mutex_exit(&p->p_maplock);
337         mutex_enter(&p->p_lock);
338         continue_lwps(p);
339         mutex_exit(&p->p_lock);
340         mutex_enter(&p->p_maplock);
341     }
342     AS_LOCK_ENTER(as, &as->a_lock, RW_READER);
343 }
344 AS_LOCK_EXIT(as, &as->a_lock);
345 mutex_exit(&p->p_maplock);
346
347 return (rv);
348 }

```

unchanged\_portion\_omitted\_

```

376 /*
377  * Function called by an lwp after it resumes from stop().
378 */
379 void
380 setallwatch(void)
381 {
382     proc_t *p = curproc;
383     struct as *as = curproc->p_as;
384     struct watched_page *pwp, *next;
385     struct seg *seg;
386     caddr_t vaddr;
387     uint_t prot;
388     int err, retrycnt;
389
390     if (p->p_wprot == NULL)
391         return;

```

```

393     ASSERT(MUTEX_NOT_HELD(&curproc->p_lock));
394
395     AS_LOCK_ENTER(as, &as->a_lock, RW_WRITER);
396
397     pwp = p->p_wprot;
398     while (pwp != NULL) {
399
400         vaddr = pwp->wp_vaddr;
401         retrycnt = 0;
402
403         retry:
404         ASSERT(pwp->wp_flags & WP_SETPROT);
405         if ((seg = as_segat(as, vaddr)) != NULL &&
406             !(pwp->wp_flags & WP_NOWATCH)) {
407             prot = pwp->wp_prot;
408             err = segop_setprot(seg, vaddr, PAGESIZE, prot);
409             err = SEGOP_SETPROT(seg, vaddr, PAGESIZE, prot);
410             if (err == IE_RETRY) {
411                 ASSERT(retrycnt == 0);
412                 retrycnt++;
413                 goto retry;
414             }
415         }
416
417         next = pwp->wp_list;
418
419         if (pwp->wp_read + pwp->wp_write + pwp->wp_exec == 0) {
420             /*
421              * No watched areas remain in this page.
422              * Free the watched_page structure.
423              */
424             avl_remove(&as->a_wpage, pwp);
425             kmem_free(pwp, sizeof (struct watched_page));
426         } else {
427             pwp->wp_flags &= ~WP_SETPROT;
428         }
429
430         pwp = next;
431         p->p_wprot = NULL;
432     }
433 }
```

unchanged\_portion\_omitted\_

```
*****
193225 Tue Nov 24 09:34:47 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;
5610
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)) {
5614
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)
5629             if (SEGOP_GETVP(seg, seg->s_base, &vp) != 0 || vp == NULL)
5630                 continue;
5631         if (!vn_can_change_zones(vp)) { /* bail on first match */
5632             allow = 0;
5633             break;
5634         }
5635     AS_LOCK_EXIT(as, &as->a_lock);
5636     return (allow);
5637 }
_____unchanged_portion_omitted_____
```

```
*****
4198 Tue Nov 24 09:34:47 2015
new/usr/src/uts/common/sys/watchpoint.h
patch lower-case-segops
*****
1 /*
2 * CDDL HEADER START
3 *
4 * The contents of this file are subject to the terms of the
5 * Common Development and Distribution License, Version 1.0 only
6 * (the "License"). You may not use this file except in compliance
7 * with the License.
8 *
9 * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
10 * or http://www.opensolaris.org/os/licensing.
11 * See the License for the specific language governing permissions
12 * and limitations under the License.
13 *
14 * When distributing Covered Code, include this CDDL HEADER in each
15 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
16 * If applicable, add the following below this CDDL HEADER, with the
17 * fields enclosed by brackets "[]" replaced with your own identifying
18 * information: Portions Copyright [yyyy] [name of copyright owner]
19 *
20 * CDDL HEADER END
21 */
22 /*
23 * Copyright 2004 Sun Microsystems, Inc. All rights reserved.
24 * Use is subject to license terms.
25 */

27 #ifndef _SYS_WATCHPOINT_H
28 #define _SYS_WATCHPOINT_H

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

30 #include <sys/types.h>
31 #include <vm/seg_enum.h>
32 #include <sys/copyops.h>
33 #include <sys/avl.h>

35 #ifdef __cplusplus
36 extern "C" {
37 #endif

39 /*
40 * Definitions for the VM implementation of watchpoints.
41 * See proc(4) and <sys/procfs.h> for definitions of the user interface.
42 */
43 /*
44 * Each process with watchpoints has a linked list of watched areas.
45 * The list is kept sorted by user-level virtual address.
46 */
47

48 typedef struct watched_area {
49     avl_node_t wa_link; /* link in AVL tree */
50     caddr_t wa_vaddr; /* virtual address of watched area */
51     caddr_t wa_eaddr; /* virtual address plus size */
52     ulong_t wa_flags; /* watch type flags (see <sys/procfs.h>) */
53 } watched_area_t;
54 unchanged_portion_omitted

55 /* wp_flags */
56 #define WP_NOWATCH 0x01 /* protections temporarily restored */
57 #define WP_SETPROT 0x02 /* segop_setprot() needed on this page */
58 #define WP_SETPROT 0x02 /* SEGOP_SETPROT() needed on this page */


```

```
77 #ifdef _KERNEL

79 /*
80 * These functions handle the necessary logic to perform the copy operation
81 * while ignoring watchpoints.
82 */
83 extern int copyin_nowatch(const void *, void *, size_t);
84 extern int copyout_nowatch(const void *, void *, size_t);
85 extern int fuword32_nowatch(const void *, uint32_t *);
86 extern int suword32_nowatch(void *, uint32_t);
87 #ifdef _LP64
88 extern int suword64_nowatch(void *, uint64_t);
89 extern int fuword64_nowatch(const void *, uint64_t *);
90#endif

92 /*
93 * Disable watchpoints for a given region of memory. When bracketed by these
94 * calls, functions can use copyops and ignore watchpoints.
95 */
96 extern int watch_disable_addr(const void *, size_t, enum seg_rw);
97 extern void watch_enable_addr(const void *, size_t, enum seg_rw);

99 /*
100 * Enable/Disable watchpoints for an entire thread.
101 */
102 extern void watch_enable(kthread_id_t);
103 extern void watch_disable(kthread_id_t);

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

118 extern struct copyops watch_copyops;

120 extern watched_area_t *pr_find_watched_area(struct proc *, watched_area_t *,
121                                              avl_index_t *);
122#endif

125 #ifdef __cplusplus
126 }
unchanged_portion_omitted
```

```
new/usr/src/uts/common/syscall/utssys.c
```

```
*****  
23713 Tue Nov 24 09:34:47 2015  
new/usr/src/uts/common/syscall/utssys.c  
patch lower-case-segops  
*****  
1 /*  
2  * CDDL HEADER START  
3 *  
4  * The contents of this file are subject to the terms of the  
5  * Common Development and Distribution License, Version 1.0 only  
6  * (the "License"). You may not use this file except in compliance  
7  * with the License.  
8 *  
9  * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE  
10 * or http://www.opensolaris.org/os/licensing.  
11 * See the License for the specific language governing permissions  
12 * and limitations under the License.  
13 *  
14 * When distributing Covered Code, include this CDDL HEADER in each  
15 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.  
16 * If applicable, add the following below this CDDL HEADER, with the  
17 * fields enclosed by brackets "[]" replaced with your own identifying  
18 * information: Portions Copyright [yyyy] [name of copyright owner]  
19 *  
20 * CDDL HEADER END  
21 */  
23 /*  
24  * Copyright 2004 Sun Microsystems, Inc. All rights reserved.  
25  * Use is subject to license terms.  
26 */  
28 /*      Copyright (c) 1984, 1986, 1987, 1988, 1989 AT&T */  
29 /*          All Rights Reserved */  
  
32 #pragma ident "%Z%%M% %I%     %E% SMI"  
  
32 #include <sys/param.h>  
33 #include <sys/inttypes.h>  
34 #include <sys/types.h>  
35 #include <sys/sysmacros.h>  
36 #include <sys/system.h>  
37 #include <sys/user.h>  
38 #include <sys/errno.h>  
39 #include <sys/vfs.h>  
40 #include <sys/vnode.h>  
41 #include <sys/file.h>  
42 #include <sys/proc.h>  
43 #include <sys/session.h>  
44 #include <sys/var.h>  
45 #include <sys/utsname.h>  
46 #include <sys/utssys.h>  
47 #include <sys/ustat.h>  
48 #include <sys/statvfs.h>  
49 #include <sys/kmem.h>  
50 #include <sys/debug.h>  
51 #include <sys pathname.h>  
52 #include <sys/modctl.h>  
53 #include <sys/fs/snode.h>  
54 #include <sys/sunldi_impl.h>  
55 #include <sys/ddi.h>  
56 #include <sys/sunddi.h>  
57 #include <sys/cmn_err.h>  
58 #include <sys/ddipropdefs.h>  
59 #include <sys/ddi_impldefs.h>
```

```
1
```

```
new/usr/src/uts/common/syscall/utssys.c  
*****  
60 #include <sys/modctl.h>  
61 #include <sys/flock.h>  
62 #include <sys/share.h>  
63 #include <vm/as.h>  
64 #include <vm/seg.h>  
65 #include <vm/seg_vn.h>  
66 #include <util/qsort.h>  
67 #include <sys/zone.h>  
69 /*  
70  * utssys()  
71  */  
72 static int           uts_fusers(char *, int, intptr_t);  
73 static int           _statvfs64_by_dev(dev_t, struct statvfs64 *);  
75 #if defined(_ILP32) || defined(_SYSCALL32_IMPL)  
77 static int utssys_uname32(caddr_t, rval_t *);  
78 static int utssys_ustat32(dev_t, struct ustat32 *);  
80 int64_t  
81 utssys32(void *buf, int arg, int type, void *outbp)  
82 {  
83     int error;  
84     rval_t rv;  
86     rv.r_vals = 0;  
88     switch (type) {  
89         case UTS_UNAME:  
90             /*  
91              * This is an obsolete way to get the utsname structure  
92              * (it only gives you the first 8 characters of each field!)  
93              * uname(2) is the preferred and better interface.  
94             */  
95             error = utssys_uname32(buf, &rv);  
96             break;  
97         case UTS_USTAT:  
98             error = utssys_ustat32(expldev((dev32_t)arg), buf);  
99             break;  
100        case UTS_FUSERS:  
101            error = uts_fusers(buf, arg, (intptr_t)outbp);  
102            break;  
103        default:  
104            error = EINVAL;  
105            break;  
106    }  
108    return (error == 0 ? rv.r_vals : (int64_t)set_errno(error));  
109 }  
_____  
111 static fu_data_t *  
112 dofusers(vnode_t *fvp, int flags)  
113 {  
114     fu_data_t      *fu_data;  
115     proc_t          *prp;  
116     vfs_t           *cvfsp;  
117     pid_t           npids, pidx, *pidlist;  
118     int              v_proc = v.v_proc;      /* max # of procs */  
119     int              pcnt = 0;  
120     int              contained = (flags & F_CONTAINED);  
121     int              nbmandonly = (flags & F_NBMANDLIST);  
122     int              dip_usage = (flags & F_DEVINFO);  
123     int              fvp_isdev = vn_matchops(fvp, spec_get vnodeops());  
124     zone_t *zone = curproc->p_zone;
```

```
2
```

```

325     int inglobal = INGLOBALZONE(curproc);
327
328     /* get a pointer to the file system containing this vnode */
329     cvfsp = fvp->v_vfsp;
330     ASSERT(cvfsp);
331
332     /* allocate the data structure to return our results in */
333     fu_data = kmem_alloc(fu_data_size(v_proc), KM_SLEEP);
334     fu_data->fud_user_max = v_proc;
335     fu_data->fud_user_count = 0;
336
337     /* get a snapshot of all the pids we're going to check out */
338     pidlist = kmem_alloc(v_proc * sizeof (pid_t), KM_SLEEP);
339     mutex_enter(&pidlock);
340     for (npids = 0, prp = practical; prp != NULL; prp = prp->p_next) {
341         if (inglobal || prp->p_zone == zone)
342             pidlist[npids++] = prp->p_pid;
343     }
344     mutex_exit(&pidlock);
345
346     /* grab each process and check its file usage */
347     for (pidx = 0; pidx < npids; pidx++) {
348         locklist_t    *llp = NULL;
349         uf_info_t     *fip;
350         vnode_t       *vp;
351         user_t        *up;
352         sess_t        *sp;
353         uid_t         uid;
354         pid_t          pid = pidlist[pidx];
355         int            i, use_flag = 0;
356
357         /*
358          * grab prp->p_lock using sprlock()
359          * if sprlock() fails the process does not exists anymore
360          */
361         prp = sprlock(pid);
362         if (prp == NULL)
363             continue;
364
365         /* get the processes credential info in case we need it */
366         mutex_enter(&prp->p_crlock);
367         uid = crgetruid(prp->p_cred);
368         mutex_exit(&prp->p_crlock);
369
370         /*
371          * it's safe to drop p_lock here because we
372          * called sprlock() before and it set the SPRLOCK
373          * flag for the process so it won't go away.
374          */
375         mutex_exit(&prp->p_lock);
376
377         /*
378          * now we want to walk a processes open file descriptors
379          * to do this we need to grab the fip->fi_lock. (you
380          * can't hold p_lock when grabbing the fip->fi_lock.)
381          */
382         fip = P_FINFO(prp);
383         mutex_enter(&fip->fi_lock);
384
385         /*
386          * Snapshot nbmand locks for pid
387          */
388         llp = flk_active_nbmand_locks(prp->p_pid);
389         for (i = 0; i < fip->fi_nfiles; i++) {
390             uf_entry_t     *ufp;
391             file_t         *fp;

```

```

392
393     UF_ENTER(ufp, fip, i);
394     if (((fp = ufp->uf_file) == NULL) ||
395         ((vp = fp->f vnode) == NULL)) {
396         UF_EXIT(ufp);
397         continue;
398     }
399
400     /*
401      * if the target file (fvp) is not a device
402      * and corresponds to the root of a filesystem
403      * (cvfsp), then check if it contains the file
404      * is use by this process (vp).
405      */
406     if (contained && (vp->v_vfsp == cvfsp))
407         use_flag |= F_OPEN;
408
409     /*
410      * if the target file (fvp) is not a device,
411      * then check if it matches the file in use
412      * by this process (vp).
413      */
414     if (!fvp_isdev && VN_CMP(fvp, vp))
415         use_flag |= F_OPEN;
416
417     /*
418      * if the target file (fvp) is a device,
419      * then check if the current file in use
420      * by this process (vp) maps to the same device
421      * minor node.
422      */
423     if (fvp_isdev &&
424         vn_matchops(vp, spec_getvnodeops()) &&
425         (fvp->v_rdev == vp->v_rdev))
426         use_flag |= F_OPEN;
427
428     /*
429      * if the target file (fvp) is a device,
430      * and we're checking for device instance
431      * usage, then check if the current file in use
432      * by this process (vp) maps to the same device
433      * instance.
434      */
435     if (dip_usage &&
436         vn_matchops(vp, spec_getvnodeops()) &&
437         (VTOCS(fvp)->s_dip == VTOCS(vp)->s_dip))
438         use_flag |= F_OPEN;
439
440     /*
441      * if the current file in use by this process (vp)
442      * doesn't match what we're looking for, move on
443      * to the next file in the process.
444      */
445     if ((use_flag & F_OPEN) == 0) {
446         UF_EXIT(ufp);
447         continue;
448     }
449
450     if (proc_has_nbmand_on_vp(vp, prp->p_pid, llp)) {
451         /*
452          * A nbmand found so we're done. */
453         use_flag |= F_NBM;
454         UF_EXIT(ufp);
455         break;
456     }
457     UF_EXIT(ufp);

```

```

457     if (llp)
458         flk_free_locklist(llp);
460
461     mutex_exit(&fip->fi_lock);
462
463     /*
464      * If nbmand usage tracking is desired and no nbmand was
465      * found for this process, then no need to do further
466      * usage tracking for this process.
467     */
468     if (nbmandonly && (!(use_flag & F_NBM))) {
469
470         /*
471          * grab the process lock again, clear the SPRLOCK
472          * flag, release the process, and continue.
473        */
474
475         mutex_enter(&prp->p_lock);
476         sprunlock(prp);
477         continue;
478     }
479
480     /*
481      * All other types of usage.
482      * For the next few checks we need to hold p_lock.
483    */
484
485     mutex_enter(&prp->p_lock);
486     up = PTOU(prp);
487     if (fvp_isdev) {
488
489         /*
490          * if the target file (fvp) is a device
491          * then check if it matches the processes tty
492          *
493          * we grab s_lock to protect ourselves against
494          * freectt() freeing the vnode out from under us.
495        */
496
497         sp = prp->p_sessp;
498         mutex_enter(&sp->s_lock);
499         vp = prp->p_sessp->s_vp;
500         if (vp != NULL) {
501             if (fvp->v_rdev == vp->v_rdev)
502                 use_flag |= F_TTY;
503
504             if (dip_usage &&
505                 (VTOCS(fvp)->s_dip == VTOCS(vp)->s_dip))
506                 use_flag |= F_TTY;
507         }
508
509         mutex_exit(&sp->s_lock);
510     } else {
511
512         /* check the processes current working directory */
513         if (up->u_cdir &&
514             (VN_CMP(fvp, up->u_cdir) ||
515              (contained && (up->u_cdir->v_vfsp == cvfsp))))
516             use_flag |= F_CDIR;
517
518         /* check the processes root directory */
519         if (up->u_rdir &&
520             (VN_CMP(fvp, up->u_rdir) ||
521              (contained && (up->u_rdir->v_vfsp == cvfsp))))
522             use_flag |= F_RDIR;
523
524         /* check the program text vnode */
525         if (prp->p_exec &&
526             (VN_CMP(fvp, prp->p_exec) ||
527              (contained && (prp->p_exec->v_vfsp == cvfsp))))
528             use_flag |= F_TEXT;
529     }

```

```

523
524
525     /* Now we can drop p_lock again */
526     mutex_exit(&prp->p_lock);
527
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;
536
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             /*
542              * if we can't get a backing vnode for this
543              * segment then skip it
544            */
545             vp = NULL;
546             if ((segop_getvp(seg, seg->s_base, &vp)) ||
547                 ((SEGOP_GETVP(seg, seg->s_base, &vp)) ||
548                  (vp == NULL)))
549                 continue;
550
551             /*
552              * if the target file (fvp) is not a device
553              * and corresponds to the root of a filesystem
554              * (cvfsp), then check if it contains the
555              * vnode backing this segment (vp).
556            */
557             if (contained && (vp->v_vfsp == cvfsp)) {
558                 use_flag |= F_MAP;
559                 break;
560             }
561
562             /*
563              * if the target file (fvp) is not a device,
564              * check if it matches the the vnode backing
565              * this segment (vp).
566            */
567             if (!fvp_isdev && VN_CMP(fvp, vp)) {
568                 use_flag |= F_MAP;
569                 break;
570             }
571
572             /*
573              * if the target file (fvp) isn't a device,
574              * or the the vnode backing this segment (vp)
575              * isn't a device then continue.
576            */
577             if (!fvp_isdev ||
578                 !vn_matchops(vp, spec_getvnodeops()))
579                 continue;
580
581             /*
582              * check if the vnode backing this segment
583              * (vp) maps to the same device minor node
584              * as the target device (fvp)
585            */
586             if (fvp->v_rdev == vp->v_rdev) {
587                 use_flag |= F_MAP;
588                 break;
589             }
590         }
591     }

```

```
588         * if we're checking for device instance
589         * usage, then check if the vnode backing
590         * this segment (vp) maps to the same device
591         * instance as the target device (fvp).
592     */
593     if (dip_usage &&
594         (VTOCS(fvp)->s_dip == VTOCS(vp)->s_dip)) {
595         use_flag |= F_MAP;
596         break;
597     }
598     AS_LOCK_EXIT(as, &as->a_lock);
599 }
600
601 if (use_flag) {
602     ASSERT(pcnt < fu_data->fud_user_max);
603     fu_data->fud_user[pcnt].fu_flags = use_flag;
604     fu_data->fud_user[pcnt].fu_pid = pid;
605     fu_data->fud_user[pcnt].fu_uid = uid;
606     pcnt++;
607 }
608
609 /*
610  * grab the process lock again, clear the SPRLOCK
611  * flag, release the process, and continue.
612  */
613 mutex_enter(&prp->p_lock);
614 sprunlock(prp);
615 }
616
617 kmem_free(pidlist, v_proc * sizeof (pid_t));
618
619 fu_data->fud_user_count = pcnt;
620 return (fu_data);
621 }
622 }
```

unchanged portion omitted

new/usr/src/uts/common/vm/seg.h

1

```
*****  
10256 Tue Nov 24 09:34:48 2015  
new/usr/src/uts/common/vm/seg.h  
patch lower-case-segops  
*****  
_____ unchanged_portion_omitted _____  
147 #ifdef _KERNEL  
  
149 /*  
150 * Generic segment operations  
151 */  
152 extern void seg_init(void);  
153 extern struct seg *seg_alloc(struct as *as, caddr_t base, size_t size);  
154 extern int seg_attach(struct as *as, caddr_t base, size_t size,  
155                      struct seg *seg);  
156 extern void seg_unmap(struct seg *seg);  
157 extern void seg_free(struct seg *seg);  
  
159 /*  
160 * functions for pagelock cache support  
161 */  
162 typedef int (*seg_preclaim_cbfunc_t)(void *, caddr_t, size_t,  
163                                     struct page **, enum seg_rw, int);  
  
165 extern struct page **seg_plookup(struct seg *seg, struct anon_map *amp,  
166                                   caddr_t addr, size_t len, enum seg_rw rw, uint_t flags);  
167 extern void seg_pinactive(struct seg *seg, struct anon_map *amp,  
168                           caddr_t addr, size_t len, struct page **pp, enum seg_rw rw,  
169                           uint_t flags, seg_preclaim_cbfunc_t callback);  
  
171 extern void seg_ppurge(struct seg *seg, struct anon_map *amp,  
172                         uint_t flags);  
173 extern void seg_ppurge_wiredpp(struct page **pp);  
  
175 extern int seg_pinsert_check(struct seg *seg, struct anon_map *amp,  
176                               caddr_t addr, size_t len, uint_t flags);  
177 extern int seg_pinsert(struct seg *seg, struct anon_map *amp,  
178                        caddr_t addr, size_t len, size_t wlen, struct page **pp, enum seg_rw rw,  
179                        uint_t flags, seg_preclaim_cbfunc_t callback);  
  
181 extern void seg_pasync_thread(void);  
182 extern void seg_pread(void);  
183 extern int seg_p_disable(void);  
184 extern void seg_p_enable(void);  
  
186 extern segadvstat_t segadvstat;  
  
188 /*  
189 * Flags for pagelock cache support.  
190 * Flags argument is passed as uint_t to pcache routines. upper 16 bits of  
191 * the flags argument are reserved for alignment page shift when SEGP_PSHIFT  
192 * is set.  
193 */  
194 #define SEGP_FORCE_WIRED      0x1      /* skip check against seg_pwindow */  
195 #define SEGP_AMP               0x2      /* anon map's pcache entry */  
196 #define SEGP_PSHIFT             0x4      /* addr pgsize shift for hash function */  
  
198 /*  
199 * Return values for seg_pinsert and seg_pinsert_check functions.  
200 */  
201 #define SEGP_SUCCESS           0        /* seg_pinsert() succeeded */  
202 #define SEGP_FAIL              1        /* seg_pinsert() failed */  
  
204 /* Page status bits for segop_incore */  
205 #define SEG_PAGE_INCORE        0x01    /* VA has a page backing it */
```

new/usr/src/uts/common/vm/seg.h

2

```
206 #define SEG_PAGE_LOCKED          0x02    /* VA has a page that is locked */  
207 #define SEG_PAGE_HASCOW          0x04    /* VA has a page with a copy-on-write */  
208 #define SEG_PAGE_SOFTLOCK         0x08    /* VA has a page with softlock held */  
209 #define SEG_PAGE_VNODEBACKED     0x10    /* Segment is backed by a vnode */  
210 #define SEG_PAGE_ANON            0x20    /* VA has an anonymous page */  
211 #define SEG_PAGE_VNODE           0x40    /* VA has a vnode page backing it */  
  
213 #define seg_page(seg, addr) \  
214     (((uintptr_t)((addr) - (seg)->s_base)) >> PAGESHIFT)  
  
216 #define seg_pages(seg) \  
217     (((uintptr_t)((seg)->s_size + PAGEOFFSET)) >> PAGESHIFT)  
  
219 #define IE_NOMEM                -1      /* internal to seg layer */  
220 #define IE_RETRY                 -2      /* internal to seg layer */  
221 #define IE_REATTACH              -3      /* internal to seg layer */  
  
223 /* Values for segop_inherit */  
223 /* Values for SEGP_INHERIT */  
224 #define SEGP_INH_ZERO            0x01  
  
226 int seg_inherit_notsup(struct seg *, caddr_t, size_t, uint_t);  
  
228 /* Delay/retry factors for seg_p_mem_config_pre_del */  
229 #define SEGP_PREDEL_DELAY_FACTOR 4  
230 /*  
231 * As a workaround to being unable to purge the pagelock  
232 * cache during a DR delete memory operation, we use  
233 * a stall threshold that is twice the maximum seen  
234 * during testing. This workaround will be removed  
235 * when a suitable fix is found.  
236 */  
237 #define SEGP_STALL_SECONDS       25  
238 #define SEGP_STALL_THRESHOLD \  
239     (SEGP_STALL_SECONDS * SEGP_PREDEL_DELAY_FACTOR)  
  
241 #ifdef VMDEBUG  
  
243 uint_t seg_page(struct seg *, caddr_t);  
244 uint_t seg_pages(struct seg *);  
  
246 #endif /* VMDEBUG */  
  
248 boolean_t seg_can_change_zones(struct seg *);  
249 size_t seg_swresv(struct seg *);  
  
251 /* segop wrappers */  
252 extern int segop_dup(struct seg *, struct seg *);  
253 extern int segop_unmap(struct seg *, caddr_t, size_t);  
254 extern void segop_free(struct seg *);  
255 extern faultcode_t segop_fault(struct hat *, struct seg *, caddr_t, size_t,  
256                                 enum fault_type, enum seg_rw);  
257 extern faultcode_t segop_faulta(struct seg *, caddr_t);  
258 extern int segop_setprot(struct seg *, caddr_t, size_t, uint_t);  
259 extern int segop_checkprot(struct seg *, caddr_t, size_t, uint_t);  
260 extern int segop_kluster(struct seg *, caddr_t, ssize_t);  
261 extern size_t segop_swapout(struct seg *);  
262 extern int segop_sync(struct seg *, caddr_t, size_t, int, uint_t);  
263 extern size_t segop_incore(struct seg *, caddr_t, size_t, char *);  
264 extern int segop_lockop(struct seg *, caddr_t, size_t, int, int, ulong_t *,  
265                         size_t);  
266 extern int segop_getprot(struct seg *, caddr_t, size_t, uint_t *);  
267 extern u_offset_t segop_getoffset(struct seg *, caddr_t);  
268 extern int segop_gettime(struct seg *, caddr_t);  
269 extern int segop_getvp(struct seg *, caddr_t, struct vnode **);  
270 extern int segop_advise(struct seg *, caddr_t, size_t, uint_t);
```

```
271 extern void segop_dump(struct seg *);  
272 extern int segop_pagelock(struct seg *, caddr_t, size_t, struct page ***,  
273     enum lock_type, enum seg_rw);  
274 extern int segop_setpagesize(struct seg *, caddr_t, size_t, uint_t);  
275 extern int segop_getmemid(struct seg *, caddr_t, memid_t *);  
276 extern struct lgrp_mem_policy_info *segop_getpolicy(struct seg *, caddr_t);  
277 extern int segop_capable(struct seg *, segcapability_t);  
278 extern int segop_inherit(struct seg *, caddr_t, size_t, uint_t);  
  
280 #endif /* _KERNEL */  
282 #ifdef __cplusplus  
283 }  
unchanged_portion_omitted_
```

new/usr/src/uts/common/vm/seg\_dev.c

```
*****
114107 Tue Nov 24 09:34:48 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;
365
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));
371
372     hat_map(seg->s_as->a_hat, seg->s_base, seg->s_size, HAT_MAP);
373
374     sdp = sdp_alloc();
375
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;
384
385     if (sdp->mapfunc == NULL)
386         sdp->devmap_data = dhp;
387     else
388         sdp->devmap_data = dhp = NULL;
389
390     sdp->hat_flags = a->hat_flags;
391     sdp->hat_attr = a->hat_attr;
392
393     /*
394      * Currently, hat_flags supports only HAT_LOAD_NOCONSIST
395      */
396     ASSERT(!(sdp->hat_flags & ~HAT_LOAD_NOCONSIST));
397
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);
405
406     seg->s_ops = &segdev_ops;
407     seg->s_data = sdp;
408
409     while (dhp != NULL) {
410         dhp->dh_seg = seg;
411         dhp = dhp->dh_next;
412     }
413 }
```

1

new/usr/src/uts/common/vm/seg\_dev.c

```
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);
424
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 |
435             if ((SEGOP_GETTYPE(seg, (seg)->s_base) & (MAP_SHARED |
436                 MAP_PRIVATE)) == 0) {
437                     seg->s_as->a_resvsize -= seg->s_size;
438                 }
439             }
440
441     }_____unchanged_portion_omitted_____
442 }
```

2

```
new/usr/src/uts/common/vm/seg_dev.h
```

```
1
```

```
*****  
4471 Tue Nov 24 09:34:48 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) & \  
122        (MAP_SHARED | MAP_PRIVATE)) == 0))  
121      ((SEGOP_GETTYPE(seg, (seg)->s_base) & (MAP_SHARED | MAP_PRIVATE)) == 0))  
  
124 extern void segdev_init(void);  
  
126 extern int segdev_create(struct seg *, void *);  
  
128 extern int segdev_copyto(struct seg *, caddr_t, const void *, void *, size_t);  
129 extern int segdev_copyfrom(struct seg *, caddr_t, const void *, void *, size_t);  
130 extern struct seg_ops segdev_ops;  
  
132 #endif /* __KERNEL */  
  
134 #ifdef __cplusplus  
135 }unchanged_portion_omitted
```

new/usr/src/uts/common/vm/seg\_kmem.c

```
*****
45463 Tue Nov 24 09:34:48 2015
new/usr/src/uts/common/vm/seg_kmem.c
patch lower-case-segops
*****
_____ unchanged_portion_omitted _____
437 #define SEGKMEM_BADOP(t)          (t(*)())segkmem_badop
439 /*ARGSUSED*/
440 static faultcode_t
441 segkmem_fault(struct hat *hat, struct seg *seg, caddr_t addr, size_t size,
442                 enum fault_type type, enum seg_rw rw)
443 {
444     pgcnt_t npages;
445     spgcnt_t pg;
446     page_t *pp;
447     struct vnode *vp = seg->s_data;
448
449     ASSERT(RW_READ_HELD(&seg->s_as->a_lock));
450
451     if (seg->s_as != &kas || size > seg->s_size ||
452         addr < seg->s_base || addr + size > seg->s_base + seg->s_size)
453         panic("segkmem_fault: bad args");
454
455     /*
456      * If it is one of segkp pages, call segkp_fault.
457      */
458     if (segkp_bitmap && seg == &kvseg &&
459         BT_TEST(segkp_bitmap, btop((uintptr_t)(addr - seg->s_base))))
460         return (segop_fault(hat, segkp, addr, size, type, rw));
461         return (SEGOP_FAULT(hat, segkp, addr, size, type, rw));
462
463     if (rw != S_READ && rw != S_WRITE && rw != S_OTHER)
464         return (FC_NOSUPPORT);
465
466     npages = btopr(size);
467
468     switch (type) {
469     case F_SOFTLOCK:           /* lock down already-loaded translations */
470         for (pg = 0; pg < npages; pg++) {
471             pp = page_lookup(vp, (u_offset_t)(uintptr_t)addr,
472                               SE_SHARED);
473             if (pp == NULL) {
474                 /*
475                  * Hmm, no page. Does a kernel mapping
476                  * exist for it?
477                 */
478                 if (!hat_probe(kas.a_hat, addr)) {
479                     addr -= PAGESIZE;
480                     while (--pg >= 0) {
481                         pp = page_find(vp, (u_offset_t)
482                                         (uintptr_t)addr);
483                         if (pp)
484                             page_unlock(pp);
485                         addr -= PAGESIZE;
486                     }
487                     return (FC_NOMAP);
488                 }
489             }
490             addr += PAGESIZE;
491         }
492         if (rw == S_OTHER)
493             hat_reserve(seg->s_as, addr, size);
494     case F_SOFTUNLOCK:
495     }
```

1

new/usr/src/uts/common/vm/seg\_kmem.c

```
*****
495         while (npages--) {
496             pp = page_find(vp, (u_offset_t)(uintptr_t)addr);
497             if (pp)
498                 page_unlock(pp);
499             addr += PAGESIZE;
500         }
501         return (0);
502     default:
503         return (FC_NOSUPPORT);
504     }
505     /*NOTREACHED*/
506 }
508 static int
509 segkmem_setprot(struct seg *seg, caddr_t addr, size_t size, uint_t prot)
510 {
511     ASSERT(RW_LOCK_HELD(&seg->s_as->a_lock));
512
513     if (seg->s_as != &kas || size > seg->s_size ||
514         addr < seg->s_base || addr + size > seg->s_base + seg->s_size)
515         panic("segkmem_setprot: bad args");
516
517     /*
518      * If it is one of segkp pages, call segkp.
519      */
520     if (segkp_bitmap && seg == &kvseg &&
521         BT_TEST(segkp_bitmap, btop((uintptr_t)(addr - seg->s_base))))
522         return (segop_setprot(segkp, addr, size, prot));
523         return (SEGOP_SETPROT(segkp, addr, size, prot));
524
525     if (prot == 0)
526         hat_unload(kas.a_hat, addr, size, HAT_UNLOAD);
527     else
528         hat_chgprot(kas.a_hat, addr, size, prot);
529 }
531 /*
532  * This is a dummy segkmem function overloaded to call segkp
533  * when segkp is under the heap.
534 */
535 /*ARGSUSED*/
536 static int
537 segkmem_checkprot(struct seg *seg, caddr_t addr, size_t size, uint_t prot)
538 {
539     ASSERT(RW_LOCK_HELD(&seg->s_as->a_lock));
540
541     if (seg->s_as != &kas)
542         segkmem_badop();
543
544     /*
545      * If it is one of segkp pages, call into segkp.
546      */
547     if (segkp_bitmap && seg == &kvseg &&
548         BT_TEST(segkp_bitmap, btop((uintptr_t)(addr - seg->s_base))))
549         return (segop_checkprot(segkp, addr, size, prot));
550         return (SEGOP_CHECKPROT(segkp, addr, size, prot));
551
552     segkmem_badop();
553 }
555 /*
556  * This is a dummy segkmem function overloaded to call segkp
557  * when segkp is under the heap.
558 */
```

2

```

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));
564
565     if (seg->s_as != &kas)
566         segkmem_badop();
567
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));
574         return (SEGOP_KLUSTER(segkp, addr, delta));
575
576     segkmem_badop();
577 }


---



unchanged_portion omitted


569 /*
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;
686
687     ASSERT(ppp != NULL);
688
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));
695         return (SEGOP_PAGELOCK(segkp, addr, len, ppp, type, rw));
696
697     npages = btopr(len);
698     nb = sizeof (page_t *) * npages;
699
700     if (type == L_PAGEUNLOCK) {
701         pplist = *ppp;
702         ASSERT(pplist != NULL);
703
704         for (pg = 0; pg < npages; pg++) {
705             pp = pplist[pg];
706             page_unlock(pp);
707         }
708         kmem_free(pplist, nb);
709     }
710
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     }
718
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     }
731
732     *ppp = pplist;
733     return (0);
734 }
735
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));
745
746     if (seg->s_as != &kas)
747         segkmem_badop();
748
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));
755         return (SEGOP_GETMEMID(segkp, addr, memidp));
756
757     segkmem_badop();
758 }


---



unchanged_portion omitted


```

new/usr/src/uts/common/vm/seg\_vn.c

1

```
*****  
286002 Tue Nov 24 09:34:48 2015  
new/usr/src/uts/common/vm/seg_vn.c  
patch lower-case-segops  
*****  
unchanged_portion_omitted  
  
6086 /*  
6087 * segvn_setpagesize is called via segop_setpagesize from as_setpagesize,  
6087 * segvn_setpagesize is called via SEGOP_SETPAGESIZE from as_setpagesize,  
6088 * to determine if the seg is capable of mapping the requested szc.  
6089 */  
6090 static int  
6091 segvn_setpagesize(struct seg *seg, caddr_t addr, size_t len, uint_t szc)  
6092 {  
6093     struct segvn_data *svd = (struct segvn_data *)seg->s_data;  
6094     struct segvn_data *nsvd;  
6095     struct anon_map *amp = svd->amp;  
6096     struct seg *nseg;  
6097     caddr_t eaddr = addr + len, a;  
6098     size_t pgsz = page_get_pagesize(szc);  
6099     pgcnt_t pgcnt = page_get_pagecnt(szc);  
6100     int err;  
6101     u_offset_t off = svd->offset + (uintptr_t)(addr - seg->s_base);  
  
6103     ASSERT(seg->s_as && AS_WRITE_HELD(seg->s_as, &seg->s_as->a_lock));  
6104     ASSERT(addr >= seg->s_base && eaddr <= seg->s_base + seg->s_size);  
  
6106     if (seg->s_szc == szc || segvn_lpg_disable != 0) {  
6107         return (0);  
6108     }  
  
6110     /*  
6111      * addr should always be pgsz aligned but eaddr may be misaligned if  
6112      * it's at the end of the segment.  
6113      *  
6114      * XXX we should assert this condition since as_setpagesize() logic  
6115      * guarantees it.  
6116      */  
6117     if (!IS_P2ALIGNED(addr, pgsz) ||  
6118         (!IS_P2ALIGNED(eaddr, pgsz) &&  
6119          eaddr != seg->s_base + seg->s_size)) {  
  
6121         segvn_setpgsz_align_err++;  
6122         return (EINVAL);  
6123     }  
  
6125     if (amp != NULL && svd->type == MAP_SHARED) {  
6126         ulong_t an_idx = svd->anon_index + seg_page(seg, addr);  
6127         if (!IS_P2ALIGNED(an_idx, pgcnt)) {  
  
6129             segvn_setpgsz_anon_align_err++;  
6130             return (EINVAL);  
6131         }  
6132     }  
  
6134     if ((svd->flags & MAP_NORESERVE) || seg->s_as == &kas ||  
6135         szc > segvn_maxpgszc) {  
6136         return (EINVAL);  
6137     }  
  
6139     /* paranoid check */  
6140     if (svd->vp != NULL &&  
6141         (IS_SWAPFSVP(svd->vp) || VN_ISKAS(svd->vp))) {  
6142         return (EINVAL);  
6143     }
```

new/usr/src/uts/common/vm/seg\_vn.c

2

```
6145     if (seg->s_szc == 0 && svd->vp != NULL &&  
6146         map_addr_vacalign_check(addr, off)) {  
6147             return (EINVAL);  
6148         }  
  
6150         /*  
6151          * Check that protections are the same within new page  
6152          * size boundaries.  
6153          */  
6154         if (svd->pageprot) {  
6155             for (a = addr; a < eaddr; a += pgsz) {  
6156                 if ((a + pgsz) > eaddr) {  
6157                     if (!sameprot(seg, a, eaddr - a)) {  
6158                         return (EINVAL);  
6159                     }  
6160                 } else {  
6161                     if (!sameprot(seg, a, pgsz)) {  
6162                         return (EINVAL);  
6163                     }  
6164                 }  
6165             }  
6166         }  
  
6168         /*  
6169          * Since we are changing page size we first have to flush  
6170          * the cache. This makes sure all the pagelock calls have  
6171          * to recheck protections.  
6172          */  
6173         if (svd->softlockcnt > 0) {  
6174             ASSERT(svd->tr_state == SEGVN_TR_OFF);  
  
6176             /*  
6177              * If this is shared segment non 0 softlockcnt  
6178              * means locked pages are still in use.  
6179              */  
6180             if (svd->type == MAP_SHARED) {  
6181                 return (EAGAIN);  
6182             }  
  
6184             /*  
6185              * Since we do have the segvn writers lock nobody can fill  
6186              * the cache with entries belonging to this seg during  
6187              * the purge. The flush either succeeds or we still have  
6188              * pending I/Os.  
6189              */  
6190             segvn_purge(seg);  
6191             if (svd->softlockcnt > 0) {  
6192                 return (EAGAIN);  
6193             }  
6194         }  
  
6196         if (HAT_IS_REGION_COOKIE_VALID(svd->rcookie)) {  
6197             ASSERT(svd->amp == NULL);  
6198             ASSERT(svd->tr_state == SEGVN_TR_OFF);  
6199             hat_leave_region(seg->s_as->a_hat, svd->rcookie,  
6200                             HAT_REGION_TEXT);  
6201             svd->rcookie = HAT_INVALID_REGION_COOKIE;  
6202         } else if (svd->tr_state == SEGVN_TR_INIT) {  
6203             svd->tr_state = SEGVN_TR_OFF;  
6204         } else if (svd->tr_state == SEGVN_TR_ON) {  
6205             ASSERT(svd->amp != NULL);  
6206             segvn_textunrepr(seg, 1);  
6207             ASSERT(svd->amp == NULL && svd->tr_state == SEGVN_TR_OFF);  
6208             amp = NULL;  
6209         }
```

```

6211      /*
6212       * Operation for sub range of existing segment.
6213       */
6214      if (addr != seg->s_base || eaddr != (seg->s_base + seg->s_size)) {
6215          if (szc < seg->s_szc) {
6216              VM_STAT_ADD(segvnvmstats.demoterange[2]);
6217              err = segvn_demote_range(seg, addr, len, SDR_RANGE, 0);
6218              if (err == 0) {
6219                  return (IE_RETRY);
6220              }
6221              if (err == ENOMEM) {
6222                  return (IE_NOMEM);
6223              }
6224          }
6225          return (err);
6226      }
6227      if (addr != seg->s_base) {
6228          nseg = segvn_split_seg(seg, addr);
6229          if (eaddr != (nseg->s_base + nseg->s_size)) {
6230              /* eaddr is szc aligned */
6231              (void) segvn_split_seg(nseg, eaddr);
6232          }
6233          return (IE_RETRY);
6234      }
6235      if (eaddr != (seg->s_base + seg->s_size)) {
6236          /* eaddr is szc aligned */
6237          (void) segvn_split_seg(seg, eaddr);
6238      }
6239      return (IE_RETRY);
6240  }
6241  /*
6242   * Break any low level sharing and reset seg->s_szc to 0.
6243   */
6244  if ((err = segvn_clrszc(seg)) != 0) {
6245      if (err == ENOMEM) {
6246          err = IE_NOMEM;
6247      }
6248      return (err);
6249  }
6250  ASSERT(seg->s_szc == 0);
6251  /*
6252   * If the end of the current segment is not pgsz aligned
6253   * then attempt to concatenate with the next segment.
6254   */
6255  if (!IS_P2ALIGNED(eaddr, pgsz)) {
6256      nseg = AS_SEGNEXT(seg->s_as, seg);
6257      if (nseg == NULL || nseg == seg || eaddr != nseg->s_base) {
6258          return (ENOMEM);
6259      }
6260      if (nseg->s_ops != &segvn_ops) {
6261          return (EINVAL);
6262      }
6263      nsvd = (struct segvn_data *)nseg->s_data;
6264      if (nsvd->softlockcnt > 0) {
6265          /*
6266           * If this is shared segment non 0 softlockcnt
6267           * means locked pages are still in use.
6268           */
6269          if (nsvd->type == MAP_SHARED) {
6270              return (EAGAIN);
6271          }
6272          segvn_purge(nseg);
6273          if (nsvd->softlockcnt > 0) {
6274              return (EAGAIN);
6275      }

```

```

6276          }
6277      }
6278      err = segvn_clrszc(nseg);
6279      if (err == ENOMEM) {
6280          err = IE_NOMEM;
6281      }
6282      if (err != 0) {
6283          return (err);
6284      }
6285      ASSERT(nsvd->rcookie == HAT_INVALID_REGION_COOKIE);
6286      err = segvn_concat(seg, nseg, 1);
6287      if (err == -1) {
6288          return (EINVAL);
6289      }
6290      if (err == -2) {
6291          return (IE_NOMEM);
6292      }
6293      return (IE_RETRY);
6294  }
6295  /*
6296   * May need to re-align anon array to
6297   * new szc.
6298   */
6299  if (amp != NULL) {
6300      if (!IS_P2ALIGNED(svd->anon_index, pgcnt)) {
6301          struct anon_hdr *nahp;
6302
6303          ASSERT(svd->type == MAP_PRIVATE);
6304
6305          ANON_LOCK_ENTER(&amp->a_rwlock, RW_WRITER);
6306          ASSERT(amp->refcnt == 1);
6307          nahp = anon_create(bttop(amp->size), ANON_NOSLEEP);
6308          if (nahp == NULL) {
6309              ANON_LOCK_EXIT(&amp->a_rwlock);
6310              return (IE_NOMEM);
6311          }
6312          if (anon_copy_ptr(amp->ahp, svd->anon_index,
6313                           nahp, 0, bttop(seg->s_size), ANON_NOSLEEP)) {
6314              anon_release(nahp, bttop(amp->size));
6315              ANON_LOCK_EXIT(&amp->a_rwlock);
6316          }
6317          return (IE_NOMEM);
6318      }
6319      anon_release(amp->ahp, bttop(amp->size));
6320      amp->ahp = nahp;
6321      svd->anon_index = 0;
6322      ANON_LOCK_EXIT(&amp->a_rwlock);
6323  }
6324  if (svd->vp != NULL && szc != 0) {
6325      struct vattr va;
6326      u_offset_t eoffpage = svd->offset;
6327      va.va_mask = AT_SIZE;
6328      eoffpage += seg->s_size;
6329      eoffpage = btopr(eoffpage);
6330      if (VOP_GETATTR(svd->vp, &va, 0, svd->cred, NULL) != 0) {
6331          segvn_setpgsz_getattr_err++;
6332          return (EINVAL);
6333      }
6334      if (btopr(va.va_size) < eoffpage) {
6335          segvn_setpgsz_eof_err++;
6336          return (EINVAL);
6337      }
6338      if (amp != NULL) {
6339          /*
6340           * anon_fill_cow_holes() may call VOP_GETPAGE().
6341
6342           */
6343      }
6344  }

```

```
6342             * don't take anon map lock here to avoid holding it
6343             * across VOP_GETPAGE() calls that may call back into
6344             * segvn for klsutering checks. We don't really need
6345             * anon map lock here since it's a private segment and
6346             * we hold as level lock as writers.
6347             */
6348     if ((err = anon_fill_cow_holes(seg, seg->s_base,
6349         amp->ahp, svd->anon_index, svd->vp, svd->offset,
6350         seg->s_size, szc, svd->prot, svd->vpage,
6351         svd->cred)) != 0) {
6352         return (EINVAL);
6353     }
6354     segvn_set vnode_mpss(svd->vp);
6355 }
6356
6357 if (amp != NULL) {
6358     ANON_LOCK_ENTER(&amp->a_rwlock, RW_WRITER);
6359     if (svd->type == MAP_PRIVATE) {
6360         amp->a_szc = szc;
6361     } else if (sdc > amp->a_sdc) {
6362         amp->a_sdc = szc;
6363     }
6364     ANON_LOCK_EXIT(&amp->a_rwlock);
6365 }
6366
6367 seg->s_sdc = szc;
6368
6369 return (0);
6370 }
6371 }
```

unchanged portion omitted

new/usr/src/uts/common/vm/vm\_as.c

```

***** Tue Nov 24 09:34:49 2015 *****
new/usr/src/uts/common/vm/vm_as.c
patch lower-case-segops
*****
_____ unchanged_portion_omitted _____
679 /*
680  * Free an address space data structure.
681  * Need to free the hat first and then
682  * all the segments on this as and finally
683  * the space for the as struct itself.
684 */
685 void
686 as_free(struct as *as)
687 {
688     struct hat *hat = as->a_hat;
689     struct seg *seg, *next;
690     int called = 0;
691
692 top:
693     /*
694      * Invoke ALL callbacks. as_do_callbacks will do one callback
695      * per call, and not return (-1) until the callback has completed.
696      * When as_do_callbacks returns zero, all callbacks have completed.
697      */
698     mutex_enter(&as->a_contents);
699     while (as->a_callbacks && as_do_callbacks(as, AS_ALL_EVENT, 0, 0))
700         ;
701
702     /* This will prevent new XHATs from attaching to as */
703     if (!called)
704         AS_SETBUSY(as);
705     mutex_exit(&as->a_contents);
706     AS_LOCK_ENTER(as, &as->a_lock, RW_WRITER);
707
708     if (!called) {
709         called = 1;
710         hat_free_start(hat);
711         if (as->a_xhat != NULL)
712             xhat_free_start_all(as);
713     }
714     for (seg = AS_SGFIRST(as); seg != NULL; seg = next) {
715         int err;
716
717         next = AS_SEGNEXT(as, seg);
718     retry:
719         err = segop_unmap(seg, seg->s_base, seg->s_size);
720         err = SEGOP_UNMAP(seg, seg->s_base, seg->s_size);
721         if (err == EAGAIN) {
722             mutex_enter(&as->a_contents);
723             if (as->a_callbacks) {
724                 AS_LOCK_EXIT(as, &as->a_lock);
725             } else if (!AS_ISNOUNMAPWAIT(as)) {
726                 /*
727                  * Memory is currently locked. Wait for a
728                  * cv_signal that it has been unlocked, then
729                  * try the operation again.
730                 */
731                 if (AS_ISUNMAPWAIT(as) == 0)
732                     cv_broadcast(&as->a_cv);
733                 AS_SETUNMAPWAIT(as);
734                 AS_LOCK_EXIT(as, &as->a_lock);
735                 while (AS_ISUNMAPWAIT(as))
736                     cv_wait(&as->a_cv, &as->a_contents);
737             } else {
738             }
739         }
740     }
741     AS_LOCK_EXIT(as, &as->a_lock);
742 }

```

new/usr/src/uts/common/vm/vm\_as.

```

new/usr/src/uts/common/vm/vm_as.c

803     mutex_exit(&newas->a_contents);
805
806     (void) hat_dup(as->a_hat, newas->a_hat, NULL, 0, HAT_DUP_SRD);
807
808     for (seg = AS_SEGFIRST(as); seg != NULL; seg = AS_SEGNEXT(as, seg)) {
809
810         if (seg->s_flags & S_PURGE) {
811             purgesize += seg->s_size;
812             continue;
813         }
814
815         newseg = seg_alloc(newas, seg->s_base, seg->s_size);
816         if (newseg == NULL) {
817             AS_LOCK_EXIT(newas, &newas->a_lock);
818             as_setwatch(as);
819             mutex_enter(&as->a_contents);
820             AS_CLRBUSY(as);
821             mutex_exit(&as->a_contents);
822             AS_LOCK_EXIT(as, &as->a_lock);
823             as_free(newas);
824             return (-1);
825         }
826         if ((error = segop_dup(seg, newseg)) != 0) {
827             if ((error = SEGOP_DUP(seg, newseg)) != 0) {
828                 /*
829                  * We call seg_free() on the new seg
830                  * because the segment is not set up
831                  * completely; i.e. it has no ops.
832                 */
833                 as_setwatch(as);
834                 mutex_enter(&as->a_contents);
835                 AS_CLRBUSY(as);
836                 mutex_exit(&as->a_contents);
837                 AS_LOCK_EXIT(as, &as->a_lock);
838                 seg_free(newseg);
839                 AS_LOCK_EXIT(newas, &newas->a_lock);
840                 as_free(newas);
841                 return (error);
842             }
843             newas->a_size += seg->s_size;
844         }
845         newas->a_resvsize = as->a_resvsize - purgesize;
846
847         error = hat_dup(as->a_hat, newas->a_hat, NULL, 0, HAT_DUP_ALL);
848         if (as->a_xhat != NULL)
849             error |= xhat_dup_all(as, newas, NULL, 0, HAT_DUP_ALL);
850
851         mutex_enter(&newas->a_contents);
852         AS_CLRBUSY(newas);
853         mutex_exit(&newas->a_contents);
854         AS_LOCK_EXIT(newas, &newas->a_lock);
855
856         as_setwatch(as);
857         mutex_enter(&as->a_contents);
858         AS_CLRBUSY(as);
859         mutex_exit(&as->a_contents);
860         AS_LOCK_EXIT(as, &as->a_lock);
861         if (error != 0) {
862             as_free(newas);
863             return (error);
864         }
865     }
866     forkedproc->p_as = newas;
867     return (0);
868 }

869 /*
```

```
new/usr/src/uts/common/vm/vm_as.c

868     * Handle a ``fault'' at addr for size bytes.
869     */
870     faultcode_t
871     as_fault(struct hat *hat, struct as *as, caddr_t addr, size_t size,
872             enum fault_type type, enum seg_rw rw)
873     {
874         struct seg *seg;
875         caddr_t raddr;                      /* rounded down addr */
876         size_t rsize;                      /* rounded up size */
877         size_t ssize;
878         faultcode_t res = 0;
879         caddr_t addrsav;
880         struct seg *segssav;
881         int as_lock_held;
882         klwp_t *lwp = ttolwp(curthread);
883         int is_xhat = 0;
884         int holding_wpage = 0;
885         extern struct seg_ops    segdev_ops;

889         if (as->a_hat != hat) {
890             /* This must be an XHAT then */
891             is_xhat = 1;
893             if ((type != F_INVAL) || (as == &kas))
894                 return (FC_NOSUPPORT);
895         }
897     retry:
898         if (!is_xhat) {
899             /*
900             * Indicate that the lwp is not to be stopped while waiting
901             * for a pagefault. This is to avoid deadlock while debugging
902             * a process via /proc over NFS (in particular).
903             */
904             if (lwp != NULL)
905                 lwp->lwp_nostop++;
907             /*
908             * same length must be used when we softlock and softunlock.
909             * We don't support softunlocking lengths less than
910             * the original length when there is largepage support.
911             * See seg_dev.c for more comments.
912             */
913             switch (type) {
915                 case F_SOFTLOCK:
916                     CPU_STATS_ADD_K(vm, softlock, 1);
917                     break;
919                 case F_SOFTUNLOCK:
920                     break;
922                 case F_PROT:
923                     CPU_STATS_ADD_K(vm, prot_fault, 1);
924                     break;
926                 case F_INVAL:
927                     CPU_STATS_ENTER_K();
928                     CPU_STATS_ADDQ(CPU, vm, as_fault, 1);
929                     if (as == &kas)
930                         CPU_STATS_ADDQ(CPU, vm, kernel_asflt, 1);
931                     CPU_STATS_EXIT_K();
932                     break;
933             }

```

new/usr/src/uts/common/vm/vm\_as.c

```

934
935     /* Kernel probe */
936     TNF_PROBE_3(address_fault, "vm pagefault", /* CSTYLED */,
937                 tnf_opaque, address,           addr,
938                 tnf_fault_type,        fault_type,      type,
939                 tnf_seg_access,       access,          rw);
940
941     raddr = (caddr_t)((uintptr_t)addr & (uintptr_t)PAGEMASK);
942     rsize = (((size_t)(addr + size) + PAGEOFFSET) & PAGEMASK) -
943             (size_t)raddr;
944
945     /*
946     * XXX -- Don't grab the as lock for segkmap. We should grab it for
947     * correctness, but then we could be stuck holding this lock for
948     * a LONG time if the fault needs to be resolved on a slow
949     * filesystem, and then no-one will be able to exec new commands,
950     * as exec'ing requires the write lock on the as.
951     */
952     if (as == &kas && segkmap && segkmap->s_base <= raddr &&
953         raddr + size < segkmap->s_base + segkmap->s_size) {
954         /*
955         * if (as==&kas), this can't be XHAT: we've already returned
956         * FC_NOSUPPORT.
957         */
958         seg = segkmap;
959         as_lock_held = 0;
960     } else {
961         AS_LOCK_ENTER(as, &as->a_lock, RW_READER);
962         if (is_xhat && avl_numnodes(&as->a_wpage) != 0) {
963             /*
964             * Grab and hold the writers' lock on the as
965             * if the fault is to a watched page.
966             * This will keep CPUs from "peeking" at the
967             * address range while we're temporarily boosting
968             * the permissions for the XHAT device to
969             * resolve the fault in the segment layer.
970             *
971             * We could check whether faulted address
972             * is within a watched page and only then grab
973             * the writer lock, but this is simpler.
974             */
975             AS_LOCK_EXIT(as, &as->a_lock);
976             AS_LOCK_ENTER(as, &as->a_lock, RW_WRITER);
977         }
978     }
979
980     seg = as_segat(as, raddr);
981     if (seg == NULL) {
982         AS_LOCK_EXIT(as, &as->a_lock);
983         if ((lwp != NULL) && (!is_xhat))
984             lwp->lwp_nostop--;
985         return (FC_NOMAP);
986     }
987
988     as_lock_held = 1;
989 }
990
991     addrsav = raddr;
992     segsav = seg;
993
994     for (; rsize != 0; rsize -= ssize, raddr += ssize) {
995         if (raddr >= seg->s_base + seg->s_size) {
996             seg = AS_SEGNEXT(as, seg);
997             if (seg == NULL || raddr != seg->s_base) {
998                 res = FC_NOMAP;
999                 break;

```

5

```

new/usr/src/uts/common/vm/vm_as.c

1000
1001
1002     }
1003     if (raddr + rsize > seg->s_base + seg->s_size)
1004         ssize = seg->s_base + seg->s_size - raddr;
1005     else
1006         ssize = rsize;
1007
1008     if (!is_xhat || (seg->s_ops != &segdev_ops)) {
1009
1010         if (is_xhat && avl_numnodes(&as->a_wpage) != 0 &&
1011             pr_is_watchpage_as(raddr, rw, as)) {
1012             /*
1013             * Handle watch pages. If we're faulting on a
1014             * watched page from an X-hat, we have to
1015             * restore the original permissions while we
1016             * handle the fault.
1017             */
1018             as_clearwatch(as);
1019             holding_wpage = 1;
1020         }
1021
1022     res = segop_fault(hat, seg, raddr, ssize, type, rw);
1023     res = SEGOP_FAULT(hat, seg, raddr, ssize, type, rw);
1024
1025     /* Restore watchpoints */
1026     if (holding_wpage) {
1027         as_setwatch(as);
1028         holding_wpage = 0;
1029     }
1030
1031     if (res != 0)
1032         break;
1033 } else {
1034     /* XHAT does not support seg_dev */
1035     res = FC_NOSUPPORT;
1036     break;
1037 }
1038
1039 /*
1040 * If we were SOFTLOCKING and encountered a failure,
1041 * we must SOFTUNLOCK the range we already did. (Maybe we
1042 * should just panic if we are SOFTLOCKING or even SOFTUNLOCKING
1043 * right here...)
1044 */
1045 if (res != 0 && type == F_SOFTLOCK) {
1046     for (seg = segsav; addrsav < raddr; addrsav += ssize) {
1047         if (addrsav >= seg->s_base + seg->s_size)
1048             seg = AS_SEGNEXT(as, seg);
1049         ASSERT(seg != NULL);
1050
1051         /*
1052         * Now call the fault routine again to perform the
1053         * unlock using S_OTHER instead of the rw variable
1054         * since we never got a chance to touch the pages.
1055         */
1056         if (raddr > seg->s_base + seg->s_size)
1057             ssize = seg->s_base + seg->s_size - addrsav;
1058         else
1059             ssize = raddr - addrsav;
1060         (void) segop_fault(hat, seg, addrsav, ssize,
1061                           (void) SEGOP_FAULT(hat, seg, addrsav, ssize,
1062                                             F_SOFTUNLOCK, S_OTHER));
1063     }
1064
1065     if (as_lock_held)
1066         AS_LOCK_EXIT(as, &as->a_lock);

```

```

1064     if ((lwp != NULL) && (!is_xhat))
1065         lwp->lwp_nostop--;
1066
1067     /*
1068      * If the lower levels returned EDEADLK for a fault,
1069      * It means that we should retry the fault. Let's wait
1070      * a bit also to let the deadlock causing condition clear.
1071      * This is part of a gross hack to work around a design flaw
1072      * in the ufs/sds logging code and should go away when the
1073      * logging code is re-designed to fix the problem. See bug
1074      * 4125102 for details of the problem.
1075    */
1076    if (FC_ERRNO(res) == EDEADLK) {
1077        delay(deadlk_wait);
1078        res = 0;
1079        goto retry;
1080    }
1081    return (res);
1082 }

1086 /*
1087  * Asynchronous ``fault'' at addr for size bytes.
1088 */
1089 faultcode_t
1090 as_faulta(struct as *as, caddr_t addr, size_t size)
1091 {
1092     struct seg *seg;
1093     caddr_t raddr;           /* rounded down addr */
1094     size_t rsize;            /* rounded up size */
1095     faultcode_t res = 0;
1096     klwp_t *lwp = ttolwp(curthread);

1098 retry:
1099     /*
1100      * Indicate that the lwp is not to be stopped while waiting
1101      * for a pagefault. This is to avoid deadlock while debugging
1102      * a process via /proc over NFS (in particular).
1103    */
1104    if (lwp != NULL)
1105        lwp->lwp_nostop++;

1107     raddr = (caddr_t)((uintptr_t)addr & (uintptr_t)PAGEMASK);
1108     rsize = (((size_t)(addr + size) + PAGEOFFSET) & PAGEMASK) -
1109             (size_t)raddr;

1111     AS_LOCK_ENTER(as, &as->a_lock, RW_READER);
1112     seg = as_segat(as, raddr);
1113     if (seg == NULL) {
1114         AS_LOCK_EXIT(as, &as->a_lock);
1115         if (lwp != NULL)
1116             lwp->lwp_nostop--;
1117         return (FC_NOMAP);
1118     }

1120     for (; rsize != 0; rsize -= PAGESIZE, raddr += PAGESIZE) {
1121         if (raddr >= seg->s_base + seg->s_size) {
1122             seg = AS_SEGNEXT(as, seg);
1123             if (seg == NULL || raddr != seg->s_base) {
1124                 res = FC_NOMAP;
1125                 break;
1126             }
1127         }
1128         res = segop_faulta(seg, raddr);
1129         res = SEGOP_FAULTA(seg, raddr);

```

```

1129         if (res != 0)
1130             break;
1131     }
1132     AS_LOCK_EXIT(as, &as->a_lock);
1133     if (lwp != NULL)
1134         lwp->lwp_nostop--;
1135
1136     /*
1137      * If the lower levels returned EDEADLK for a fault,
1138      * It means that we should retry the fault. Let's wait
1139      * a bit also to let the deadlock causing condition clear.
1140      * This is part of a gross hack to work around a design flaw
1141      * in the ufs/sds logging code and should go away when the
1142      * logging code is re-designed to fix the problem. See bug
1143      * 4125102 for details of the problem.
1144    */
1144    if (FC_ERRNO(res) == EDEADLK) {
1145        delay(deadlk_wait);
1146        res = 0;
1147        goto retry;
1148    }
1149    return (res);
1150 }

1152 /*
1153  * Set the virtual mapping for the interval from [addr : addr + size)
1154  * in address space 'as' to have the specified protection.
1155  * It is ok for the range to cross over several segments,
1156  * as long as they are contiguous.
1157 */
1158 int
1159 as_setprot(struct as *as, caddr_t addr, size_t size, uint_t prot)
1160 {
1161     struct seg *seg;
1162     struct as_callback *cb;
1163     size_t ssize;
1164     caddr_t raddr;           /* rounded down addr */
1165     size_t rsize;            /* rounded up size */
1166     int error = 0, writer = 0;
1167     caddr_t saveraddr;
1168     size_t saversize;

1170 setprot_top:
1171     raddr = (caddr_t)((uintptr_t)addr & (uintptr_t)PAGEMASK);
1172     rsize = (((size_t)(addr + size) + PAGEOFFSET) & PAGEMASK) -
1173             (size_t)raddr;
1174
1175     if (raddr + rsize < raddr)          /* check for wraparound */
1176         return (ENOMEM);
1177
1178     saveraddr = raddr;
1179     saversize = rsize;
1180
1181     /*
1182      * Normally we only lock the as as a reader. But
1183      * if due to setprot the segment driver needs to split
1184      * a segment it will return IE_RETRY. Therefore we re-acquire
1185      * the as lock as a writer so the segment driver can change
1186      * the seg list. Also the segment driver will return IE_RETRY
1187      * after it has changed the segment list so we therefore keep
1188      * locking as a writer. Since these operations should be rare
1189      * want to only lock as a writer when necessary.
1190    */
1191    if (writer || avl_numnodes(&as->a_wpage) != 0) {
1192        AS_LOCK_ENTER(as, &as->a_lock, RW_WRITER);
1193    } else {
1194        AS_LOCK_ENTER(as, &as->a_lock, RW_READER);

```

```

1195     }
1196
1197     as_clearwatchprot(as, raddr, rsize);
1198     seg = as_segat(as, raddr);
1199     if (seg == NULL) {
1200         as_setwatch(as);
1201         AS_LOCK_EXIT(as, &as->a_lock);
1202         return (ENOMEM);
1203     }
1204
1205     for (; rsize != 0; rsize -= ssize, raddr += ssize) {
1206         if (raddr >= seg->s_base + seg->s_size) {
1207             seg = AS_SEGNEXT(as, seg);
1208             if (seg == NULL || raddr != seg->s_base) {
1209                 error = ENOMEM;
1210                 break;
1211             }
1212         }
1213         if ((raddr + rsize) > (seg->s_base + seg->s_size))
1214             ssize = seg->s_base + seg->s_size - raddr;
1215         else
1216             ssize = rsize;
1217     retry:
1218     error = segop_setprot(seg, raddr, ssize, prot);
1219     error = SEGOP_SETPROT(seg, raddr, ssize, prot);
1220
1221     if (error == IE_NOMEM) {
1222         error = EAGAIN;
1223         break;
1224     }
1225
1226     if (error == IE_RETRY) {
1227         AS_LOCK_EXIT(as, &as->a_lock);
1228         writer = 1;
1229         goto setprot_top;
1230     }
1231
1232     if (error == EAGAIN) {
1233         /*
1234          * Make sure we have a_lock as writer.
1235         */
1236         if (writer == 0) {
1237             AS_LOCK_EXIT(as, &as->a_lock);
1238             writer = 1;
1239             goto setprot_top;
1240         }
1241
1242         /*
1243          * Memory is currently locked. It must be unlocked
1244          * before this operation can succeed through a retry.
1245          * The possible reasons for locked memory and
1246          * corresponding strategies for unlocking are:
1247          * (1) Normal I/O
1248          *      wait for a signal that the I/O operation
1249          *      has completed and the memory is unlocked.
1250          * (2) Asynchronous I/O
1251          *      The aio subsystem does not unlock pages when
1252          *      the I/O is completed. Those pages are unlocked
1253          *      when the application calls aiowait/aiocerror.
1254          *      So, to prevent blocking forever, cv_broadcast()
1255          *      is done to wake up aio_cleanup_thread.
1256          *      Subsequently, segvn_reclaim will be called, and
1257          *      that will do AS_CLRUNMAPWAIT() and wake us up.
1258          * (3) Long term page locking:
1259          *      Drivers intending to have pages locked for a
1260          *      period considerably longer than for normal I/O
1261
1262
1263
1264
1265
1266
1267
1268
1269
1270
1271
1272
1273
1274
1275
1276
1277
1278
1279
1280
1281
1282
1283
1284
1285
1286
1287
1288
1289
1290
1291
1292
1293
1294
1295
1296
1297
1298
1299
1300
1301
1302
1303
1304
1305
1306
1307
1308
1309
1310
1311
1312
1313
1314
1315
1316
1317
1318
1319
1320
1321
1322
1323
1324
1325

```

```

* (essentially forever) may have registered for a
* callback so they may unlock these pages on
* request. This is needed to allow this operation
* to succeed. Each entry on the callback list is
* examined. If the event or address range pertains
* to the callback is invoked (unless it already is in
* progress). The a_contents lock must be dropped
* before the callback, so only one callback can
* be done at a time. Go to the top and do more
* until zero is returned. If zero is returned,
* either there were no callbacks for this event
* or they were already in progress.
*/
mutex_enter(&as->a_contents);
if (as->a_callbacks &&
    (cb = as_find_callback(as, AS_SETPROT_EVENT,
                          seg->s_base, seg->s_size))) {
    AS_LOCK_EXIT(as, &as->a_lock);
    as_execute_callback(as, cb, AS_SETPROT_EVENT);
} else if (!AS_ISUNMAPWAIT(as)) {
    if (AS_ISUNMAPWAIT(as) == 0)
        cv_broadcast(&as->a_cv);
    AS_SETUNMAPWAIT(as);
    AS_LOCK_EXIT(as, &as->a_lock);
    while (AS_ISUNMAPWAIT(as))
        cv_wait(&as->a_cv, &as->a_contents);
} else {
/*
 * We may have raced with
 * segvn_reclaim() / segspt_reclaim(). In this
 * case clean noumapwait flag and retry since
 * softlockcnt in this segment may be already
 * 0. We don't drop as writer lock so our
 * number of retries without sleeping should
 * be very small. See segvn_reclaim() for
 * more comments.
*/
    AS_CLRUNMAPWAIT(as);
    mutex_exit(&as->a_contents);
    goto retry;
}
mutex_exit(&as->a_contents);
goto setprot_top;
} else if (error != 0)
    break;
}
if (error != 0) {
    as_setwatch(as);
} else {
    as_setwatchprot(as, saveraddr, saversize, prot);
}
AS_LOCK_EXIT(as, &as->a_lock);
return (error);

/*
 * Check to make sure that the interval [addr, addr + size)
 * in address space 'as' has at least the specified protection.
 * It is ok for the range to cross over several segments, as long
 * as they are contiguous.
*/
int
as_checkprot(struct as *as, caddr_t addr, size_t size, uint_t prot)
{
    struct seg *seg;
    size_t ssze;

```

```

1326     caddr_t raddr;           /* rounded down addr */
1327     size_t rsize;           /* rounded up size */
1328     int error = 0;
1329
1330     raddr = (caddr_t)((uintptr_t)addr & (uintptr_t)PAGEMASK);
1331     rsize = ((size_t)(addr + size) + PAGEOFFSET) & PAGEMASK) -
1332         (size_t)raddr;
1333
1334     if (raddr + rsize < raddr)          /* check for wraparound */
1335         return (ENOMEM);
1336
1337     /*
1338      * This is ugly as sin...
1339      * Normally, we only acquire the address space readers lock.
1340      * However, if the address space has watchpoints present,
1341      * we must acquire the writer lock on the address space for
1342      * the benefit of as_clearwatchprot() and as_setwatchprot().
1343      */
1344     if (avl_numnodes(&as->a_wpage) != 0)
1345         AS_LOCK_ENTER(as, &as->a_lock, RW_WRITER);
1346     else
1347         AS_LOCK_ENTER(as, &as->a_lock, RW_READER);
1348     as_clearwatchprot(as, raddr, rsize);
1349     seg = as_segat(as, raddr);
1350     if (seg == NULL) {
1351         as_setwatch(as);
1352         AS_LOCK_EXIT(as, &as->a_lock);
1353         return (ENOMEM);
1354     }
1355
1356     for (; rsize != 0; rsize -= ssize, raddr += ssize) {
1357         if (raddr >= seg->s_base + seg->s_size) {
1358             seg = AS_SEGNEXT(as, seg);
1359             if (seg == NULL || raddr != seg->s_base) {
1360                 error = ENOMEM;
1361                 break;
1362             }
1363         }
1364         if ((raddr + rsize) > (seg->s_base + seg->s_size))
1365             ssize = seg->s_base + seg->s_size - raddr;
1366         else
1367             ssize = rsize;
1368
1369         error = segop_checkprot(seg, raddr, ssize, prot);
1370         error = SEGOP_CHECKPROT(seg, raddr, ssize, prot);
1371         if (error != 0)
1372             break;
1373     as_setwatch(as);
1374     AS_LOCK_EXIT(as, &as->a_lock);
1375     return (error);
1376 }
1377 int
1378 as_unmap(struct as *as, caddr_t addr, size_t size)
1379 {
1380     struct seg *seg, *seg_next;
1381     struct as_callback *cb;
1382     caddr_t raddr, eaddr;
1383     size_t ssize, rsize = 0;
1384     int err;
1385
1386 top:
1387     raddr = (caddr_t)((uintptr_t)addr & (uintptr_t)PAGEMASK);
1388     eaddr = (caddr_t)((((uintptr_t)(addr + size) + PAGEOFFSET) &
1389                      (uintptr_t))PAGEMASK);

```

```

1392     AS_LOCK_ENTER(as, &as->a_lock, RW_WRITER);
1393     as->a_updatedir = 1;      /* inform /proc */
1394     gethrestime(&as->a_updatetime);
1395
1396     /*
1397      * Use as_findseg to find the first segment in the range, then
1398      * step through the segments in order, following s_next.
1399      */
1400     as_clearwatchprot(as, raddr, eaddr - raddr);
1401
1402     for (seg = as_findseg(as, raddr, 0); seg != NULL; seg = seg->s_next) {
1403         if (eaddr <= seg->s_base)
1404             break;                /* eaddr was in a gap; all done */
1405
1406         /* this is implied by the test above */
1407         ASSERT(raddr < eaddr);
1408
1409         if (raddr < seg->s_base)
1410             raddr = seg->s_base;    /* raddr was in a gap */
1411
1412         if (eaddr > (seg->s_base + seg->s_size))
1413             ssize = seg->s_base + seg->s_size - raddr;
1414         else
1415             ssize = eaddr - raddr;
1416
1417         /*
1418          * Save next segment pointer since seg can be
1419          * destroyed during the segment unmap operation.
1420          */
1421         seg_next = AS_SEGNEXT(as, seg);
1422
1423         /*
1424          * We didn't count /dev/null mappings, so ignore them here.
1425          * We'll handle MAP_NORESERVE cases in segvn_unmap(). (Again,
1426          * we have to do this check here while we have seg.)
1427          */
1428         rsize = 0;
1429         if (!SEG_IS_DEVNULL_MAPPING(seg) &&
1430             !SEG_IS_PARTIAL_RESV(seg))
1431             rsize = ssize;
1432
1433     retry:
1434     err = segop_unmap(seg, raddr, ssize);
1435     err = SEGOP_UNMAP(seg, raddr, ssize);
1436     if (err == EAGAIN) {
1437         /*
1438          * Memory is currently locked. It must be unlocked
1439          * before this operation can succeed through a retry.
1440          * The possible reasons for locked memory and
1441          * corresponding strategies for unlocking are:
1442          * (1) Normal I/O
1443          *      wait for a signal that the I/O operation
1444          *      has completed and the memory is unlocked.
1445          * (2) Asynchronous I/O
1446          *      The aio subsystem does not unlock pages when
1447          *      the I/O is completed. Those pages are unlocked
1448          *      when the application calls aiowait/aioerror.
1449          *      So, to prevent blocking forever, cv_broadcast()
1450          *      is done to wake up aio_cleanup_thread.
1451          *      Subsequently, segvn_reclaim will be called, and
1452          *      that will do AS_CLRUNMAPWAIT() and wake us up.
1453          * (3) Long term page locking:
1454          *      Drivers intending to have pages locked for a
1455          *      period considerably longer than for normal I/O

```

```

1456      * (essentially forever) may have registered for a
1457      * callback so they may unlock these pages on
1458      * request. This is needed to allow this operation
1459      * to succeed. Each entry on the callback list is
1460      * examined. If the event or address range pertains
1461      * the callback is invoked (unless it already is in
1462      * progress). The a_contents lock must be dropped
1463      * before the callback, so only one callback can
1464      * be done at a time. Go to the top and do more
1465      * until zero is returned. If zero is returned,
1466      * either there were no callbacks for this event
1467      * or they were already in progress.
1468 */
1469 mutex_enter(&as->a_contents);
1470 if (as->a_callbacks &&
1471     (cb = as_find_callback(as, AS_UNMAP_EVENT,
1472                           seg->s_base, seg->s_size))) {
1473     AS_LOCK_EXIT(as, &as->a_lock);
1474     as_execute_callback(as, cb, AS_UNMAP_EVENT);
1475 } else if (!AS_ISUNMAPWAIT(as)) {
1476     if (AS_ISUNMAPWAIT(as) == 0)
1477         cv_broadcast(&as->a_cv);
1478     AS_SETUNMAPWAIT(as);
1479     AS_LOCK_EXIT(as, &as->a_lock);
1480     while (AS_ISUNMAPWAIT(as))
1481         cv_wait(&as->a_cv, &as->a_contents);
1482 } else {
1483     /*
1484     * We may have raced with
1485     * segvn_reclaim() / segspt_reclaim(). In this
1486     * case clean noumapwait flag and retry since
1487     * softlockcnt in this segment may be already
1488     * 0. We don't drop as writer lock so our
1489     * number of retries without sleeping should
1490     * be very small. See segvn_reclaim() for
1491     * more comments.
1492     */
1493     AS_CLRNUMAPWAIT(as);
1494     mutex_exit(&as->a_contents);
1495     goto retry;
1496 }
1497 mutex_exit(&as->a_contents);
1498 goto top;
1499 } else if (err == IE_RETRY) {
1500     AS_LOCK_EXIT(as, &as->a_lock);
1501     goto top;
1502 } else if (err) {
1503     as_setwatch(as);
1504     AS_LOCK_EXIT(as, &as->a_lock);
1505     return (-1);
1506 }
1507 as->a_size -= ssize;
1508 if (rsize)
1509     as->a_resvsize -= rsize;
1510 raddr += ssize;
1511 }
1512 AS_LOCK_EXIT(as, &as->a_lock);
1513 return (0);
1514 }
1515 }

unchanged_portion_omitted_

1848 /*
1849 * Delete all segments in the address space marked with S_PURGE.
1850 * This is currently used for Sparc V9nofault ASI segments (seg_nf.c).

```

```

1851  * These segments are deleted as a first step before calls to as_gap(), so
1852  * that they don't affect mmap() or shmat().
1853  */
1854 void
1855 as_purge(struct as *as)
1856 {
1857     struct seg *seg;
1858     struct seg *next_seg;
1859
1860     /*
1861      * the setting of NEEDSPURGE is protect by as_rangelock(), so
1862      * no need to grab a_contents mutex for this check
1863      */
1864     if ((as->a_flags & AS_NEEDSPURGE) == 0)
1865         return;
1866
1867     AS_LOCK_ENTER(as, &as->a_lock, RW_WRITER);
1868     next_seg = NULL;
1869     seg = AS_SEGFIRST(as);
1870     while (seg != NULL) {
1871         next_seg = AS_SEGNEXT(as, seg);
1872         if (seg->s_flags & S_PURGE)
1873             (void) segop_unmap(seg, seg->s_base, seg->s_size);
1874             SEGOP_UNMAP(seg, seg->s_base, seg->s_size);
1875         seg = next_seg;
1876     }
1877     AS_LOCK_EXIT(as, &as->a_lock);
1878
1879     mutex_enter(&as->a_contents);
1880     as->a_flags &= ~AS_NEEDSPURGE;
1881     mutex_exit(&as->a_contents);
1882 }

unchanged_portion_omitted_

2143 /*
2144  * Swap the pages associated with the address space as out to
2145  * secondary storage, returning the number of bytes actually
2146  * swapped.
2147  *
2148  * The value returned is intended to correlate well with the process's
2149  * memory requirements. Its usefulness for this purpose depends on
2150  * how well the segment-level routines do at returning accurate
2151  * information.
2152  */
2153 size_t
2154 as_swapout(struct as *as)
2155 {
2156     struct seg *seg;
2157     size_t swpcnt = 0;

2158     /*
2159      * Kernel-only processes have given up their address
2160      * spaces. Of course, we shouldn't be attempting to
2161      * swap out such processes in the first place...
2162      */
2163     if (as == NULL)
2164         return (0);

2165     AS_LOCK_ENTER(as, &as->a_lock, RW_READER);

2166     /* Prevent XHATS from attaching */
2167     mutex_enter(&as->a_contents);
2168     AS_SETBUSY(as);
2169     mutex_exit(&as->a_contents);

```

```

2175     /*
2176      * Free all mapping resources associated with the address
2177      * space. The segment-level swapout routines capitalize
2178      * on this unmapping by scavenging pages that have become
2179      * unmapped here.
2180     */
2181     hat_swapout(as->a_hat);
2182     if (as->a_xhat != NULL)
2183         xhat_swapout_all(as);

2185     mutex_enter(&as->a_contents);
2186     AS_CLRBUSY(as);
2187     mutex_exit(&as->a_contents);

2189     /*
2190      * Call the swapout routines of all segments in the address
2191      * space to do the actual work, accumulating the amount of
2192      * space reclaimed.
2193     */
2194     for (seg = AS_SEGFIRST(as); seg != NULL; seg = AS_SEGNEXT(as, seg)) {
2195         struct seg_ops *ov = seg->s_ops;

2197         /*
2198          * We have to check to see if the seg has
2199          * an ops vector because the seg may have
2200          * been in the middle of being set up when
2201          * the process was picked for swapout.
2202        */
2203         if ((ov != NULL) && (ov->swapout != NULL))
2204             swpcnt += segop_swapout(seg);
2205         swpcnt += SEGOP_SWAPOUT(seg);

2206     AS_LOCK_EXIT(as, &as->a_lock);
2207     return (swpcnt);

2208 }

2210 /*
2211  * Determine whether data from the mappings in interval [addr, addr + size)
2212  * are in the primary memory (core) cache.
2213 */
2214 int
2215 as_incore(struct as *as, caddr_t addr,
2216           size_t size, char *vec, size_t *sizep)
2217 {
2218     struct seg *seg;
2219     size_t ssize;
2220     caddr_t raddr;           /* rounded down addr */
2221     size_t rsize;            /* rounded up size */
2222     size_t isize;            /* iteration size */
2223     int error = 0;           /* result, assume success */

2225     *sizep = 0;
2226     raddr = (caddr_t)((uintptr_t)addr & (uintptr_t)PAGEMASK);
2227     rsize = (((size_t)addr + size) + PAGEOFFSET) & PAGEMASK) -
2228             (size_t)raddr;

2230     if (raddr + rsize < raddr)           /* check for wraparound */
2231         return (ENOMEM);

2233     AS_LOCK_ENTER(as, &as->a_lock, RW_READER);
2234     seg = as_segat(as, raddr);
2235     if (seg == NULL) {
2236         AS_LOCK_EXIT(as, &as->a_lock);
2237         return (-1);
2238     }

```

```

2240     for (; rsize != 0; rsize -= ssize, raddr += ssize) {
2241         if (raddr >= seg->s_base + seg->s_size) {
2242             seg = AS_SEGNEXT(as, seg);
2243             if (seg == NULL || raddr != seg->s_base) {
2244                 error = -1;
2245                 break;
2246             }
2247         }
2248         if ((raddr + rsize) > (seg->s_base + seg->s_size))
2249             ssize = seg->s_base + seg->s_size - raddr;
2250         else
2251             ssize = rsize;
2252         *sizep += isize = segop_incore(seg, raddr, ssize, vec);
2253         *sizep += isize = SEGOP_INCORE(seg, raddr, ssize, vec);
2254         if (isize != ssize) {
2255             error = -1;
2256             break;
2257         }
2258         vec += bttop(ssize);
2259     }
2260     AS_LOCK_EXIT(as, &as->a_lock);
2261     return (error);
2262 }

2263 static void
2264 as_segunlock(struct seg *seg, caddr_t addr, int attr,
2265               ulong_t *bitmap, size_t position, size_t npages)
2266 {
2267     caddr_t range_start;
2268     size_t pos1 = position;
2269     size_t pos2;
2270     size_t size;
2271     size_t end_pos = npages + position;

2273     while (bt_range(bitmap, &pos1, &pos2, end_pos)) {
2274         size = ptob((pos2 - pos1));
2275         range_start = (caddr_t)((uintptr_t)addr +
2276                               ptob(pos1 - position));

2278         (void) segop_lockop(seg, range_start, size, attr, MC_UNLOCK,
2279                             (void) SEGOP_LOCKOP(seg, range_start, size, attr, MC_UNLOCK,
2280                             (ulong_t *)NULL, (size_t)NULL));
2281         pos1 = pos2;
2282     }
2283     unchanged_portion_omitted

2307 /*
2308  * Cache control operations over the interval [addr, addr + size) in
2309  * address space "as".
2310 */
2311 /*ARGSUSED*/
2312 int
2313 as_ctl(struct as *as, caddr_t addr, size_t size, int func, int attr,
2314         uintptr_t arg, ulong_t *lock_map, size_t pos)
2315 {
2316     struct seg *seg;           /* working segment */
2317     caddr_t raddr;            /* rounded down addr */
2318     caddr_t initraddr;        /* saved initial rounded down addr */
2319     size_t rsize;              /* rounded up size */
2320     size_t initrsize;          /* saved initial rounded up size */
2321     size_t ssize;              /* size of seg */
2322     int error = 0;              /* result */
2323     size_t mlock_size;          /* size of bitmap */
2324     ulong_t *mlock_map;        /* pointer to bitmap used */
2325     /* to represent the locked */

```

```

2326             /* pages. */
2327     retry:
2328         if (error == IE_RETRY)
2329             AS_LOCK_ENTER(as, &as->a_lock, RW_WRITER);
2330         else
2331             AS_LOCK_ENTER(as, &as->a_lock, RW_READER);
2332
2333         /*
2334          * If these are address space lock/unlock operations, loop over
2335          * all segments in the address space, as appropriate.
2336         */
2337     if (func == MC_LOCKAS) {
2338         size_t npages, idx;
2339         size_t rlen = 0;           /* rounded as length */
2340
2341         idx = pos;
2342
2343         if (arg & MCL_FUTURE) {
2344             mutex_enter(&as->a_contents);
2345             AS_SETPGLCK(as);
2346             mutex_exit(&as->a_contents);
2347         }
2348         if ((arg & MCL_CURRENT) == 0) {
2349             AS_LOCK_EXIT(as, &as->a_lock);
2350             return (0);
2351         }
2352
2353         seg = AS_SEGFIRST(as);
2354         if (seg == NULL) {
2355             AS_LOCK_EXIT(as, &as->a_lock);
2356             return (0);
2357         }
2358         do {
2359             raddr = (caddr_t)((uintptr_t)seg->s_base &
2360                               (uintptr_t)PAGEMASK);
2361             rlen += (((uintptr_t)(seg->s_base + seg->s_size) +
2362                       PAGEOFFSET) & PAGEMASK) - (uintptr_t)raddr;
2363         } while ((seg = AS_SEGNEXT(as, seg)) != NULL);
2364
2365         mlock_size = BT_BITOUL(btopr(rlen));
2366         if ((mlock_map = (ulong_t *)kmem_zalloc(mlock_size *
2367                                               sizeof (ulong_t), KM_NOSLEEP)) == NULL) {
2368             AS_LOCK_EXIT(as, &as->a_lock);
2369             return (EAGAIN);
2370         }
2371
2372         for (seg = AS_SEGFIRST(as); seg; seg = AS_SEGNEXT(as, seg)) {
2373             error = segop_lockop(seg, seg->s_base,
2374                                 error = SEGOP_LOCKOP(seg, seg->s_base,
2375                                         seg->s_size, attr, MC_LOCK, mlock_map, pos);
2376             if (error != 0)
2377                 break;
2378             pos += seg_pages(seg);
2379         }
2380
2381         if (error) {
2382             for (seg = AS_SEGFIRST(as); seg != NULL;
2383                  seg = AS_SEGNEXT(as, seg)) {
2384
2385                 raddr = (caddr_t)((uintptr_t)seg->s_base &
2386                                   (uintptr_t)PAGEMASK);
2387                 npages = seg_pages(seg);
2388                 as_segunlock(seg, raddr, attr, mlock_map,
2389                             idx, npages);
2390                 idx += npages;

```

```

2391             }
2392         }
2393         kmem_free(mlock_map, mlock_size * sizeof (ulong_t));
2394         AS_LOCK_EXIT(as, &as->a_lock);
2395         goto lockerr;
2396     } else if (func == MC_UNLOCKAS) {
2397         mutex_enter(&as->a_contents);
2398         AS_CLRPGLCK(as);
2399         mutex_exit(&as->a_contents);
2400
2401         for (seg = AS_SEGFIRST(as); seg; seg = AS_SEGNEXT(as, seg)) {
2402             error = segop_lockop(seg, seg->s_base,
2403                                 error = SEGOP_LOCKOP(seg, seg->s_base,
2404                                         seg->s_size, attr, MC_UNLOCK, NULL, 0);
2405             if (error != 0)
2406                 break;
2407         }
2408         AS_LOCK_EXIT(as, &as->a_lock);
2409         goto lockerr;
2410     }
2411
2412     /*
2413      * Normalize addresses and sizes.
2414      */
2415     initraddr = raddr = (caddr_t)((uintptr_t)addr & (uintptr_t)PAGEMASK);
2416     initrsize = rsize = (((size_t)(addr + size) + PAGEOFFSET) & PAGEMASK) -
2417                   (size_t)raddr;
2418
2419     if (raddr + rsize < raddr) {           /* check for wraparound */
2420         AS_LOCK_EXIT(as, &as->a_lock);
2421         return (ENOMEM);
2422     }
2423
2424     /*
2425      * Get initial segment.
2426      */
2427     if ((seg = as_segat(as, raddr)) == NULL) {
2428         AS_LOCK_EXIT(as, &as->a_lock);
2429         return (ENOMEM);
2430     }
2431
2432     if (func == MC_LOCK) {
2433         mlock_size = BT_BITOUL(btopr(rsize));
2434         if ((mlock_map = (ulong_t *)kmem_zalloc(mlock_size *
2435                                               sizeof (ulong_t), KM_NOSLEEP)) == NULL) {
2436             AS_LOCK_EXIT(as, &as->a_lock);
2437             return (EAGAIN);
2438         }
2439     }
2440
2441     /*
2442      * Loop over all segments. If a hole in the address range is
2443      * discovered, then fail. For each segment, perform the appropriate
2444      * control operation.
2445      */
2446     while (rsize != 0) {
2447
2448         /*
2449          * Make sure there's no hole, calculate the portion
2450          * of the next segment to be operated over.
2451          */
2452         if (raddr >= seg->s_base + seg->s_size) {
2453             seg = AS_SEGNEXT(as, seg);
2454             if (seg == NULL || raddr != seg->s_base) {

```

```

2456
2457     if (func == MC_LOCK) {
2458         as_unlockerr(as, attr, mlock_map,
2459                     initraddr, initrsize - rsize);
2460         kmem_free(mlock_map,
2461                   mlock_size * sizeof (ulong_t));
2462     }
2463     AS_LOCK_EXIT(as, &as->a_lock);
2464     return (ENOMEM);
2465 }
2466 if ((raddr + rsize) > (seg->s_base + seg->s_size))
2467     ssize = seg->s_base + seg->s_size - raddr;
2468 else
2469     ssize = rsize;
2470 /*
2471 * Dispatch on specific function.
2472 */
2473 switch (func) {
2474 /*
2475 * Synchronize cached data from mappings with backing
2476 * objects.
2477 */
2478 case MC_SYNC:
2479     if (error = segop_sync(seg, raddr, ssize,
2480                           if (error = SEGOP_SYNC(seg, raddr, ssize,
2481                                     attr, (uint_t)arg)) {
2482                                         AS_LOCK_EXIT(as, &as->a_lock);
2483                                         return (error);
2484                                     }
2485     break;
2486 /*
2487 * Lock pages in memory.
2488 */
2489 case MC_LOCK:
2490     if (error = segop_lockop(seg, raddr, ssize,
2491                           if (error = SEGOP_LOCKOP(seg, raddr, ssize,
2492                                     attr, func, mlock_map, pos)) {
2493                                         as_unlockerr(as, attr, mlock_map, initraddr,
2494                                         initrsize - rsize + ssize);
2495                                         kmem_free(mlock_map, mlock_size *
2496                                         sizeof (ulong_t));
2497                                         AS_LOCK_EXIT(as, &as->a_lock);
2498                                         goto lockerr;
2499                                     }
2500     break;
2501 /*
2502 * Unlock mapped pages.
2503 */
2504 case MC_UNLOCK:
2505     if (void) segop_lockop(seg, raddr, ssize, attr, func,
2506                           (void) SEGOP_LOCKOP(seg, raddr, ssize, attr, func,
2507                                         (ulong_t *)NULL, (size_t)NULL);
2508     break;
2509 /*
2510 * Store VM advise for mapped pages in segment layer.
2511 */
2512 case MC_ADVISE:
2513     error = segop_advise(seg, raddr, ssize, (uint_t)arg);
2514     error = SEGOP_ADVISE(seg, raddr, ssize, (uint_t)arg);
2515
2516 */
2517

```

```

2518     * Check for regular errors and special retry error
2519     */
2520     if (error) {
2521         if (error == IE_RETRY) {
2522             /*
2523             * Need to acquire writers lock, so
2524             * have to drop readers lock and start
2525             * all over again
2526             */
2527             AS_LOCK_EXIT(as, &as->a_lock);
2528             goto retry;
2529         } else if (error == IE_REATTACH) {
2530             /*
2531             * Find segment for current address
2532             * because current segment just got
2533             * split or concatenated
2534             */
2535             seg = as_segat(as, raddr);
2536             if (seg == NULL) {
2537                 AS_LOCK_EXIT(as, &as->a_lock);
2538                 return (ENOMEM);
2539             }
2540         } else {
2541             /*
2542             * Regular error
2543             */
2544             AS_LOCK_EXIT(as, &as->a_lock);
2545             return (error);
2546         }
2547     }
2548     break;
2549
2550 case MC_INHERIT_ZERO:
2551     if (seg->s_ops->inherit == NULL) {
2552         error = ENOTSUP;
2553     } else {
2554         error = segop_inherit(seg, raddr, ssize,
2555                               SEGOP_INHERIT(seg, raddr, ssize,
2556                                             SEGP_INH_ZERO));
2557     }
2558     if (error != 0) {
2559         AS_LOCK_EXIT(as, &as->a_lock);
2560         return (error);
2561     }
2562     break;
2563 /*
2564 * Can't happen.
2565 */
2566 default:
2567     panic("as_ctl: bad operation %d", func);
2568     /*NOTREACHED*/
2569 }
2570
2571     rsize -= ssize;
2572     raddr += ssize;
2573 }
2574
2575 if (func == MC_LOCK)
2576     kmem_free(mlock_map, mlock_size * sizeof (ulong_t));
2577     AS_LOCK_EXIT(as, &as->a_lock);
2578     return (0);
2579 lockerr:
2580
2581 /*
2582 * If the lower levels returned EDEADLK for a segment lockop,

```

new/usr/src/uts/common/vm/vm\_as.c

21

```

2583 * it means that we should retry the operation. Let's wait
2584 * a bit also to let the deadlock causing condition clear.
2585 * This is part of a gross hack to work around a design flaw
2586 * in the ufs/sds logging code and should go away when the
2587 * logging code is re-designed to fix the problem. See bug
2588 * 4125102 for details of the problem.
2589 */
2590     if (error == EDEADLK) {
2591         delay(deadlk_wait);
2592         error = 0;
2593         goto retry;
2594     }
2595     return (error);
2596 }



---


unchanged_portion_omitted

2617 /*
2618  * Pagelock pages from a range that spans more than 1 segment. Obtain shadow
2619  * lists from each segment and copy them to one contiguous shadow list (plist)
2620  * as expected by the caller. Save pointers to per segment shadow lists at
2621  * the tail of plist so that they can be used during as_pageunlock().
2622 */
2623 static int
2624 as_pagelock_segs(struct as *as, struct seg *seg, struct page ***ppp,
2625 caddr_t addr, size_t size, enum seg_rw rw)
2626 {
2627     caddr_t sv_addr = addr;
2628     size_t sv_size = size;
2629     struct seg *sv_seg = seg;
2630     ulong_t segcnt = 1;
2631     ulong_t cnt;
2632     size_t ssize;
2633     pgcnt_t npages = btop(size);
2634     page_t **plist;
2635     page_t **pl;
2636     int error;
2637     caddr_t eaddr;
2638     faultcode_t fault_err = 0;
2639     pgcnt_t pl_off;
2640     extern struct seg_ops segspt_shmops;

2642     ASSERT(AS_LOCK_HELD(as, &as->a_lock));
2643     ASSERT(seg != NULL);
2644     ASSERT(addr >= seg->s_base && addr < seg->s_base + seg->s_size);
2645     ASSERT(addr + size > seg->s_base + seg->s_size);
2646     ASSERT(IS_P2ALIGNED(size, PAGESIZE));
2647     ASSERT(IS_P2ALIGNED(addr, PAGESIZE));

2649 /*
2650  * Count the number of segments covered by the range we are about to
2651  * lock. The segment count is used to size the shadow list we return
2652  * back to the caller.
2653 */
2654 for (; size != 0; size -= ssize, addr += ssize) {
2655     if (addr >= seg->s_base + seg->s_size) {

2657         seg = AS_SEGNEXT(as, seg);
2658         if (seg == NULL || addr != seg->s_base) {
2659             AS_LOCK_EXIT(as, &as->a_lock);
2660             return (EFAULT);
2661         }
2662     /*
2663      * Do a quick check if subsequent segments
2664      * will most likely support pagelock.
2665      */
2666     if (seg->s_ops == &segvn_ops) {

```

new/usr/src/uts/common/vm/vm\_as.c

```

2667 vnode_t *vp;
2668
2669 if (segop_getvp(seg, addr, &vp) != 0 ||

2670 if (SEGOP_GETVP(seg, addr, &vp) != 0 ||

2671 vp != NULL) {
2672     AS_LOCK_EXIT(as, &as->a_lock);
2673     goto slow;
2674 }
2675 } else if (seg->s_ops != &segsp->shmops) {
2676     AS_LOCK_EXIT(as, &as->a_lock);
2677     goto slow;
2678 }
2679 segcnt++;
2680 if (addr + size > seg->s_base + seg->s_size) {
2681     ssize = seg->s_base + seg->s_size - addr;
2682 } else {
2683     ssize = size;
2684 }
2685 }
2686 ASSERT(segcnt > 1);

2687 plist = kmalloc_zalloc((npages + segcnt) * sizeof(page_t *), KM_SLEEP);

2688 addr = sv_addr;
2689 size = sv_size;
2690 seg = sv_seg;

2691 for (cnt = 0, pl_off = 0; size != 0; size -= ssize, addr += ssize) {
2692     if (addr >= seg->s_base + seg->s_size) {
2693         seg = AS_SEGNEXT(as, seg);
2694         ASSERT(seg != NULL && addr == seg->s_base);
2695         cnt++;
2696         ASSERT(cnt < segcnt);
2697     }
2698     if (addr + size > seg->s_base + seg->s_size) {
2699         ssize = seg->s_base + seg->s_size - addr;
2700     } else {
2701         ssize = size;
2702     }
2703     pl = &plist[npages + cnt];
2704     error = segop_pagelock(seg, addr, ssize, (page_t *** )pl,
2705     error = SEGOP_PAGELOCK(seg, addr, ssize, (page_t *** )pl,
2706     L_PAGELOCK, rw);
2707     if (error) {
2708         break;
2709     }
2710     ASSERT(plist[npages + cnt] != NULL);
2711     ASSERT(pl_off + btop(ssize) <= npages);
2712     bcopy(plist[npages + cnt], &plist[pl_off],
2713           btop(ssize) * sizeof(page_t *));
2714     pl_off += btop(ssize);
2715 }
2716
2717 if (size == 0) {
2718     AS_LOCK_EXIT(as, &as->a_lock);
2719     ASSERT(cnt == segcnt - 1);
2720     *ppp = plist;
2721     return (0);
2722 }
2723
2724 /*
2725 * one of pagelock calls failed. The error type is in error variable.
2726 * Unlock what we've locked so far and retry with F_SOFTLOCK if error
2727 * type is either EFAULT or ENOTSUP. Otherwise just return the error
2728 * back to the caller.
2729 */

```

```

2731     */
2732
2733     eaddr = addr;
2734     seg = sv_seg;
2735
2736     for (cnt = 0, addr = sv_addr; addr < eaddr; addr += ssize) {
2737         if (addr >= seg->s_base + seg->s_size) {
2738             seg = AS_SEGNEXT(as, seg);
2739             ASSERT(seg != NULL && addr == seg->s_base);
2740             cnt++;
2741             ASSERT(cnt < segcnt);
2742         }
2743         if (eaddr > seg->s_base + seg->s_size) {
2744             ssize = seg->s_base + seg->s_size - addr;
2745         } else {
2746             ssize = eaddr - addr;
2747         }
2748         pl = &plist[npages + cnt];
2749         ASSERT(*pl != NULL);
2750         (void) segop_pagelock(seg, addr, ssize, (page_t ***)pl,
2751                               (void) SEGOP_PAGELOCK(seg, addr, ssize, (page_t ***))pl,
2752                               L_PAGEUNLOCK, rw);
2753     }
2754     AS_LOCK_EXIT(as, &as->a_lock);
2755
2756     kmem_free(plist, (npages + segcnt) * sizeof (page_t *));
2757
2758     if (error != ENOTSUP && error != EFAULT) {
2759         return (error);
2760     }
2761
2762 slow:
2763     /*
2764      * If we are here because pagelock failed due to the need to cow fault
2765      * in the pages we want to lock F_SOFTLOCK will do this job and in
2766      * next as_pagelock() call for this address range pagelock will
2767      * hopefully succeed.
2768     */
2769     fault_err = as_fault(as->a_hat, as, sv_addr, sv_size, F_SOFTLOCK, rw);
2770     if (fault_err != 0) {
2771         return (fc_decode(fault_err));
2772     }
2773     *ppp = NULL;
2774
2775     return (0);
2776 }
2777 */
2778 * lock pages in a given address space. Return shadow list. If
2779 * the list is NULL, the MMU mapping is also locked.
2780 */
2781 */
2782 int
2783 as_pagelock(struct as *as, struct page ***ppp, caddr_t addr,
2784               size_t size, enum seg_rw rw)
2785 {
2786     size_t rsize;
2787     caddr_t raddr;
2788     faultcode_t fault_err;
2789     struct seg *seg;
2790     int err;
2791
2792     TRACE_2(TR_FAC_PHYSIO, TR_PHYSIO_AS_LOCK_START,
2793             "as_pagelock_start: addr %p size %ld", addr, size);
2794
2795     raddr = (caddr_t)((uintptr_t)addr & (uintptr_t)PAGEMASK);

```

```

2796     rsize = (((size_t)(addr + size) + PAGEOFFSET) & PAGEMASK) -
2797             (size_t)raddr;
2798
2799     /*
2800      * if the request crosses two segments let
2801      * as_fault handle it.
2802     */
2803     AS_LOCK_ENTER(as, &as->a_lock, RW_READER);
2804
2805     seg = as_segat(as, raddr);
2806     if (seg == NULL) {
2807         AS_LOCK_EXIT(as, &as->a_lock);
2808         return (EFAULT);
2809     }
2810     ASSERT(raddr >= seg->s_base && raddr < seg->s_base + seg->s_size);
2811     if (raddr + rsize > seg->s_base + seg->s_size) {
2812         return (as_pagelock_segs(as, seg, ppp, raddr, rsize, rw));
2813     }
2814     if (raddr + rsize <= raddr) {
2815         AS_LOCK_EXIT(as, &as->a_lock);
2816         return (EFAULT);
2817     }
2818
2819     TRACE_2(TR_FAC_PHYSIO, TR_PHYSIO_SEG_LOCK_START,
2820             "seg_lock_1_start: raddr %p rsize %ld", raddr, rsize);
2821
2822     /*
2823      * try to lock pages and pass back shadow list
2824     */
2825     err = segop_pagelock(seg, raddr, rsize, ppp, L_PAGELOCK, rw);
2826     err = SEGOP_PAGELOCK(seg, raddr, rsize, ppp, L_PAGELOCK, rw);
2827
2828     TRACE_0(TR_FAC_PHYSIO, TR_PHYSIO_SEG_LOCK_END, "seg_lock_1_end");
2829     AS_LOCK_EXIT(as, &as->a_lock);
2830
2831     if (err == 0 || (err != ENOTSUP && err != EFAULT)) {
2832         return (err);
2833     }
2834
2835     /*
2836      * Use F_SOFTLOCK to lock the pages because pagelock failed either due
2837      * to no pagelock support for this segment or pages need to be cow
2838      * faulted in. If fault is needed F_SOFTLOCK will do this job for
2839      * this as_pagelock() call and in the next as_pagelock() call for the
2840      * same address range pagelock call will hopefully succeed.
2841     */
2842     fault_err = as_fault(as->a_hat, as, addr, size, F_SOFTLOCK, rw);
2843     if (fault_err != 0) {
2844         return (fc_decode(fault_err));
2845     }
2846     *ppp = NULL;
2847
2848     TRACE_0(TR_FAC_PHYSIO, TR_PHYSIO_AS_LOCK_END, "as_pagelock_end");
2849     return (0);
2850 }
2851
2852 /*
2853  * unlock pages locked by as_pagelock_segs(). Retrieve per segment shadow
2854  * lists from the end of plist and call pageunlock interface for each segment.
2855  * Drop as lock and free plist.
2856 */
2857 static void
2858 as_pageunlock_segs(struct as *as, struct seg *seg, caddr_t addr, size_t size,
2859                      struct page **plist, enum seg_rw rw)
2860 {

```

```

2861     ulong_t cnt;
2862     caddr_t eaddr = addr + size;
2863     pgcnt_t npages = btop(size);
2864     size_t ssize;
2865     page_t **pl;
2866
2867     ASSERT(AS_LOCK_HELD(as, &as->a_lock));
2868     ASSERT(seg != NULL);
2869     ASSERT(addr >= seg->s_base && addr < seg->s_base + seg->s_size);
2870     ASSERT(addr + size > seg->s_base + seg->s_size);
2871     ASSERT(IS_P2ALIGNED(size, PAGESIZE));
2872     ASSERT(IS_P2ALIGNED(addr, PAGESIZE));
2873     ASSERT(plist != NULL);
2874
2875     for (cnt = 0; addr < eaddr; addr += ssize) {
2876         if (addr >= seg->s_base + seg->s_size) {
2877             seg = AS_SEGNEXT(as, seg);
2878             ASSERT(seg != NULL && addr == seg->s_base);
2879             cnt++;
2880
2881         if (eaddr > seg->s_base + seg->s_size) {
2882             ssize = seg->s_base + seg->s_size - addr;
2883         } else {
2884             ssize = eaddr - addr;
2885         }
2886         pl = &plist[npages + cnt];
2887         ASSERT(*pl != NULL);
2888         (void) segop_pagelock(seg, addr, ssize, (page_t ***)pl,
2889                               (void) SEGOP_PAGELOCK(seg, addr, ssize, (page_t ***)pl,
2890                                         L_PAGEUNLOCK, rw);
2891     }
2892     ASSERT(cnt > 0);
2893     AS_LOCK_EXIT(as, &as->a_lock);
2894
2895     cnt++;
2896     kmem_free(plist, (npages + cnt) * sizeof (page_t *));
2897 }
2898 */
2899 /* unlock pages in a given address range
2900 */
2901 void
2902 as_pageunlock(struct as *as, struct page **pp, caddr_t addr, size_t size,
2903   enum seg_rw rw)
2904 {
2905     struct seg *seg;
2906     size_t rsize;
2907     caddr_t raddr;
2908
2909     TRACE_2(TR_FAC_PHYSIO, TR_PHYSIO_AS_UNLOCK_START,
2910             "as_pageunlock_start: addr %p size %ld", addr, size);
2911
2912     /*
2913      * if the shadow list is NULL, as_pagelock was
2914      * falling back to as_fault
2915     */
2916     if (pp == NULL) {
2917         (void) as_fault(as->a_hat, as, addr, size, F_SOFTUNLOCK, rw);
2918         return;
2919     }
2920
2921     raddr = (caddr_t)((uintptr_t)addr & (uintptr_t)PAGEMASK);
2922     rsize = ((size_t)(addr + size) + PAGEOFFSET) & PAGEMASK) -
2923             (size_t)raddr;
2924
2925     AS_LOCK_ENTER(as, &as->a_lock, RW_READER);

```

```

2926     seg = as_segat(as, raddr);
2927     ASSERT(seg != NULL);
2928
2929     TRACE_2(TR_FAC_PHYSIO, TR_PHYSIO_SEG_UNLOCK_START,
2930             "seg_unlock_start: raddr %p rsize %ld", raddr, rsize);
2931
2932     ASSERT(raddr >= seg->s_base && raddr < seg->s_base + seg->s_size);
2933     if (raddr + rsize <= seg->s_base + seg->s_size) {
2934         (void) segop_pagelock(seg, raddr, rsize, &pp, L_PAGEUNLOCK, rw);
2935         SEGOP_PAGELOCK(seg, raddr, rsize, &pp, L_PAGEUNLOCK, rw);
2936     } else {
2937         as_pageunlock_segs(as, seg, raddr, rsize, pp, rw);
2938         return;
2939     }
2940     AS_LOCK_EXIT(as, &as->a_lock);
2941     TRACE_0(TR_FAC_PHYSIO, TR_PHYSIO_AS_UNLOCK_END, "as_pageunlock_end");
2942 }
2943 int
2944 as_setpagesize(struct as *as, caddr_t addr, size_t size, uint_t szc,
2945   boolean_t wait)
2946 {
2947     struct seg *seg;
2948     size_t ssize;
2949     caddr_t raddr; /* rounded down addr */
2950     size_t rsize; /* rounded up size */
2951     int error = 0;
2952     size_t pgsz = page_get_pagesize(szc);
2953
2954     setpgsz_top:
2955     if (!IS_P2ALIGNED(addr, pgsz) || !IS_P2ALIGNED(size, pgsz)) {
2956         return (EINVAL);
2957     }
2958
2959     raddr = addr;
2960     rsize = size;
2961
2962     if (raddr + rsize < raddr) /* check for wraparound */
2963         return (ENOMEM);
2964
2965     AS_LOCK_ENTER(as, &as->a_lock, RW_WRITER);
2966     as_clearwatchprot(as, raddr, rsize);
2967     seg = as_segat(as, raddr);
2968     if (seg == NULL) {
2969         as_setwatch(as);
2970         AS_LOCK_EXIT(as, &as->a_lock);
2971         return (ENOMEM);
2972     }
2973
2974     for (; rsize != 0; rsize -= ssize, raddr += ssize) {
2975         if (raddr >= seg->s_base + seg->s_size) {
2976             seg = AS_SEGNEXT(as, seg);
2977             if (seg == NULL || raddr != seg->s_base) {
2978                 error = ENOMEM;
2979                 break;
2980             }
2981         }
2982         if ((raddr + rsize) > (seg->s_base + seg->s_size)) {
2983             ssize = seg->s_base + seg->s_size - raddr;
2984         } else {
2985             ssize = rsize;
2986         }
2987
2988     retry:
2989         error = segop_setpagesize(seg, raddr, ssize, szc);
2990         error = SEGOP_SETPAGESIZE(seg, raddr, ssize, szc);
2991     }
2992 }

```

```

2991     if (error == IE_NOMEM) {
2992         error = EAGAIN;
2993         break;
2994     }
2995
2996     if (error == IE_RETRY) {
2997         AS_LOCK_EXIT(as, &as->a_lock);
2998         goto setpagesize_top;
2999     }
3000
3001     if (error == ENOTSUP) {
3002         error = EINVAL;
3003         break;
3004     }
3005
3006     if (wait && (error == EAGAIN)) {
3007         /*
3008          * Memory is currently locked. It must be unlocked
3009          * before this operation can succeed through a retry.
3010          * The possible reasons for locked memory and
3011          * corresponding strategies for unlocking are:
3012          * (1) Normal I/O
3013          *      wait for a signal that the I/O operation
3014          *      has completed and the memory is unlocked.
3015          * (2) Asynchronous I/O
3016          *      The aio subsystem does not unlock pages when
3017          *      the I/O is completed. Those pages are unlocked
3018          *      when the application calls aiowait/aiocerror.
3019          *      So, to prevent blocking forever, cv_broadcast()
3020          *      is done to wake up aio_cleanup_thread.
3021          *      Subsequently, segvn_reclaim will be called, and
3022          *      that will do AS_CLRUNMAPWAIT() and wake us up.
3023          * (3) Long term page locking:
3024          *      This is not relevant for as_setpagesize()
3025          *      because we cannot change the page size for
3026          *      driver memory. The attempt to do so will
3027          *      fail with a different error than EAGAIN so
3028          *      there's no need to trigger as callbacks like
3029          *      as_unmap, as_setprot or as_free would do.
3030          */
3031     mutex_enter(&as->a_contents);
3032     if (!AS_ISUNMAPWAIT(as)) {
3033         if (AS_ISUNMAPWAIT(as) == 0) {
3034             cv_broadcast(&as->a_cv);
3035         }
3036         AS_SETUNMAPWAIT(as);
3037         AS_LOCK_EXIT(as, &as->a_lock);
3038         while (AS_ISUNMAPWAIT(as)) {
3039             cv_wait(&as->a_cv, &as->a_contents);
3040         }
3041     } else {
3042         /*
3043          * We may have raced with
3044          * segvn_reclaim() / segspt_reclaim(). In this
3045          * case clean nouunmapwait flag and retry since
3046          * softlockcnt in this segment may be already
3047          * 0. We don't drop as writer lock so our
3048          * number of retries without sleeping should
3049          * be very small. See segvn_reclaim() for
3050          * more comments.
3051          */
3052         AS_CLRNUUNMAPWAIT(as);
3053         mutex_exit(&as->a_contents);
3054         goto retry;
3055     }

```

```

3056             mutex_exit(&as->a_contents);
3057             goto setpagesize_top;
3058         } else if (error != 0) {
3059             break;
3060         }
3061     }
3062     as_setwatch(as);
3063     AS_LOCK_EXIT(as, &as->a_lock);
3064     return (error);
3065 }
3066 /*
3067  * as_iset3_default_lpsize() just calls segop_setpagesize() on all segments
3068  * as_iset3_default_lpsize() just calls SEGOP_SETPAGESIZE() on all segments
3069  * in its chunk where s_szc is less than the szc we want to set.
3070 */
3071 static int
3072 as_iset3_default_lpsize(struct as *as, caddr_t raddr, size_t rsize, uint_t szc,
3073     int *retry)
3074 {
3075     struct seg *seg;
3076     size_t ssize;
3077     int error;
3078
3079     ASSERT(AS_WRITE_HELD(as, &as->a_lock));
3080
3081     seg = as_segat(as, raddr);
3082     if (seg == NULL) {
3083         panic("as_iset3_default_lpsize: no seg");
3084     }
3085
3086     for (; rsize != 0; rsize -= ssize, raddr += ssize) {
3087         if (raddr >= seg->s_base + seg->s_size) {
3088             seg = AS_SEGNEXT(as, seg);
3089             if (seg == NULL || raddr != seg->s_base) {
3090                 panic("as_iset3_default_lpsize: as changed");
3091             }
3092             if ((raddr + rsize) > (seg->s_base + seg->s_size)) {
3093                 ssize = seg->s_base + seg->s_size - raddr;
3094             } else {
3095                 ssize = rsize;
3096             }
3097         }
3098         if (szc > seg->s_szc) {
3099             error = segop_setpagesize(seg, raddr, ssize, szc);
3100             error = SEGOP_SETPAGESIZE(seg, raddr, ssize, szc);
3101             /* Only retry on EINVAL segments that have no vnode. */
3102             if (error == EINVAL) {
3103                 vnode_t *vp = NULL;
3104                 if (((segop_gettype(seg, raddr) & MAP_SHARED) &&
3105                     (segop_getvp(seg, raddr, &vp) != 0 ||

3106                     if (((SEGOP_GETTYPE(seg, raddr) & MAP_SHARED) &&
3107                         (SEGOP_GETVP(seg, raddr, &vp) != 0 ||
3108                         vp == NULL)) {
3109                             *retry = 1;
3110                         } else {
3111                             *retry = 0;
3112                         }
3113                         if (error) {
3114                             return (error);
3115                         }
3116                     }
3117             }
3118             return (0);
3119         }

```

```

3118 }
_____unchanged_portion_omitted_____
3305 /*
3306  * Set the default large page size for the range. Called via memcntl with
3307  * page size set to 0. as_set_default_lpsize breaks the range down into
3308  * chunks with the same type/flags, ignores non-segvn segments, and passes
3309  * each chunk to as_iset_default_lpsize().
3310 */
3311 int
3312 as_set_default_lpsize(struct as *as, caddr_t addr, size_t size)
3313 {
3314     struct seg *seg;
3315     caddr_t raddr;
3316     size_t rsize;
3317     size_t ssize;
3318     int rtype, rflags;
3319     int stype, sflags;
3320     int error;
3321     caddr_t setaddr;
3322     size_t setsize;
3323     int segvn;
3324
3325     if (size == 0)
3326         return (0);
3327
3328 again: AS_LOCK_ENTER(as, &as->a_lock, RW_WRITER);
3329     error = 0;
3330
3331     raddr = (caddr_t)((uintptr_t)addr & (uintptr_t)PAGEMASK);
3332     rsize = (((size_t)(addr + size) + PAGEOFFSET) & PAGEMASK) -
3333             (size_t)raddr;
3334
3335     if (raddr + rsize < raddr) { /* check for wraparound */
3336         AS_LOCK_EXIT(as, &as->a_lock);
3337         return (ENOMEM);
3338     }
3339     as_clearwatchprot(as, raddr, rsize);
3340     seg = as_segat(as, raddr);
3341     if (seg == NULL) {
3342         as_setwatch(as);
3343         AS_LOCK_EXIT(as, &as->a_lock);
3344         return (ENOMEM);
3345     }
3346     if (seg->s_ops == &segvn_ops) {
3347         rtype = segop_gettype(seg, addr);
3348         rtype = SEGOP_GETTYPE(seg, addr);
3349         rflags = rtype & (MAP_TEXT | MAP_INITDATA);
3350         rtype = rtype & (MAP_SHARED | MAP_PRIVATE);
3351         segvn = 1;
3352     } else {
3353         segvn = 0;
3354     }
3355     setaddr = raddr;
3356     setsize = 0;
3357
3358     for (; rsize != 0; rsize -= ssize, raddr += ssize, setsize += ssize) {
3359         if (raddr >= (seg->s_base + seg->s_size)) {
3360             seg = AS_SEGNEXT(as, seg);
3361             if (seg == NULL || raddr != seg->s_base) {
3362                 error = ENOMEM;
3363                 break;
3364             }
3365             if (seg->s_ops == &segvn_ops) {
3366                 stype = segop_gettype(seg, raddr);

```

```

3366
3367
3368
3369
3370
3371
3372
3373
3374
3375
3376
3377
3378
3379
3380
3381
3382
3383
3384
3385
3386
3387
3388
3389
3390
3391
3392
3393
3394
3395
3396
3397
3398
3399
3400
3401
3402
3403
3404
3405
3406
3407
3408
3409
3410
3411
3412
3413
3414
3415
3416
3417
3418
3419
3420
3421
3422
3423
3424
3425
3426
3427
3428
3429
3430
3431
3432
3433
3434
3435
3436
3437
3438
3439
3440
3441
3442
3443
3444
3445
3446
3447
3448
3449
3450
3451
3452
3453
3454
3455
3456
3457
3458
3459
3460
3461
3462
3463
3464
3465
3466
3467
3468
3469
3470
3471
3472
3473
3474
3475
3476
3477
3478
3479
3480
3481
3482
3483
3484
3485
3486
3487
3488
3489
3490
3491
3492
3493
3494
3495
3496
3497
3498
3499
3500
3501
3502
3503
3504
3505
3506
3507
3508
3509
3510
3511
3512
3513
3514
3515
3516
3517
3518
3519
3520
3521
3522
3523
3524
3525
3526
3527
3528
3529
3530
3531
3532
3533
3534
3535
3536
3537
3538
3539
3540
3541
3542
3543
3544
3545
3546
3547
3548
3549
3550
3551
3552
3553
3554
3555
3556
3557
3558
3559
3560
3561
3562
3563
3564
3565
3566
3567
3568
3569
3570
3571
3572
3573
3574
3575
3576
3577
3578
3579
3580
3581
3582
3583
3584
3585
3586
3587
3588
3589
3590
3591
3592
3593
3594
3595
3596
3597
3598
3599
3600
3601
3602
3603
3604
3605
3606
3607
3608
3609
3610
3611
3612
3613
3614
3615
3616
3617
3618
3619
3620
3621
3622
3623
3624
3625
3626
3627
3628
3629
3630
3631
3632
3633
3634
3635
3636
3637
3638
3639
3640
3641
3642
3643
3644
3645
3646
3647
3648
3649
3650
3651
3652
3653
3654
3655
3656
3657
3658
3659
3660
3661
3662
3663
3664
3665
3666
3667
3668
3669
3670
3671
3672
3673
3674
3675
3676
3677
3678
3679
3680
3681
3682
3683
3684
3685
3686
3687
3688
3689
3690
3691
3692
3693
3694
3695
3696
3697
3698
3699
3700
3701
3702
3703
3704
3705
3706
3707
3708
3709
3710
3711
3712
3713
3714
3715
3716
3717
3718
3719
3720
3721
3722
3723
3724
3725
3726
3727
3728
3729
3730
3731
3732
3733
3734
3735
3736
3737
3738
3739
3740
3741
3742
3743
3744
3745
3746
3747
3748
3749
3750
3751
3752
3753
3754
3755
3756
3757
3758
3759
3760
3761
3762
3763
3764
3765
3766
3767
3768
3769
3770
3771
3772
3773
3774
3775
3776
3777
3778
3779
3780
3781
3782
3783
3784
3785
3786
3787
3788
3789
3790
3791
3792
3793
3794
3795
3796
3797
3798
3799
3800
3801
3802
3803
3804
3805
3806
3807
3808
3809
3810
3811
3812
3813
3814
3815
3816
3817
3818
3819
3820
3821
3822
3823
3824
3825
3826
3827
3828
3829
3830
3831
3832
3833
3834
3835
3836
3837
3838
3839
3840
3841
3842
3843
3844
3845
3846
3847
3848
3849
3850
3851
3852
3853
3854
3855
3856
3857
3858
3859
3860
3861
3862
3863
3864
3865
3866
3867
3868
3869
3870
3871
3872
3873
3874
3875
3876
3877
3878
3879
3880
3881
3882
3883
3884
3885
3886
3887
3888
3889
3890
3891
3892
3893
3894
3895
3896
3897
3898
3899
3900
3901
3902
3903
3904
3905
3906
3907
3908
3909
3910
3911
3912
3913
3914
3915
3916
3917
3918
3919
3920
3921
3922
3923
3924
3925
3926
3927
3928
3929
3930
3931
3932
3933
3934
3935
3936
3937
3938
3939
3940
3941
3942
3943
3944
3945
3946
3947
3948
3949
3950
3951
3952
3953
3954
3955
3956
3957
3958
3959
3960
3961
3962
3963
3964
3965
3966
3967
3968
3969
3970
3971
3972
3973
3974
3975
3976
3977
3978
3979
3980
3981
3982
3983
3984
3985
3986
3987
3988
3989
3990
3991
3992
3993
3994
3995
3996
3997
3998
3999
3999
4000
4001
4002
4003
4004
4005
4006
4007
4008
4009
4010
4011
4012
4013
4014
4015
4016
4017
4018
4019
4020
4021
4022
4023
4024
4025
4026
4027
4028
4029
4030
4031
4032
4033
4034
4035
4036
4037
4038
4039
4040
4041
4042
4043
4044
4045
4046
4047
4048
4049
4050
4051
4052
4053
4054
4055
4056
4057
4058
4059
4060
4061
4062
4063
4064
4065
4066
4067
4068
4069
4070
4071
4072
4073
4074
4075
4076
4077
4078
4079
4080
4081
4082
4083
4084
4085
4086
4087
4088
4089
4090
4091
4092
4093
4094
4095
4096
4097
4098
4099
4099
4100
4101
4102
4103
4104
4105
4106
4107
4108
4109
4110
4111
4112
4113
4114
4115
4116
4117
4118
4119
4120
4121
4122
4123
4124
4125
4126
4127
4128
4129
4130
4131
4132
4133
4134
4135
4136
4137
4138
4139
4139
4140
4141
4142
4143
4144
4145
4146
4147
4148
4149
4149
4150
4151
4152
4153
4154
4155
4156
4157
4158
4159
4159
4160
4161
4162
4163
4164
4165
4166
4167
4168
4169
4169
4170
4171
4172
4173
4174
4175
4176
4177
4178
4179
4179
4180
4181
4182
4183
4184
4185
4186
4187
4188
4189
4189
4190
4191
4192
4193
4194
4195
4196
4197
4198
4199
4199
4200
4201
4202
4203
4204
4205
4206
4207
4208
4209
4209
4210
4211
4212
4213
4214
4215
4216
4217
4218
4219
4219
4220
4221
4222
4223
4224
4225
4226
4227
4228
4229
4229
4230
4231
4232
4233
4234
4235
4236
4237
4238
4239
4239
4240
4241
4242
4243
4244
4245
4246
4247
4248
4249
4249
4250
4251
4252
4253
4254
4255
4256
4257
4258
4259
4259
4260
4261
4262
4263
4264
4265
4266
4267
4268
4269
4269
4270
4271
4272
4273
4274
4275
4276
4277
4278
4279
4279
4280
4281
4282
4283
4284
4285
4286
4287
4288
4289
4289
4290
4291
4292
4293
4294
4295
4296
4297
4298
4299
4299
4300
4301
4302
4303
4304
4305
4306
4307
4308
4309
4309
4310
4311
4312
4313
4314
4315
4316
4317
4318
4319
4319
4320
4321
4322
4323
4324
4325
4326
4327
4328
4329
4329
4330
4331
4332
4333
4334
4335
4336
4337
4338
4339
4339
4340
4341
4342
4343
4344
4345
4346
4347
4348
4349
4349
4350
4351
4352
4353
4354
4355
4356
4357
4358
4359
4359
4360
4361
4362
4363
4364
4365
4366
4367
4368
4369
4369
4370
4371
4372
4373
4374
4375
4376
4377
4378
4379
4379
4380
4381
4382
4383
4384
4385
4386
4387
4388
4389
4389
4390
4391
4392
4393
4394
4395
4396
4397
4398
4399
4399
4400
4401
4402
4403
4404
4405
4406
4407
4408
4409
4409
4410
4411
4412
4413
4414
4415
4416
4417
4418
4419
4419
4420
4421
4422
4423
4424
4425
4426
4427
4428
4429
4429
4430
4431
4432
4433
4434
4435
4436
4437
4438
4439
4439
4440
4441
4442
4443
4444
4445
4446
4447
4448
4449
4449
4450
4451
4452
4453
4454
4455
4456
4457
4458
4459
4459
4460
4461
4462
4463
4464
4465
4466
4467
4468
4469
4469
4470
4471
4472
4473
4474
4475
4476
4477
4478
4479
4479
4480
4481
4482
4483
4484
4485
4486
4487
4488
4489
4489
4490
4491
4492
4493
4494
4495
4496
4497
4498
4498
4499
4499
4500
4501
4502
4503
4504
4505
4506
4507
4508
4509
4509
4510
4511
4512
4513
4514
4515
4516
4517
4518
4519
4519
4520
4521
4522
4523
4524
4525
4526
4527
4528
4529
4529
4530
4531
4532
4533
4534
4535
4536
4537
4538
4539
4539
4540
4541
4542
4543
4544
4545
4546
4547
4548
4549
4549
4550
4551
4552
4553
4554
4555
4556
4557
4558
4559
4559
4560
4561
4562
4563
4564
4565
4566
4567
4568
4569
4569
4570
4571
4572
4573
4574
4575
4576
4577
4578
4579
4579
4580
4581
4582
4583
4584
4585
4586
4587
4588
4589
4589
4590
4591
4592
4593
4594
4595
4596
4597
4598
4598
4599
4599
4600
4601
4602
4603
4604
4605
4606
4607
4608
4609
4609
4610
4611
4612
4613
4614
4615
4616
4617
4618
4619
4619
4620
4621
4622
4623
4624
4625
4626
4627
4628
4629
4629
4630
4631
4632
4633
4634
4635
4636
4637
4638
4639
4639
4640
4641
4642
4643
4644
4645
4646
4647
4648
4649
4649
4650
4651
4652
4653
4654
4655
4656
4657
4658
4659
4659
4660
4661
4662
4663
4664
4665
4666
4667
4668
4669
4669
4670
4671
4672
4673
4674
4675
4676
4677
4678
4679
4679
4680
4681
4682
4683
4684
4685
4686
4687
4688
4689
4689
4690
4691
4692
4693
4694
4695
4696
4697
4698
4698
4699
4699
4700
4701
4702
4703
4704
4705
4706
4707
4708
4709
4709
4710
4711
4712
4713
4714
4715
4716
4717
4718
4719
4719
4720
4721
4722
4723
4724
4725
4726
4727
4728
4729
4729
4730
4731
4732
4733
4734
4735
4736
4737
4738
4739
4739
4740
4741
4742
4743
4744
4745
4746
4747
4748
4749
4749
4750
4751
4752
4753
4754
4755
4756
4757
4758
4759
4759
4760
4761
4762
4763
4764
4765
4766
4767
4768
4769
4769
4770
4771
4772
4773
4774
4775
4776
4777
4778
4779
4779
4780
4781
4782
4783
4784
4785
4786
4787
4788
4789
4789
4790
4791
4792
4793
4794
4795
4796
4797
4798
4798
4799
4799
4800
4801
4802
4803
4804
4805
4806
4807
4808
4809
4809
4810
4811
4812
4813
4814
4815
4816
4817
4818
4819
4819
4820
4821
4822
4823
4824
4825
4826
4827
4828
4829
4829
4830
4831
4832
4833
4834
4835
4836
4837
4838
4839
4839
4840
4841
4842
4843
4844
4845
4846
4847
4848
4849
4849
4850
4851
4852
4853
4854
4855
4856
4857
4858
4859
4859
4860
4861
4862
4863
4864
4865
4866
4867
4868
4869
4869
4870
4871
4872
4873
4874
4875
4876
4877
4878
4879
4879
4880
4881
4882
4883
4884
4885
4886
4887
4888
4889
4889
4890
4891
4892
4893
4894
4895
4896
4897
4898
4898
4899
4899
4900
4901
4902
4903
4904
4905
4906
4907
4908
4909
4909
4910
4911
4912
4913
4914
4915
4916
4917
4918
4919
4919
4920
4921
4922
4923
4924
4925
4926
4927
4928
4929
4929
4930
4931
4932
4933
4934
4935
4936
4937
4938
4939
4939
4940
4941
4942
4943
4944
4945
4946
4947
4948
4949
4949
4950
4951
4952
4953
4954
4955
4956
4957
4958
4959
4959
4960
4961
4962
4963
4964
4965
4966
4967
4968
4969
4969
4970
4971
4972
4973
4974
4975
4976
4977
4978
4979
4979
4980
4981
4982
4983
4984
4985
4986
4987
4988
4989
4989
4990
4991
4992
4993
4994
4995
4996
4997
4998
4999
4999
5000
5001
5002
5003
5004
5005
5006
5007
5008
5009
5009
5010
5011
5012
5013
5014
5015
5016
5017
5018
5019
5019
5020
5021
5022
5023
5024
5025
5026
5027
5028
5029
5029
5030
5031
5032
5033
5034
5035
5036
5037
5038
5039
5039
5040
5041
5042
5043
5044
5045
5046
5047
5048
5049
5049
5050
5051
5052
5053
5054
5055
5056
5057
5058
5059
5059
5060
5061
5062
5063
5064
5065
5066
5067
5068
5069
5069
5070
5071
5072
5073
5074
5075
5076
5077
5078
5079
5079
5080
5081
5082
5083
5084
5085
5086
5087
5088
5089
5089
5090
5091
5092
5093
5094
5095
5096
5097
5098
5099
5099
5100
5101
5102
5103
5104
5105
5106
5107
5108
5109
5109
5110
5111
5112
5113
5114
5115
5116
5117
5118
5119
5119
5120
5121
5122
5123
5124
5125
5126
5127
5128
5129
5129
5130
5131
5132
5133
5134
5135
5136
5137
5138
5139
5139
5140
5141
5142
5143
5144
5145
5146
5147
5148
5149
5149
5150
5151
5152
5153
5154
5155
5156
5157
5158
5159
5159
5160
5161
5162
5163
5164
5165
5166
5167
5168
5169
5169
5170
5171
5172
5173
5174
5175
5176
5177
5178
5179
5179
5180
5181
5182
5183
5184
5185
5186
5187
5188
5189
5189
5190
5191
5192
5193
5194
5195
5196
5197
5198
5199
5199
5200
5201
5202
5203
5204
5205
5206
5207
5208
5209
5209
5210
5211
5212
5213
5214
5215
5216
5217
5218
5219
5219
5220
5221
5222
5223
5224
5225
5226
5227
5228
5229
5229
5230
5231
5232
5233
5234
5235
5236
5237
5238
5239
5239
5240
5241
5242
5243
5244
5245
5246
5247
5248
5249
5249
5250
5251
5252
5253
5254
5255
5256
5257
5258
5259
5259
5260
5261
5262
5263
5264
5265
5266
5267
5268
5269
5269
5270
5271
5272
5273
5274
5275
5276
5277
5278
5279
5279
5280
5281
5282
5283
5284
5285
5286
5287
5288
5289
5289
5290
5291
5292
5293
5294
5295
5296
5297
5298
5299
5299
5300
5301
5302
5303
5304
5305
5306
```

```

new/usr/src/uts/common/vm/vm_as.c

3432 }
3433 mutex_exit(&
3434 AS_LOCK_EXITEE
3435 } else {
3436 /*
3437 * We may have
3438 * segvn_rec
3439 * clean noun
3440 * in this se
3441 * writer loc
3442 * sleeping s
3443 * for more o
3444 */
3445 AS_CLRNOUNMAP
3446 mutex_exit(&
3447 }
3448 goto again;
3449 }

3451 as_setwatch(as);
3452 AS_LOCK_EXIT(as, &as->a_lock
3453 return (error);
3454 }

3456 /*
3457 * Setup all of the uninitialized wa
3458 */
3459 void
3460 as_setwatch(struct as *as)
3461 {
3462     struct watched_page *pwp;
3463     struct seg *seg;
3464     caddr_t vaddr;
3465     uint_t prot;
3466     int err, retrycnt;

3468     if (avl_numnodes(&as->a_wpage)
3469         return;

3471     ASSERT(AS_WRITE_HELD(as, &as

3473     for (pwp = avl_first(&as->a_w
3474         pwp = AVL_NEXT(&as->a_wp
3475         retrycnt = 0;
3476     retry:
3477         vaddr = pwp->wp_vaddr;
3478         if (pwp->wp_oprot != ~0)
3479             (seg = as_segat(a
3480             segop_getprot(seg
3481             SEGOP_GETPROT(seg
3482             continue;

3483         pwp->wp_oprot = prot;
3484         if (pwp->wp_read)
3485             prot &= ~(PROT_
3486             if (pwp->wp_write)
3487                 prot &= ~PROT_
3488             if (pwp->wp_exec)
3489                 prot &= ~PROT_
3490             if (!(pwp->wp_flags &
3491                 err = segop_i
3492                 err = SEGOP_I
3493                 if (err == IN
3494                     pwp->
3495                     ASSEE
3496                     retryp

```

31

```
new/usr/src/uts/common/vm/vm_as.c          32

3496             goto retry;
3497         }
3498     }
3499     pwp->wp_prot = prot;
3500 }
3501 }

3503 /*
3504  * Clear all of the watched pages in the address space.
3505  */
3506 void
3507 as_clearwatch(struct as *as)
3508 {
3509     struct watched_page *pwp;
3510     struct seg *seg;
3511     caddr_t vaddr;
3512     uint_t prot;
3513     int err, retrycnt;

3515 if (avl_numnodes(&as->a_wpage) == 0)
3516     return;

3518 ASSERT(AS_WRITE_HELD(as, &as->a_lock));

3520 for (pwp = avl_first(&as->a_wpage); pwp != NULL;
3521     pwp = AVL_NEXT(&as->a_wpage, pwp)) {
3522     retrycnt = 0;
3523     retry:
3524         vaddr = pwp->wp_vaddr;
3525         if (pwp->wp_oprot == 0 || /* not set up */
3526             (seg = as_segat(as, vaddr)) == NULL)
3527             continue;

3529         if ((prot = pwp->wp_oprot) != pwp->wp_prot) {
3530             err = segop_setprot(seg, vaddr, PAGESIZE, prot);
3531             err = SEGOP_SETPROT(seg, vaddr, PAGESIZE, prot);
3532             if (err == IE_RETRY) {
3533                 ASSERT(retrycnt == 0);
3534                 retrycnt++;
3535                 goto retry;
3536             }
3537             pwp->wp_oprot = 0;
3538             pwp->wp_prot = 0;
3539         }
3540     }

3542 /*
3543  * Force a new setup for all the watched pages in the range.
3544  */
3545 static void
3546 as_setwatchprot(struct as *as, caddr_t addr, size_t size, uint_t prot)
3547 {
3548     struct watched_page *pwp;
3549     struct watched_page tpw;
3550     caddr_t eaddr = addr + size;
3551     caddr_t vaddr;
3552     struct seg *seg;
3553     int err, retrycnt;
3554     uint_t wprot;
3555     avl_index_t where;

3557 if (avl_numnodes(&as->a_wpage) == 0)
3558     return;

3560 ASSERT(AS_WRITE_HELD(as, &as->a_lock));
```

```

3562     tpw.wp_vaddr = (caddr_t)((uintptr_t)addr & (uintptr_t)PAGEMASK);
3563     if ((pwp = avl_find(&as->a_wpage, &tpw, &where)) == NULL)
3564         pwp = avl_nearest(&as->a_wpage, where, AVL_AFTER);

3566     while (pwp != NULL && pwp->wp_vaddr < eaddr) {
3567         retrycnt = 0;
3568         vaddr = pwp->wp_vaddr;

3570         wprot = prot;
3571         if (pwp->wp_read)
3572             wprot &= ~(PROT_READ|PROT_WRITE|PROT_EXEC);
3573         if (pwp->wp_write)
3574             wprot &= ~PROT_WRITE;
3575         if (pwp->wp_exec)
3576             wprot &= ~(PROT_READ|PROT_WRITE|PROT_EXEC);
3577         if (!(pwp->wp_flags & WP_NOWATCH) && wprot != pwp->wp_oprot) {
3578             retry:
3579                 seg = as_segat(as, vaddr);
3580                 if (seg == NULL) {
3581                     panic("as_setwatchprot: no seg");
3582                     /*NOTREACHED*/
3583                 }
3584                 err = segop_setprot(seg, vaddr, PAGESIZE, wprot);
3585                 err = SEGOP_SETPROT(seg, vaddr, PAGESIZE, wprot);
3586                 if (err == IE_RETRY) {
3587                     ASSERT(retrycnt == 0);
3588                     retrycnt++;
3589                     goto retry;
3590                 }
3591         }
3592         pwp->wp_oprot = prot;
3593         pwp->wp_prot = wprot;

3594     }
3595     pwp = AVL_NEXT(&as->a_wpage, pwp);
3596 }

3598 /*
3599  * Clear all of the watched pages in the range.
3600 */
3601 static void
3602 as_clearwatchprot(struct as *as, caddr_t addr, size_t size)
3603 {
3604     caddr_t eaddr = addr + size;
3605     struct watched_page *pwp;
3606     struct watched_page tpw;
3607     uint_t prot;
3608     struct seg *seg;
3609     int err, retrycnt;
3610     avl_index_t where;

3612     if (avl_numnodes(&as->a_wpage) == 0)
3613         return;

3615     tpw.wp_vaddr = (caddr_t)((uintptr_t)addr & (uintptr_t)PAGEMASK);
3616     if ((pwp = avl_find(&as->a_wpage, &tpw, &where)) == NULL)
3617         pwp = avl_nearest(&as->a_wpage, where, AVL_AFTER);

3619     ASSERT(AS_WRITE_HELD(as, &as->a_lock));

3621     while (pwp != NULL && pwp->wp_vaddr < eaddr) {

3623         if ((prot = pwp->wp_oprot) != 0) {
3624             retrycnt = 0;

```

```

3626             if (prot != pwp->wp_prot) {
3627                 retry:
3628                     seg = as_segat(as, pwp->wp_vaddr);
3629                     if (seg == NULL)
3630                         continue;
3631                     err = segop_setprot(seg, pwp->wp_vaddr,
3632                         SEGOP_SETPROT(seg, pwp->wp_vaddr,
3633                             PAGESIZE, prot));
3634                     if (err == IE_RETRY) {
3635                         ASSERT(retrycnt == 0);
3636                         retrycnt++;
3637                         goto retry;
3638                     }
3639                 }
3640             pwp->wp_oprot = 0;
3641             pwp->wp_prot = 0;
3642         }
3644     }
3645 }
3646 }

_____unchanged_portion_omitted_____

3665 /*
3666  * return memory object ID
3667  */
3668 int
3669 as_getmemid(struct as *as, caddr_t addr, memid_t *memidp)
3670 {
3671     struct seg      *seg;
3672     int           sts;

3674     AS_LOCK_ENTER(as, &as->a_lock, RW_READER);
3675     seg = as_segat(as, addr);
3676     if (seg == NULL) {
3677         AS_LOCK_EXIT(as, &as->a_lock);
3678         return (EFAULT);
3679     }
3680     /*
3681      * catch old drivers which may not support getmemid
3682      */
3683     if (seg->s_ops->getmemid == NULL) {
3684         AS_LOCK_EXIT(as, &as->a_lock);
3685         return (ENODEV);
3686     }

3688     sts = segop_getmemid(seg, addr, memidp);
3688     sts = SEGOP_GETMEMID(seg, addr, memidp);

3690     AS_LOCK_EXIT(as, &as->a_lock);
3691     return (sts);
3692 }

_____unchanged_portion_omitted_____

```

new/usr/src/uts/common/vm/vm\_pvn.c

```
*****
31976 Tue Nov 24 09:34:49 2015
new/usr/src/uts/common/vm/vm_pvn.c
patch lower-case-segops
*****
1 /*
2  * CDDL HEADER START
3 *
4  * The contents of this file are subject to the terms of the
5  * Common Development and Distribution License (the "License").
6  * You may not use this file except in compliance with the License.
7 *
8  * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
9  * or http://www.opensolaris.org/os/licensing.
10 * See the License for the specific language governing permissions
11 and limitations under the License.
12 *
13 * When distributing Covered Code, include this CDDL HEADER in each
14 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
15 * If applicable, add the following below this CDDL HEADER, with the
16 * fields enclosed by brackets "[]" replaced with your own identifying
17 * information: Portions Copyright [yyyy] [name of copyright owner]
18 *
19 * CDDL HEADER END
20 */
21 /*
22 * Copyright (c) 1986, 2010, Oracle and/or its affiliates. All rights reserved.
23 * Copyright 2015 Nexenta Systems, Inc. All rights reserved.
24 */
25
26 /* Copyright (c) 1984, 1986, 1987, 1988, 1989 AT&T */
27 /* All Rights Reserved */
28
29 /*
30 * University Copyright- Copyright (c) 1982, 1986, 1988
31 * The Regents of the University of California
32 * All Rights Reserved
33 *
34 * University Acknowledgment- Portions of this document are derived from
35 * software developed by the University of California, Berkeley, and its
36 * contributors.
37 */
38
39 /*
40 * VM - paged vnode.
41 *
42 * This file supplies vm support for the vnode operations that deal with pages.
43 */
44 #include <sys/types.h>
45 #include <sys/t_lock.h>
46 #include <sys/param.h>
47 #include <sys/sysmacros.h>
48 #include <sys/sysm.h>
49 #include <sys/time.h>
50 #include <sys/buf.h>
51 #include <sys/vnode.h>
52 #include <sys/uio.h>
53 #include <sys/vmsystm.h>
54 #include <sys/mman.h>
55 #include <sys/vfs.h>
56 #include <sys/cred.h>
57 #include <sys/user.h>
58 #include <sys/kmem.h>
59 #include <sys/cmn_err.h>
60 #include <sys/debug.h>
61 #include <sys/cpuvar.h>
```

1

new/usr/src/uts/common/vm/vm\_pvn.c

```
62 #include <sys/vtrace.h>
63 #include <sys/tnf_probe.h>
64
65 #include <vm/hat.h>
66 #include <vm/as.h>
67 #include <vm/seg.h>
68 #include <vm/rm.h>
69 #include <vm/pvn.h>
70 #include <vm/page.h>
71 #include <vm/seg_map.h>
72 #include <vm/seg_kmem.h>
73 #include <sys/fs/swapnode.h>
74
75 int pvn_nofodklust = 0;
76 int pvn_write_noklust = 0;
77
78 uint_t pvn_vmodsort_supported = 0; /* set if HAT supports VMODSORT */
79 uint_t pvn_vmodsort_disable = 0; /* set in /etc/system to disable HAT */
80 /* support for vmodsort for testing */
81
82 static struct kmem_cache *marker_cache = NULL;
83
84 /*
85 * Find the largest contiguous block which contains 'addr' for file offset
86 * 'offset' in it while living within the file system block sizes ('vp_off'
87 * and 'vp_len') and the address space limits for which no pages currently
88 * exist and which map to consecutive file offsets.
89 */
90 page_t *
91 pvn_read_kluster(
92         struct vnode *vp,
93         u_offset_t off,
94         struct seg *seg,
95         caddr_t addr,
96         u_offset_t *offp,
97         size_t *lenp,
98         u_offset_t vp_off,
99         size_t vp_len,
100        int isra)
101 {
102         ssize_t deltaf, deltab;
103         page_t *pp;
104         page_t *plist = NULL;
105         spgcnt_t pagesavail;
106         u_offset_t vp_end;
107
108         ASSERT(off >= vp_off && off < vp_off + vp_len);
109
110         /*
111          * We only want to do klustering/read ahead if there
112          * is more than minfree pages currently available.
113          */
114         pagesavail = freemem - minfree;
115
116         if (pagesavail <= 0)
117                 if (isra)
118                         return ((page_t *)NULL); /* ra case - give up */
119                 else
120                         pagesavail = 1; /* must return a page */
121
122         /* We calculate in pages instead of bytes due to 32-bit overflows */
123         if (pagesavail < (spgcnt_t)btopr(vp_len)) {
124                 /*
125                  * Don't have enough free memory for the
126                  * max request, try sizing down vp request.
127                 */
128 }
```

2

```

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 TRACE_3(TR_FAC_VM, TR_PVN_READ_KLUSTER,
218         "pvn_read_kluster:seg %p addr %x isra %x",
219         seg, addr, isra);
220 return (plist);
221
222 } unchanged_portion_omitted

```

new/usr/src/uts/common/vm/vm\_seg.c

1

```
*****
54548 Tue Nov 24 09:34:49 2015
new/usr/src/uts/common/vm/vm_seg.c
patch lower-case-segops
*****
_____ unchanged_portion_omitted _____
1637 /*
1638  * Unmap a segment and free it from its associated address space.
1639  * This should be called by anybody who's finished with a whole segment's
1640  * mapping. Just calls segop_unmap() on the whole mapping . It is the
1640  * mapping. Just calls SEGOP_UNMAP() on the whole mapping . It is the
1641  * responsibility of the segment driver to unlink the the segment
1642  * from the address space, and to free public and private data structures
1643  * associated with the segment. (This is typically done by a call to
1644  * seg_free()).
1645 */
1646 void
1647 seg_unmap(struct seg *seg)
1648 {
1649 #ifdef DEBUG
1650     int ret;
1651 #endif /* DEBUG */
1652
1653     ASSERT(seg->s_as && AS_WRITE_HELD(seg->s_as, &seg->s_as->a_lock));
1654
1655     /* Shouldn't have called seg_unmap if mapping isn't yet established */
1656     ASSERT(seg->s_data != NULL);
1657
1658     /* Unmap the whole mapping */
1659 #ifdef DEBUG
1660     ret = segop_unmap(seg, seg->s_base, seg->s_size);
1661     ret = SEGOP_UNMAP(seg, seg->s_base, seg->s_size);
1662     ASSERT(ret == 0);
1663     (void) segop_unmap(seg, seg->s_base, seg->s_size);
1664     SEGOP_UNMAP(seg, seg->s_base, seg->s_size);
1665 #endif /* DEBUG */
1666
1667 /*
1668  * Free the segment from its associated as. This should only be called
1669  * if a mapping to the segment has not yet been established (e.g., if
1670  * an error occurs in the middle of doing an as_map when the segment
1671  * has already been partially set up) or if it has already been deleted
1672  * (e.g., from a segment driver unmap routine if the unmap applies to the
1673  * entire segment). If the mapping is currently set up then seg_unmap() should
1674  * be called instead.
1675 */
1676 void
1677 seg_free(struct seg *seg)
1678 {
1679     register struct as *as = seg->s_as;
1680     struct seg *tseg = as_removeseg(as, seg);
1681
1682     ASSERT(tseg == seg);
1683
1684     /*
1685      * If the segment private data field is NULL,
1686      * then segment driver is not attached yet.
1687      */
1688     if (seg->s_data != NULL)
1689         segop_free(seg);
1690     SEGOP_FREE(seg);
1691
1692     mutex_destroy(&seg->s_pmtx);
```

new/usr/src/uts/common/vm/vm\_seg.c

2

```
1692     ASSERT(seg->s_phead.p_lnext == &seg->s_phead);
1693     ASSERT(seg->s_phead.p_lprev == &seg->s_phead);
1694     kmem_cache_free(seg_cache, seg);
1695 }
_____ unchanged_portion_omitted _____
1857 /*
1858  * General not supported function for segop_inherit
1859  * General not supported function for SEGOP_INHERIT
1860 */
1861 int
1862 seg_inherit_notsup(struct seg *seg, caddr_t addr, size_t len, uint_t op)
1863 {
1864     return (ENOTSUP);
1865 }
_____ unchanged_portion_omitted _____
```